Synthesizing Specialized Sparse Tensor Accelerators for FPGAs via High-Level Functional Abstractions

Hamza Javed
McGill University
Montreal, Canada
hamza.javed2@mail.mcgill.ca

Christophe Dubach
McGill University & Mila
Montreal, Canada
christophe.dubach@mcgill.ca

***Abstract*—Sparsity is inherent in many applications such as machine learning and graph analytics. However, achieving high efficiency in sparse computations requires specialized hardware accelerators like FPGAs, as traditional accelerators typically cater to dense data. While high level synthesis enables the automatic generation of FPGA-based accelerators, generic solutions produced via C-based synthesis flows often demand extensive development time, leading designers to prioritize broad applicability over fine-grained structural specialization. Consequently, these accelerators fail to fully exploit FPGA's reconfigurablility, leaving substantial performance and efficiency gains untapped.**

This paper pushes the boundary by automatically generating specialized accelerators that match a given fixed sparse structure (*e.g.*, in static graph analytics and pruned neural networks). It achieves this by leveraging functional abstractions within high level synthesis, an approach that has already proven effective in automating the generation of specialized dense tensor accelerator. Tensor shapes are encoded directly in the type system and specialized primitives for irregular data are introduced. Together, these innovations enable a concise specification of sparse accelerators and drive advanced optimizations—including dynamic partitioning and vector sharding—to produce hardware precisely tailored to the sparsity pattern of the underlying tensors.

Compared to state-of-the-art generic accelerators (HiSparse, HiSpMV and GraphLily), the approach achieves up to a 2.8× improvement in bandwidth efficiency for sparse matrix computations and a 1.8× speedup on graph algorithms. Against the HLS4ML neural network acceleration framework, it achieves up to a 1.8× improvement in throughput with a 4× reduction in resource usage, enabling scaling to larger networks. These results establish this approach as a flexible, powerful, and rapid solution for designing high-performance specialized sparse accelerators.

***Index Terms*—Sparse Computation, Hardware Acceleration, Functional Languages, High Level Synthesis (HLS)**

I. Introduction

Sparse computations underpin applications in domains as diverse as scientific simulation, data analytics and machine learning [1]–[7]. Their defining characteristic is the presence of datasets heavily populated with zeros, which presents both challenges and opportunities for high performance. Crucially, many practical sparse workloads, *e.g.*, pruned neural networks and static graph kernels, exhibit a fixed sparsity pattern. The same matrices, index arrays, and row pointers are accessed many times over long periods. This represents an opportunity to specialize memory layouts, pipeline depths, and reduction trees to the structure (*e.g.*, non zero distribution) of the underlying sparse tensor data, exploiting the reconfigurabilty of FPGAs (Field Programmable Gate Arrays).

Most generic accelerators for sparse tensors are built via months of manual optimization [8]–[21]. To justify development effort, designers broaden each accelerator's applicability and in doing so dilute the fine-grained structural specialization that unlocks peak performance. While C-based HLS (High Level Synthesis) tools can be used to facilitate the design of such accelerators, they assume dense regular arrays and have difficulties achieving high-performance in the presence of irregular control. As a result, achieving high performance and efficiency still requires deep hardware expertise.

Functional HLS approaches [22]–[25] have demonstrated the power of rich type systems for generating highly-specialized dense tensor accelerators in hardware, yet they stop at dense data. This paper extends this approach to sparsity, presenting a functional methodology that produces fully specialized accelerators for fixed sparsity patterns.

This work makes three main contributions. First, it introduces algorithmic and hardware primitives for irregular data, by encoding the data shape using dependent types. This enables the concise expression and automatic generation of accelerators tailored to sparse structures. Second, it leverages the expressiveness to develop an automated partitioning strategy that adapts partition sizes to the actual nonzero distribution, ensuring balanced parallelism. Third, it implements a vector sharding mechanism that extracts only the necessary segments of a vector, reducing on-chip memory usage and addressing complexity. Together, these innovations enable the generation of fully specialized accelerators for sparse tensor algorithms and neural networks, resulting in high-performance.

Experimental evaluations on an Arria 10 FPGA and simulations with 250 GBPS HBM (High Bandwidth Memory) show that the proposed approach drastically improves performance for sparse workloads over prior work. In SpMV (Sparse matrix-vector multiplication), the approach attains a 1.7× gain in throughput over state-of-the-art accelerator HiSpMV-16 [21] and outperforms the hand-tuned accelerator HiSparse [19] by 2.8× in bandwidth efficiency. For graph algorithms, the approach boosts throughput and bandwidth efficiency by up to 1.8× and 2.1×, respectively, compared to GraphLily [20]. Moreover, in neural network inference, this approach surpasses HLS4ML [26] in resource utilization and throughput by 4× and 1.8×, enabling scaling to larger networks where HLS4ML fails to yield viable designs.

In summary, this paper makes the following contributions:

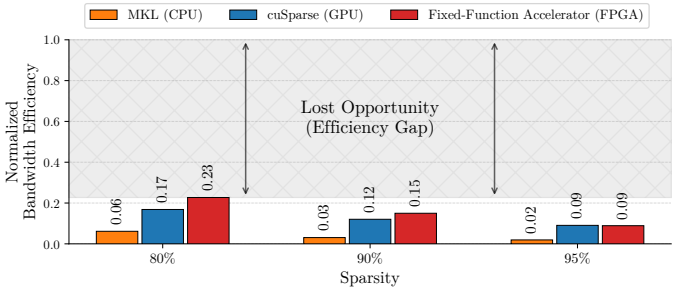

Fig. 1: Normalized Bandwidth Efficiency ($\frac{\text{Bandwidth}_{\text{effective}}}{\text{Bandwidth}_{\text{peak}}}$) of various accelerators compared to the theoretical maximum, evaluated using SpMV on a 512×33k matrix with varying sparsity.

- Design of functional primitives for dependent types that enable efficient handling of irregular data within hardware;
- Integration of automated partitioning, scheduling, and vector sharding techniques to adapt to nonzero distributions at design time, ensuring balanced parallelization while minimizing memory overhead;
- Generation and evaluation of specialized accelerators for sparse computations, graph algorithms and neural networks.

II. Motivation

FPGAs are a substrate of choice for accelerating sparse tensor operations, like pruned neural networks or static graph kernels. These applications often have a fixed-sparsity pattern, where the structure is static over long execution periods. The FPGA's ability for reconfiguration allows designers—in theory—to tailor the accelerator to the exact sparsity pattern far more aggressively than in any ASIC design. However, current design flows fall short of that potential.

State of the art generic sparse tensor engines [8]–[21] are written in C based HLS and span tens of thousands of lines refined through months of manual tuning. Specializing an accelerator for a new structure requires another labor intensive development cycle. As a result, designers tend to ignore such specialization, leaving peak performance untapped even when the sparsity pattern is fixed.

In practical deployments, memory bandwidth remains the primary bottleneck in sparse accelerators [27]. Figure 1 compares the normalized bandwidth efficiency, across a CPU, a GPU and a generic sparse tensor accelerator (HiSparse [19]). This device-independent metric reveals how close each architecture comes to the theoretical limit. The large efficiency gap shows that even expert-crafted designs fall well short of peak efficiency. A key reason for this gap is that these generic accelerators cannot tailor their dataflow and memory access patterns to a specific, static sparsity structure.

An ideal flow must therefore exploit any fixed structure in the sparse tensor, correct imbalance automatically, let developers specify algorithms in just a few high level lines, and demand little hardware expertise while still approaching the theoretical limit. To realize this vision, this paper introduces *high-level functional abstractions* that enable concise expression of fixed-structure sparse algorithms. The compiler encodes every matrix shape in the type system, identifies opportunities such as non-zero distribution, and produces a fully specialized sparse accelerator thus maximizing efficiency.

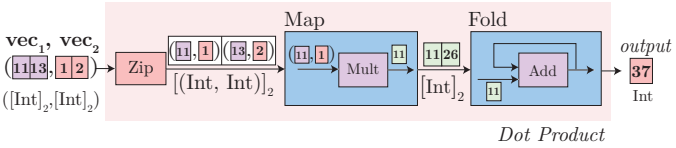

Fig. 2: Architectural representation of DotProduct generated for vectors of type $[Int]_2$.

III. Background and Overview

A. Functional HLS

This work builds on a functional accelerator design methodology similar to Lift-HLS [22], Aetherling [24], or SHIR [25], where a high-level algorithmic description is gradually transformed into hardware-ready code. At the outset, program logic is specified using a typed functional syntax, avoiding hardware details such as pipelining or parallelism. As compilation passes through multiple IRs (architectural & memory), key concepts like streams, vectors, and memory are introduced.

Streams represent data arriving sequentially, enabling pipelined execution in which each data element moves continuously through the hardware. Vectors bundle multiple data items for parallel processing, allowing higher throughput when sufficient hardware resources are available. Meanwhile, memory primitives define how on-chip block RAM and off-chip storage (*e.g.*, DRAM) are used, and they specify data access patterns so that the compiler can automatically optimize memory controllers and data flows. By the final stage, the design is expressed as RTL (Register Transfer Language) code suitable for FPGA or ASIC (Application-Specific Integrated Circuit) synthesis. A central strength of this approach lies in functional primitives. For instance, consider Map :

$$N^{NatT} \mapsto T^{DataT} \mapsto U^{DataT} \mapsto (T \to U) \to [T]_N \to [U]_N$$

where N^{NatT} indicates that N is of type NatT (Natural Number). $x \mapsto$ denotes a type parameter x (akin to C++ templates) while $y \to$ denotes an argument y. Overall, the definition of Map reads as: given a type parameter N (length of an array), then given two datatypes T and U, then given arguments, a function $(T \to U)$ and an input array $[T]_N$ (an array of N elements of type T), it produces an output array $[U]_N$. In hardware, this can translate into $\texttt{map}_{\texttt{stream}}$, where each element arrives sequentially and pipelining is facilitated, or $\texttt{map}_{\texttt{vector}}$, where all elements are processed in parallel by instantiating multiple copies of the internal $(T \to U)$ function.

A single-line functional program demonstrates the power of this abstraction by representing a dot product concisely:

```
1 def DotProduct(row: ([DataT]_N, [DataT]_N)) => {
2    Fold(+, Map(λ elems => *, Zip(row)))}
```

Conceptually, `Zip(row)` pairs corresponding elements from two input arrays, and `Map(λ elems => *)` multiplies each pair to form an intermediate array. The `Fold(+,_)` then accumulates these products by summation. The corresponding hardware architecture is shown in Fig. 2.

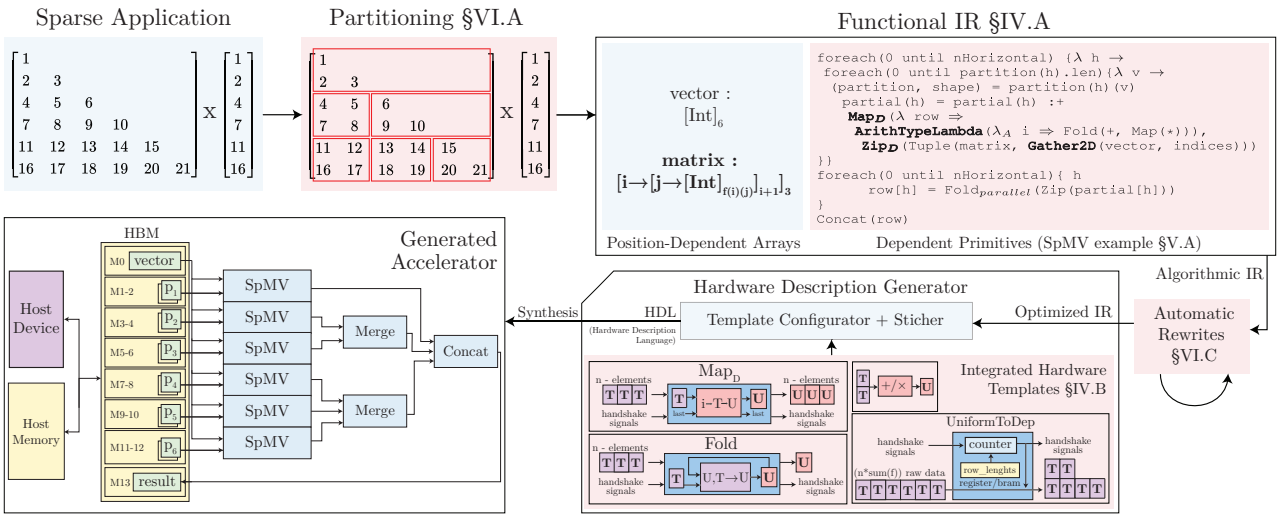

Fig. 3: End to end compilation flow of the proposed functional HLS approach, illustrated with a sparse matrix-vector multiply (SpMV). A high level functional program (top-left) is *partitioned*, encoded with *position-dependent arrays*, transformed through *dependent primitives* and automatic rewrites (top–right), and finally lowered to hardware using the integrated hardware templates that synthesize into a deeply pipelined accelerator design (bottom-left); internals for SpMV will be shown in Fig. 7. Elements shaded in pink mark the novel contributions.

B. Expressing Sparse Tensors

In many machine learning and scientific computing tasks, large datasets are represented as tensors, multidimensional arrays used in operations like convolutions and matrix multiplications. When most values are zero, the data is considered *sparse*, and exploiting this sparsity can greatly reduce both memory usage and computation. A common example is SpMV, where a sparse matrix multiplies a dense vector to yield the output. For example, a 3×3 lower triangular matrix contains about 33% zeros. A dense representation processes all nine elements, while a sparse representation compresses the matrix by omitting zeros, resulting in rows of varying lengths, something static arrays cannot easily capture.

Recent work [28], [29] shows that *position-dependent arrays*, a subset of dependent types, can represent irregular shapes in high-level functional languages. The key idea is to let the length of an inner array vary with a parameter such as the row index. For example, in a 3×3 lower triangular matrix, the i-th row has $i + 1$ elements. Formally, this matrix can be represented as $[row \mapsto [Int]_{row+1}]_3$, which explicitly encodes the varying row lengths in the type system.

This enables the encoding of a sparse matrix as value, column index pairs with row information built into the type, thus facilitating specialization across the row lengths. For example a random 3×3 sparse matrix can be represented as $[row \mapsto [(\text{val} : Float, \text{col\_idx} : Int)]_{f(row)}]_3$, where f returns the length of each row *e.g.*, $f = [2, 1, 3]$.

C. Overview

An overview of the proposed approach is shown in Fig. 3. The core contribution centers on *expressing* irregular sparse computations, while compilation itself relies on well established literature. The pipeline begins by *partitioning* the sparse application, then encodes the data with position-dependent arrays and represents the algorithm using *dependent primitives*. These abstractions efficiently handle irregular data within the functional IR. The compiler then applies *automatic rewrites* and lowers the optimized IR to hardware by instantiating pre-embedded circuit blueprints (*i.e.*, HDL (Hardware Description Language) templates) with parameters such as bit-width and buffer-depth. This process automatically generates a fully specialized accelerator optimized for the given workload, requiring no hardware expertise from the user. In the next section we will see the definition and hardware implementation of these dependent primitives.

IV. Sparse IR and Hardware Implementation

A. Dependent Primitives

To enable efficient processing of position-dependent arrays, the proposed approach introduces a set of computational primitives that extend conventional functional operators such as `map` and `zip` to handle position-dependent arrays, thereby enabling the compiler to capture and manipulate structural information at compile time.

Map_D. Irregular data sets such as triangular matrices or variable-length feature vectors, cannot be traversed with the ordinary $\text{map} : (T \to U) \to [T]_N \to [U]_N$ because the element type itself changes with the outer index. Attempting to shoehorn such collections into a dense `map` forces programmers to *pad* every row to a common length, which destroys the performance benefits of sparsity. Map_D lifts the familiar map abstraction to position-dependent arrays, allowing an inner transformation to be specialized for each row while keeping the outer pipeline streaming.

$$\text{Map}_D : \quad N^{NatT} \mapsto f_T^{NatT \to DataT} \mapsto f_U^{NatT \to DataT} \mapsto (i \to f_T(i) \to f_U(i)) \to [n_1 \mapsto f_T(n_1)]_N \to [n_2 \mapsto f_U(n_2)]_N$$

The outer length N is fixed, so no dynamic scheduling is required. Two type functions f_T and f_U specify each row's

source and result shapes as functions of the row index i. Although rows with different indices are indistinguishable at run time, their distinct static types enable the compiler to unroll or pipeline each inner loop with exact trip counts, size FIFOs and scratchpads from type-level naturals, and to prove the legality of transformations such as interchange or fusion at compile time.

Given a triangular matrix *matrix* : $[i \mapsto [Float]_{i+1}]_M$ and a scalar $x : Float$, a scaling function can be defined as:

```
scale_bad(matrix, x) =
   Map_D(λ irow ⇒
      Map(λ elem ⇒ elem*x, irow)
   , matrix)
```

This expression fails to type check because the inner Map has type $T \to U$, while Map_D requires $i \to T \to U$. Introducing a variant such as $\mathsf{Map}_{\mathrm{Param}}$ would resolve the type error but would necessitate creating a duplicate for each dense primitive.

ArithTypeLambda. To address this limitation, ArithTypeLambda is introduced. It enables dense primitives to be used within dependent contexts by wrapping a dense function ($T \to U$) to introduce an artificial arithmetic (natural numbers) dependency, resulting in a function of type $(i \to T \to U)$. In essence, it parameterizes a generic transformation over an index variable i as follows:

$$\text{ArithTypeLambda} : \quad i^{NatT} \mapsto T^{DataT} \mapsto U^{DataT} \mapsto (T \to U) \to (i \to T \to U)$$

Given any dense body g, the primitive produces a dependent version that simply ignores the index. Leveraging this new primitive, one can rewrite the scale function from earlier:

```
scale(matrix, x) =
   Map_D(λ irow ⇒
      ArithTypeLambda(λ_A i ⇒
         Map(λ elem ⇒ elem*x, irow))
   , matrix)
```

Hence `ArithTypeLambda` retrofits existing dense primitives for use inside dependent pipelines at zero hardware cost, avoiding a proliferation of special-purpose "_parameterized" variants and preserving a lean, easily-optimized core language.

Zip$_D$. Another key primitive is $\texttt{Zip}_D$. Whereas a standard `zip` pairs two arrays element by element, $\texttt{Zip}_D$ merges two position-dependent arrays of length N:

$$\text{Zip}_D : \quad N^{NatT} \mapsto f^{NatT \to DataT} \mapsto h^{NatT \to DataT} \mapsto \Big([\, n_1 \mapsto f(n_1) \,]_N, [\, n_2 \mapsto h(n_2) \,]_N \Big) \to [\, n_3 \mapsto \big(f(n_3), h(n_3)\big) \,]_N$$

Zip_D couples two position dependent streams element wise while preserving positional information. No constraint is placed on the relationship between f and h beyond agreement on outer length, which makes the primitive well suited to formats where meta data and payload have different per row shapes such as the triple of CSR arrays .

DepToUniform and UniformToDep. To manage position dependence within otherwise dense hardware dataflows, two complementary primitives are introduced. `DepToUniform` collapses a position-dependent array into a uniform array when each sub-array's type is effectively constant:

$$\text{DepToUniform} : \quad N^{NatT} \mapsto f_T^{NatT \to DataT} \to [\, n \mapsto f_T(n) \,]_N \xrightarrow{\forall n < N \;\; f_T(n) = f_T(0)} [\, f_T(0) \,]_N$$

The constraint $\forall n < N; f_T(n) = f_T(0)$ ensures that every sub-array in the input is of the same type, effectively making the array uniform. This uniformity is critical for safely collapsing the position-dependent array into a dense array.

Conversely, `UniformToDep` recreates position-dependent structure from a uniform array. This is essential as data from external memory arrives as a flat-stream and must be split on-chip to introduce position-dependence:

$$\text{UniformToDep} : \quad N^{NatT} \mapsto f^{NatT \to NatT} \mapsto T^{DataT} \to [T]_{\sum_{i=0}^{N} f(i)} \to [\, n \mapsto [T]_{f(i)} \,]_N$$

i.e., it introduces positional information by splitting the uniform array into sub-arrays of lengths $[f(0), f(1), \ldots, f(N-1)]$, thereby generating the dependent structure required for processing irregular data.

Collectively, these primitives allow sparse operations to be expressed functionally. By embedding positional information in the type system, the approach generates pipelines that precisely handle irregular data while leveraging dense primitives when applicable. This provides the flexibility required for the representation of complex sparse computations.

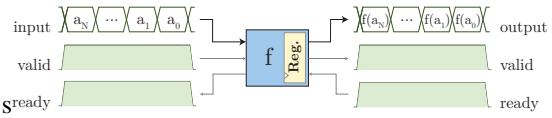

Fig. 4: Architectural representation of a pipelined function.

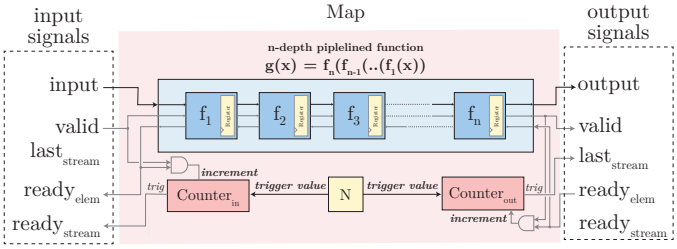

Fig. 5: Architectural representation of a counter-based `Map`.

B. Hardware Implementation of Dependent Primitives

This section outlines the hardware realization of position-dependent type primitives, focusing on the `map` operator. Fig. 4 shows a fundamental pipelined function that forms the building block of the design: it processes one input per cycle and outputs the result in the next. An internal pipeline register holds the output of f, enabling new inputs to be processed every clock cycle. The design uses a standard valid/ready/last handshake protocol between components: *valid* indicates data availability, *ready* signals the receiver is prepared, and *last* marks the end of an element or stream.

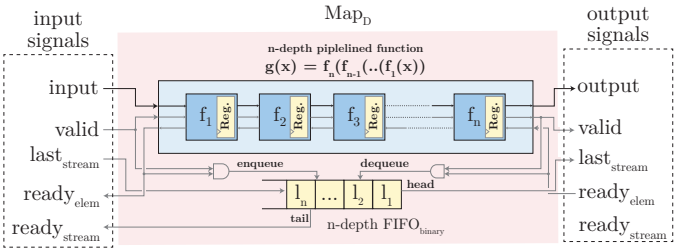

Fig. 6: Architectural representation of a FIFO-based $\texttt{Map}_D$.

Building on the pipelined function, Figure 5 shows the architecture of a standard functional `map` [25] for a 2D dense matrix. This design uses two compile-time counters to track processed elements. For dense matrices, each row has a fixed length and the map primitive guarantees that the number of input elements equals the number of output elements ($n_{\text{elem\_in}} = n_{\text{elem\_out}} = \text{constant}$), enabling the use of fixed counters. In hardware, the input counter increments with each element consumed and, upon reaching the row length, generates a ready signal to fetch the next row. Similarly, the output counter triggers a $last_{stream}$ signal once it reaches the predefined count, signaling the end of the current row. In contrast, position-dependent arrays can have varying sub-stream lengths, which fixed counters cannot accommodate.

To address this limitation, a FIFO (First-In, First-Out)-based design is proposed, as shown in Fig. 6. Dependent arrays generated by `UniformToDep` already embed handshake signals that convey each row's length, so no counter is needed. Instead, a binary FIFO buffer configured to match the pipeline depth of the internal consuming function, dynamically stores and releases these signals in first-in, first-out order. In the worst case of rows composed of single elements (length 1), the FIFO must hold one handshake token per pipeline stage, limiting the required depth to at most N. For most sparse kernels discussed in this paper, N typically ranges between 2 and 20, so the memory cost of the FIFO remains very low. In practice, the compiler automatically infers the pipeline depth of each $\texttt{map}_D$ instantiation and configures the FIFO accordingly, thereby preserving a fully pipelined architecture.

Similarly, the hardware templates for other primitives are designed with similar considerations in mind. For example, the `ZipD` operator merges two dependent streams by integrating their handshake signals to synchronize corresponding sub-arrays. Conversely, the `UniformToDep` operator converts a regular stream into a dependent one by generating "last" signals using a counter to mark sub-array boundaries; in hardware, it splits a continuous stream using a secondary stream that specifies each row's length, thus producing the appropriate $last_{stream}$ signal. Finally, when the dependent stream is identical to its dense counterpart (i.e., all sub-arrays share the same length), the `DepToUniform` operator acts solely as a type converter. The functional HLS compiler automatically lowers each primitive to its specialized hardware template, which are then composed to form a complete accelerator.

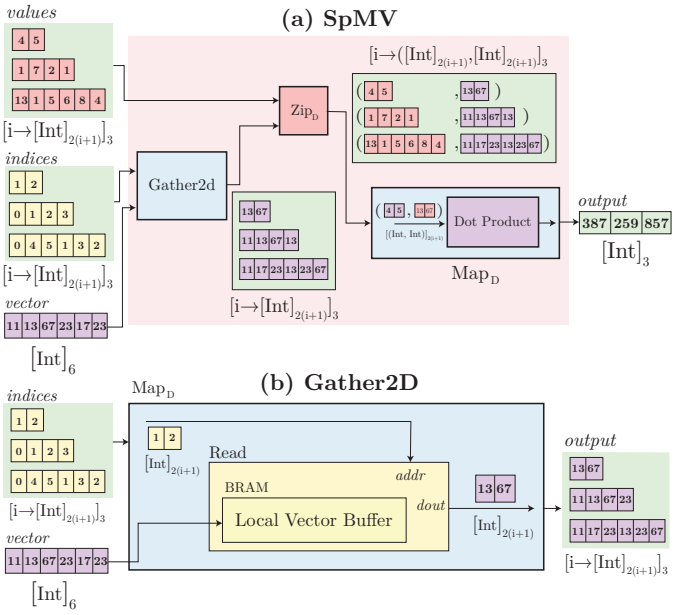

Fig. 7: Architectural representation of (a) **SpMV** and (b) **Gather2D** expressions generated for a matrix of type $[i \mapsto [Int]_{2(i+1)}]_3$ and vector of type $[Int]_3$.

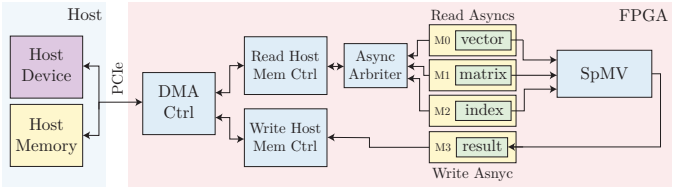

Fig. 8: Sequential SpMV (§V-A) accelerator generated by the compiler. Internals for the SpMV block are shown in Fig. 7.

V. Expressing Sparse Kernels

This section showcases the extended IR (Intermediate Representation)'s versatility with examples of sparse kernels implemented using the proposed methodology.

A. Sparse Matrix-Vector Multiplication (SpMV)

Using the position-dependent array primitives listed in the previous section, sequential SpMV can be implemented as:

```
1 Map_D(λ row ⇒
2       ArithTypeLambda(λ_A i ⇒ DotProduct(row)),
3          Zip_D(Tuple(
4             DepInput("matrix", N, f),
5             Gather2D(Input("vector", M),
6                   DepInput("idx", N, f))))
```

where $\texttt{Gather}_{2D}$ is defined as:

```
1 def Gather2D(input: [DataT]_N, idx: [i↦g(i)^StreamT]_M) => {
2    Map_D(λ row ⇒ Read(Buffer(input), row), idx)
3 }
```

$\texttt{Gather}_{2D}$ extracts the required vector elements based on index stream. $\texttt{Zip}_D$ synchronizes corresponding rows from the matrix and gathered vector, while $\texttt{Map}_D$ applies the dot product function to each row as shown in Fig. 7.

Compilation. The compiler first performs type checking of the full expression tree. Each high-level algorithmic primitive

is then lowered into an architectural counterpart, and the resulting concrete types drive the calculation of bus widths and handshake signals. Rewrite rules fuse adjacent operators, insert just enough buffering for throughput, and remove redundant logic. The refined graph feeds the HDL backend, which instantiates the internal templates and configures each with the widths, depths, and handshake wiring computed earlier. Each architectural primitive is matched to a built-in hardware template, so the user never writes any RTL. For memory-facing nodes (`Input`, `DepInput`, `Store`, *etc.*), the backend reuses standard dense memory controllers and automatically wraps dependent ports with `UniformToDep` or `DepToUniform`. When several controllers are present, the compiler also inserts the arbitration logic that multiplexes their ports. Finally, the configured templates are stitched, lowered to synthesizer-ready RTL, and passed to the vendor flow to produce the bit-stream for the specialized SpMV accelerator shown in Fig. 8.

B. PageRank

Similarly, iterative PageRank can be written as:

```
fold(λ (rank, _) ⇒
    val prod = SpMV(matrix, indices, rank)
    val scaled = Map(λ elem ⇒ Mul2(alpha, elem), prod)
    Map(Add2(), Zip(Tuple(scaled, teleport)))
, Counter(maxIter))
```

where maxIter is a constant, while matrix, indices and rank are read from memory using the input primtives shown in the SpMV example (ommited here for brevity). This design employs `SpMV` to iteratively update the rank.

These examples illustrate how the approach concisely expresses complex sparse computations while enabling seamless mapping to efficient hardware. Beyond these, the IR can represent a broad range of sparse tensor and graph algorithms some of which are evaluated later.

VI. Architectural Exploration and Optimizations

The proposed approach goes beyond traditional hardware tweaks. By leveraging a functional paradigm, it enables diverse accelerator configurations with distinct ordering, processing, and parallelization schemes tailored to specific sparsity and workload characteristics. This section highlights the advanced optimizations enabled by its expressiveness.

A. Expressing Parallelization

The proposed approach supports a variety of parallelization strategies. Three such strategies are shown in Fig. 9.

1) Row-Level Parallelization

A common way to exploit fine-grained parallelism in SpMV is to replicate multiple multiply and accumulate (MAC) units so that each row is processed concurrently (as shown in Fig. 9(a)). Because each sub-array (or row) in a sparse matrix can have anywhere from 0 to N nonzero elements, the effective parallelism depends heavily on the sparsity pattern. When a row has fewer nonzeros than the parallel factor, many MAC units remain idle, leading to underutilized resources. For SpMV, such a strategy can implemented by updating the DotProduct expression from earlier (Section III-A):

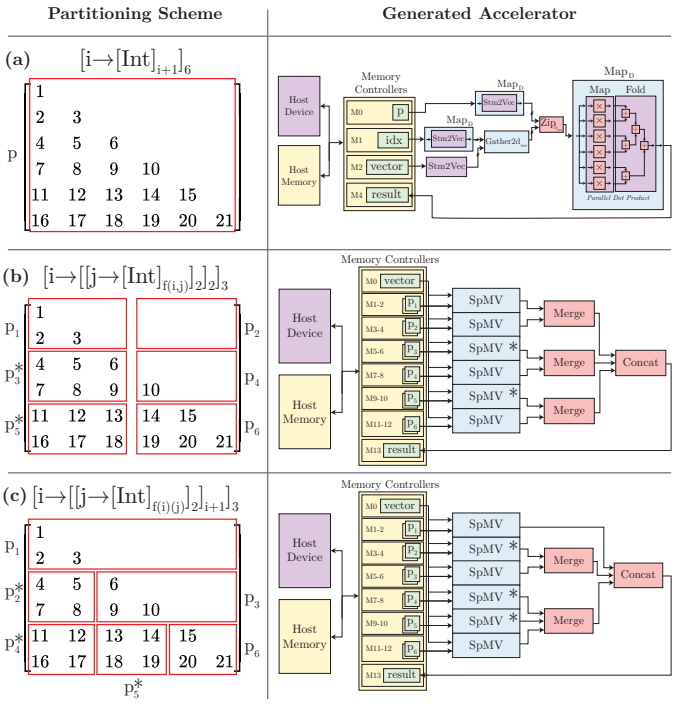

Fig. 9: Three parallelizing strategies for SpMV in hardware: (a) **Row-level parallelization**, multiple MAC units process the same row. (b) **Homogeneous partitioning**, matrix is split into fixed-size blocks for uniform parallelism. (c) **Heterogeneous partitioning**, block sizes adapt to nonzero distributions. $*$ highlights potential optimizations (explored in §VI-B)

```
DotProduct = Fold_parallel(+ ,
                Map_parallel(λ elem → ×, Zip(row)))
```

2) Homogeneous Partitioning

Figure 9(b) divides the matrix uniformly into fixed-size sub-matrices arranged in a regular $M \times N$ grid, invoking a sequential SpMV kernel for each partition. This approach simplifies control and scheduling because all sub-matrices have identical dimensions and can be processed by similar hardware modules. However, in sparse matrices with irregular data distributions—such as a lower triangular matrix—many blocks (*e.g.*, those in the upper-right region) may contain little or no data, causing load imbalance and inefficient utilization of resources. For SpMV, such a strategy can using the following pseudo-expression:

```
foreach(0 until nHorizontal){λ h →
  foreach(0 until nVertical){λ v →
    let p = partition[h, v] in
      partial[h,v] =
        SpMV(DepInput(p, p.shape), vector)
}}
foreach(0 until nHorizontal){ h
      row[h] = Fold_parallel(Zip(partial[h,_]))
}
Concat(row)
```

Here, the `foreach` is a macro and is not an inherent part of the IR. It simply duplicates the inner IR expression for every input. The same applies for *let* which is used for conciseness. For brevity, the input primitives are omitted in the expression.

Algorithm 1 Heterogeneous 2-D partitioning

Require: sparse matrix A; row slices H; total blocks N; tolerance ε
Ensure: list $\mathcal{P}$ of exactly N balanced submatrices

function SPLITBYNNZ(M, k, ε)
  return k contiguous ranges whose nnz totals differ by $\leq \varepsilon$
end function
function PART2D(M, h, v, ε)
  $\mathcal{P} \leftarrow [\,]$
  for all $R \in$ SPLITBYNNZ(M, h, ε) **do**
    $n \leftarrow \max\big(1,\ \text{round}(v \cdot nnz(R)/nnz(M))\big)$
    for all $C \in$ SPLITBYNNZ($M[R,:]^{\mathsf{T}}, n, \varepsilon$) **do**
      $\mathcal{P}$.push$\big((M[R,:]^{\mathsf{T}}[C,:])^{\mathsf{T}}\big)$
    end for
  end for
  return $\mathcal{P}$
end function
function ADAPTIVE2D(A, H, N, ε)
  $V \leftarrow N/H$ ▷ implicit column-slice count
  $P_r \leftarrow$ PART2D(A, H, V, ε)
  $P_c^{\mathsf{T}} \leftarrow$ PART2D($A^{\mathsf{T}}, V, H, \varepsilon$)
  if IMBALANCE(P_r) $\leq$ IMBALANCE(P_c^{T}) **then**
    return P_r
  else
    return $\{\, X^{\mathsf{T}} \mid X \in P_c^{\mathsf{T}} \}$
  end if
end function

3) Heterogeneous Partitioning

Figure 9(c) provides a more adaptive strategy by aligning partitions along one dimension (*e.g.*, rows) while allowing the block size in the other dimension (*e.g.*, columns) to vary according to actual nonzero counts. In such a scheme (Algorithm 1), the matrix is partitioned using both a row-first and a column-first approach, and a measurement function evaluates the load imbalance for each set of partitions. The approach yielding the lower imbalance is then chosen. Moreover, the use of dependent types allows the compiler to specialize each sequential SpMV unit to the unique characteristics of its corresponding partition, ensuring that the generated hardware is optimally tailored to the workload. This technique therefore achieves more balanced workloads than fixed homogeneous grids. For SpMV, such a strategy can be implemented as:

```
foreach(0 until nHorizontal) {λ h →
  foreach(0 until partition(h).len){λ v →
    let (partition, shape) = partition(h)(v) in
      partial(h) = partial(h) :+
        SpMV(Input(partition, shape), vector)
}} // rest of the expr. is same as the homogenous one
```

Here, partition is also a position-dependent array, and thus `partition(h)(v)` provides access to a specific partition.

B. Vector Sharding

Partitioning the matrix necessitates a corresponding distribution of the input vector. Since replicating the full vector for each parallel unit is infeasible due to on-chip memory constraints, vector sharding provides each unit with only its required segment using the following primitive:

$$\text{Select}: \quad N^{NatT} \mapsto T^{DataT} \mapsto start^{NatT} \mapsto end^{NatT} \to [T]_N \to [T]_{start\,:\,end}$$

The Select primitive extracts a sub-vector from a global vector $[T]_N$ using given start and end indices, returning the elements from the start up to (but not including) the end index. This sharded sub-vector is then provided to the corresponding processing unit. For example, in Heterogeneously Partitioned SpMV, vector sharding can be integrated as:

```
// ... code omitted from previous listing ...
   (partition, shape) = partition(h)(v)
    partial(h) = partial(h) :+
      SpMV(Input(partition, shape),
       Read(Buffer(Select(vector, p.start, p.end)))
// ...
```

By isolating only the necessary elements, vector sharding provides two main benefits. First, it offers memory savings by copying only the relevant portion of the vector for each partition, thereby avoiding the overhead of full vector replication. Second, it reduces addressing bitwidth by re-indexing the sharded sub-vector to start at 0, which simplifies addressing logic and lowers the bitwidth required to index sparse matrix elements; a crucial advantage for resource-efficient designs.

C. Automatic Rewrite Rules

Rewrite rules express sound equivalences on the IR: each rule matches a local IR pattern and replaces it with an alternative that preserves semantics while exposing new optimization opportunities. During compilation the rewriter scans the IR, applies the first rule whose left-hand pattern matches, and repeats the process until no further rules fire. The result is a program that behaves identically but can unlock higher performance.

Addressing Imbalanced Workloads. This rewrite approach enables further optimizations. For example, in Fig. 9(c), even if the total element count across partitions is balanced, an uneven per-row distribution may lead to variable output rates and stalling at the merge unit. To address this, a FIFO is inserted before the merge unit (i.e., within the fold operation in IR) using the following rule:

$$\text{Fold}\Big(\text{Zip}\Big(\text{SpMV}(\ldots(p_1)),\ \ldots,\ \text{SpMV}(\ldots(p_N))\Big)\Big) \xrightarrow[\text{Depth}=D_{\text{fifo}}]{}$$
$$\text{Fold}\Big(\text{Zip}\Big(\text{FIFO}\Big(\text{SpMV}(\ldots(p_1))\Big),\ \ldots,\ \text{FIFO}\Big(\text{SpMV}(\ldots(p_N))\Big)\Big)\Big)$$

The maximum FIFO depth D_{fifo} is determined by the imbalance in nonzero processing across partitions $(p_1, \ldots, p_n)$. For partition k, let $r_{k,i}$ denote the number of nonzeros in its ith row, and define the cumulative count as $P_k(i) = \sum_{j=1}^{i} r_{k,j}$. Then, the required FIFO depth is given by:

$$D_{\text{fifo}} = \max_{1 \le i \le n} \Big(\max_{1 \le k \le N} P_k(i) - \min_{1 \le k \le N} P_k(i) \Big),$$

where N is the number of partitions and n corresponds to rows per partition. This worst-case imbalance measure ensures that the FIFO is sufficiently deep to buffer extra elements, thus reducing stalls and improving throughput. A similar rule is also added for the `Concat` operation within Fig. 9(c).

VII. EVALUATION

A. Experimental Setup

This work builds on SHIR [25] by extending it with position-dependent types and *dependent primitives* along with their respective *hardware templates* and custom *rewrite rules*

TABLE I: Overview of matrix datasets: balanced datasets are listed in the first half, while imbalanced datasets appear in the second half.

Dataset	Size	Density	nnz	nnz/row
`transformer-XX`	33k × 512	5–50%	$8.4\text{-}84.4 \times 10^3$	$25.6 - 256$
`mouse-gene`	45K × 45K	1.42%	2.9×10^7	639.0
`googleplus`	108K × 108K	1.20×10^{-2}	1.4×10^8	1,296.0
`ogbl-ppa`	576K × 576K	1.28×10^{-4}	4.2×10^7	73.7
`hollywood`	1,069K × 1,069K	9.85×10^{-5}	1.1×10^8	105.3
`poli_large`	16K × 16K	1.36×10^{-4}	33,033	2.1
`hangGlider_3`	10K × 10K	8.81×10^{-4}	92,703	9.0
`lowThrust_7`	17K × 17K	7.00×10^{-4}	10,260	12.1
`trans5`	117K × 117K	5.49×10^{-5}	749,800	6.4

balanced: ($nnz/row > 20$), imbalanced: ($nnz/row \ll 20$)

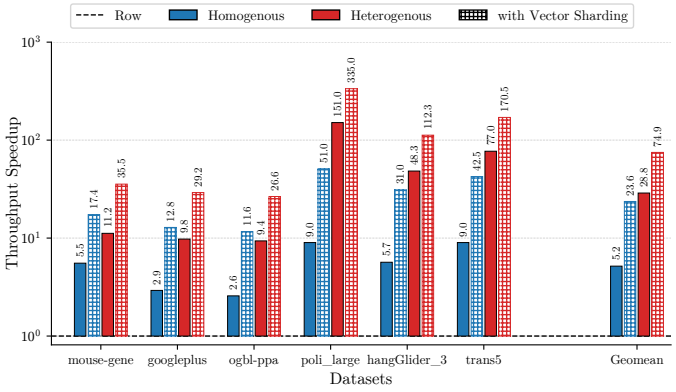

Fig. 10: SpMV throughput speedup for parallelization schemes with & without vector sharding compared against the Row baseline. Simulated with 16-channel 250 GBPS HBM. Higher is better.

related to sparse flows. The enhanced SHIR optimizes data access, integrates memory controllers for transferring data between HBM/host RAM and on-chip storage, and produces a custom VHDL accelerator for each benchmark. The generated VHDL is synthesized using Quartus Prime 19.2, and the design achieves 200 MHz for on-board execution on an Intel Arria 10 GX FPGA. The FPGA is interfaced with an Intel Xeon host via a PCIe Gen3×8 link, with benchmark data read directly from host RAM at 6.4 GBPS [25]. The HBM results are produced with a simulation of a Stratix 10 device using Questa 24.3.0 to emulate a 16-channel, 250 GBPS configuration, where data is stored on HBM. In all cases, VHDL code is generated in under one minute, and results are reported only when they fit the target chip. Finally, across all designs, the peak power consumption is 45 W, as reported by the Quartus tool.

Benchmark Datasets and Inputs. All SpMV and graph benchmarks utilize sparse matrices from SuiteSparse [30], listed in Table I. Matrices were selected to align with previous studies [19], [21] and grouped as balanced ($nnz/row > 20$) or imbalanced ($nnz/row \ll 20$). The `transformer-XX` matrices represent a feed-forward layer from a transformer pruned to XX% sparsity using magnitude-based pruning [31]. For SpMV, matrices are processed in CSR format with a dense input vector using the SpMV macro described in Section VI-A3. Graph-algorithm benchmarks use CSR adjacency matrices with dense frontier or rank vectors. Neural Network comparisons start from identical PyTorch checkpoints (Listing 1) for both HLS4ML [32] and the proposed approach. HLS4ML translates checkpoints into Intel HLS, while the

```python
class MNIST_MLP(nn.Module):
   def __init__(self):
      super().__init__()
      self.fc1 = nn.Linear(784, 256, bias=True)
      self.fc2 = nn.Linear(256, 10, bias=True)

   def forward(self, x):
      return self.fc2(F.relu(self.fc1(x)))

model = MNIST_MLP()
# train the model here
prune([(model.fc1, 'w'), (model.fc2, 'w')], amount=0.8)
torch.save(model.state_dict(), "partial_mnist_mlp.pt")
```

Listing 1: Two-layer NN (Neural Network) expressed in Pytorch, with its weights pruned and saved as a checkpoint.

```
val w1 = Partition("fc1_w.csv", sch=HETERO, nblk=64)
val w2 = Partition("fc2_w.csv", sch=HETERO, nblk=32)
val b1, b2 = Input("fc1_b.csv"), Input("fc2_b.csv")

// Layer 1: Linear (SpMV_hetero from VI.A3) + bias + ReLU
val h = Map(ReLU.asFunction(),
         Map(Add.asFunction(),
            Zip(Tuple(SpMV_hetero(w1, x), b1))))

// Layer 2: Linear (SpMV) + bias
val y = Map(Add.asFunction(),
         Zip(Tuple(SpMV_hetero(w2, h), b2)))

compile(y).generateHDL()
```

Listing 2: Two-layer sparse NN expressed in the extended SHIR.

proposed approach extracts and encodes tensors using dependent types and represents the network using a concise SHIR program (Listing 2). All implementations use 32-bit data and indices by default, unless optimized by the specific approach.

Metrics. Performance is evaluated with two metrics: throughput, in GOPS (multiplications and additions counted separately), and bandwidth efficiency, in MOPS/GBPS. Wall-clock execution time t_{exec} is recorded, and throughput is computed as $2 \times \mathrm{NNZ}/(t_{exec} \times 10^6)$. Bandwidth efficiency is obtained by dividing this throughput by the sustained off-chip bandwidth of each platform: 6.4 GBPS for Arria 10, 265–285 GBPS for the Alveo U280 designs reported by HiSparse [19], HiSpMV [21], and GraphLily [20] (depending on the number of enabled HBM channels), 288 GBPS for the CPU baseline, and 498 GBPS for the RTX 1080 Ti. For graph algorithms, throughput is reported in millions of traversed edges per second (MTEPS), calculated as $\mathrm{Edges}_{\mathrm{processed}}/(t_{exec} \times 10^6)$, and bandwidth efficiency in MTEPS/GBPS. For the designs generated by the proposed approach, Arria 10's metrics are computed from measured runtimes, whereas simulated variants derive these metrics from cycle-accurate traces.

B. Impact of Architectures and Optimizations on SpMV

The impact of parallelization strategies and optimizations such as vector sharding, are evaluated on the Stratix-10 in simulation with the three partitioning architectures (row-level, homogeneous, and heterogeneous). Automatic FIFO insertion via re-write was enabled across all designs (and will be for all future designs as well), as its absence drops throughput by about 40% on imbalanced datasets and 20% on balanced ones. Figure 10 shows the throughput speedup of homogeneous and heterogeneous partitioning schemes, with and

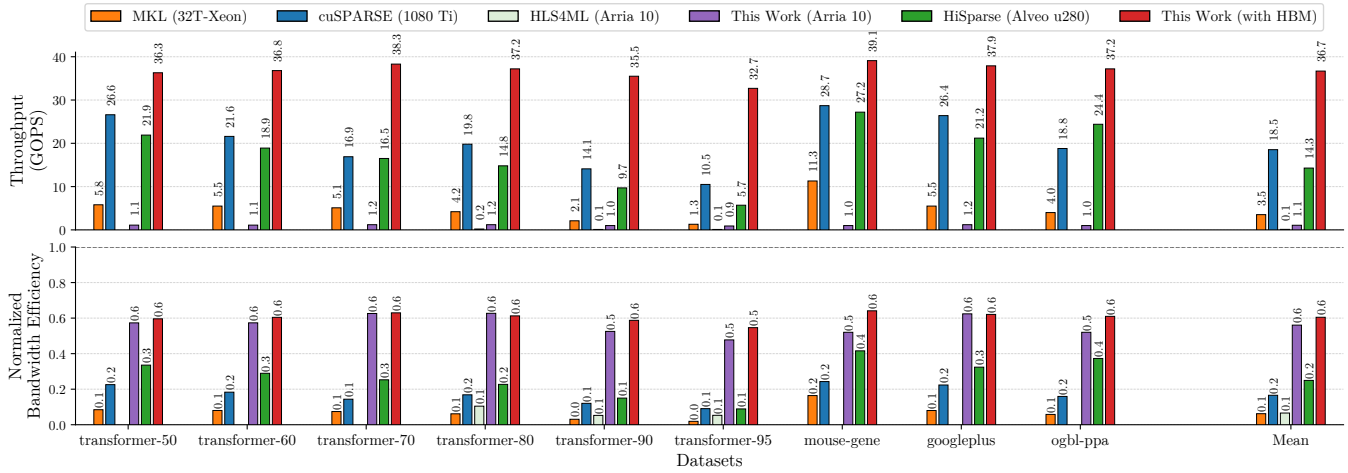

Fig. 11: Throughput (GOPS) and Normalized Bandwidth Efficiency of SpMV on **balanced datasets** ($nnz/row > 20$). *Arria 10* denotes synthesized designs; *with HBM* denotes simulations using 16 Ch. HBM @ 250 GBPS; empty bars signify infeasible designs. Higher is better.

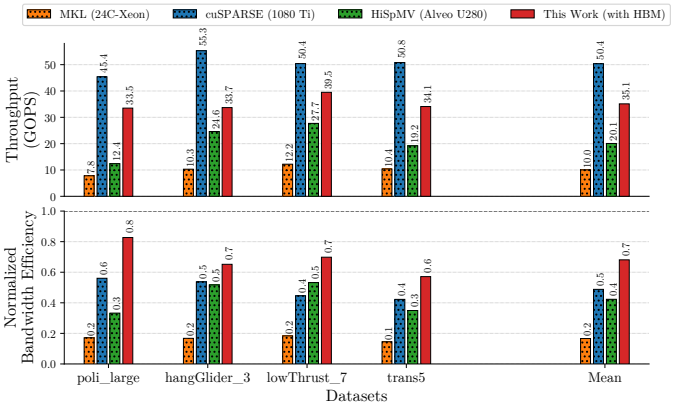

Fig. 12: Throughput (GOPS) and Normalized Bandwidth Efficiency of SpMV on **imbalanced datasets** ($nnz/row \ll 20$). *with HBM* denotes simulation using 16 Ch. HBM @ 250 GBPS. Higher is better.

without vector sharding, relative to row-level partitioning. The row-level baselines (replicating MAC units per row) suffers from idle resources when rows have fewer non-zeros than the parallel factor, yielding poor performance. Homogeneous partitioning standardizes sub-matrix dimensions, simplifying scheduling but risks load imbalance if some partitions hold little data. This is reflected by its relatively poor performance on imbalanced datasets. In contrast, heterogeneous partitioning adapts to nonzero distributions and, with vector sharding, achieves on average 70× higher throughput on average than row-level designs.

C. Sparse Matrix Vector Product (SpMV)

The proposed sparse matrix specializations are evaluated against several classes of accelerators. Reconfigurable CGRA-based approaches, such as Stellar [33] and Stardust [34], prioritize programmability over the specialization required for high performance. An analysis of their respective evaluations shows that even on smaller matrices, such designs are notably slower than kernel-specific accelerators [19]–[21], which are benchmarked using significantly larger and more memory-intensive matrices. Consequently, the state-of-the-art HiSparse [19] and HiSpMV-16 [21] are selected as the primary baselines representing the kernel-specific accelerator class. For a comprehensive comparison, standard libraries (CPU-based MKL and GPU-based cuSPARSE) and a representative template-based HLS flow using HLS4ML are also included.

The proposed heterogeneous SpMV design is evaluated in two configurations: one tested on an Arria 10 FPGA and another, *with HBM*, simulated with a 16-channel 250 GBPS HBM configuration for a fair comparison against Alveo U280-based baselines. Both configurations employ 256 heterogeneous partitions. Baseline results for balanced datasets on CPU, GPU, and HiSparse are taken from the HiSparse study [19], while those for imbalanced datasets are sourced from the HiSpMV evaluation [21].

To ensure a fair, device-independent comparison and establish an upper bound on memory efficiency, this analysis uses *normalized bandwidth efficiency*. This metric is the ratio of effective to peak memory bandwidth; equivalently, it is the ratio of measured bandwidth efficiency (BWE) to the theoretical maximum derived from operational intensity (OI).

$$\frac{\text{BWE}}{\text{OI}} = \frac{(\text{N}_{\text{ops}}/\text{t}_{\text{exec}})/\text{BW}_{\text{peak}}}{\text{N}_{\text{ops}}/\text{Data}_{\text{traffic}}} = \frac{\text{Data}_{\text{traffic}}/\text{t}_{\text{exec}}}{\text{BW}_{\text{peak}}} = \frac{\text{BW}_{\text{effec.}}}{\text{BW}_{\text{peak}}}$$

Balanced Datasets: Figure 11 compares the throughput (GOPS) and normalized bandwidth efficiency on balanced datasets. The synthesized Arria 10 design achieves, on average, 9×, 3.4×, 8.5×, and 2.2× higher bandwidth efficiency over MKL, cuSPARSE, HLS4ML and HiSparse, respectively. Moreover, these findings demonstrate that the proposed matrix specialization techniques reduce the bandwidth efficiency shortfall to a mere 40%, whereas HiSparse (the next best performer) still suffers an 80% gap. These efficiency gains stem from the automatic matrix specialization techniques that improve load-balancing and lower indexing bitwidth.

The simulated variant on the Stratix-10 (*with HBM*) boosts throughput, compared to the Arria-10, achieving 10.5×, 2×, and 2.6× gains over MKL, cuSPARSE, and HiSparse. Interestingly, it maintains a similar bandwidth efficiency to the

TABLE II: Comparison of Resource Usage (Logic, RAM), and Throughput (GOPS) across models and target parallelizations using HLS4ML and the Proposed approach tested on an Arria 10. For resource reduction and speedup values, higher is better.

Model	Parallelization	Logic (%)			RAM (%)			Throughput (GOPS)		
		HLS4ML	This Work	Reduction	HLS4ML	This Work	Reduction	HLS4ML	This Work	Speedup
JetTagger_5layer	1024	95	54	1.8×	98	82	1.2×	96.4	172.4	1.8×
	512	89	37	2.4×	96	81	1.2×	68.3	96.1	1.4×
	256	88	24	3.7×	95	79	1.2×	50.3	49.4	1.0×
JetTagger_3layer	128	64	13	4.9×	45	24	1.9×	25.3	24.3	1.0×
	64	57	11	5.2×	43	23	1.9×	13.2	12.5	0.9×
	32	47	10	4.7×	41	19	2.2×	6.5	6.2	1.0×
MNIST Classifier	1024	256	64	4.0×	180	85	2.1×	—	171.2	—
	512	200	49	4.1×	165	83	2.0×	—	96.3	—
	256	130	43	3.0×	151	83	1.8×	—	48.9	—

Designs in red do not synthesize; for all designs, DSP usage equals the parallelization factor.

TABLE III: Throughput (MTEPS) and Bandwidth Efficiency (MTEPS/GBPS) of BFS and PageRank compared to GraphLily. Values in parentheses show speedup (higher is better).

	Dataset	Throughput (MTEPS)		Bandwidth Efficiency (MTEPS/GBPS)	
		GraphLily	with-HBM‡	GraphLily	with-HBM‡
BFS	mouse-gene	4,999	9,147 (1.8×)	17.5	36.6 (2.1×)
	googleplus	5,111	8,031 (1.6×)	17.9	32.1 (1.8×)
	hollywood	6,863	9,129 (1.3×)	24.1	36.5 (1.5×)
	Geomean	**5,597**	**8,753** (1.6×)	**19.6**	**35.0** (1.8×)
PageRank	mouse-gene	6,265	12,737 (2.0×)	22.0	50.9 (2.3×)
	googleplus	7,092	12,382 (1.7×)	24.9	49.5 (2.0×)
	hollywood	7,471	12,422 (1.7×)	26.2	49.7 (1.9×)
	Geomean	**6,924**	**12,512** (1.8×)	**24.3**	**50.0** (2.1×)

with-HBM‡ refers to designs simulated with 16-Channel HBM @ 250 GBPS.

Arria 10 design, confirming that the approach presented is able to exploit the extra bandwidth available with HBM perfectly.

Imbalanced Datasets: Four imbalanced datasets are used for comparison with HiSpMV-16 [21], a state-of-the-art accelerator that outperforms HiSparse and other accelerators on such workloads. As shown in Figure 12, on average, *with HBM* delivers 1.7× higher throughput and achieves a 1.6× improvement in bandwidth efficiency compared to HiSpMV-16. Moreover, the results show that the proposed matrix specialization techniques bring the bandwidth efficiency gap down to just 30% compared to 50% of the next best performing accelerator (GPU). This demonstrates that the proposed approach specializes the generated accelerators to the underlying data shape and thus adapts well to imbalanced sparse workloads.

Closing the remaining 20-30% of bandwidth efficiency gap depends on two factors: pipeline startup bubbles in vector buffering and workload imbalance from static, heuristic partitioning. Vector buffering must fill local buffers before steady state, and the greedy load-balancing rule cannot perfectly predict nonzero distribution. Introducing adaptive buffering that overlaps data alignment with computation and enriching the partitioner with dynamic profiling could minimize these startup and imbalance overheads but is left for future work.

D. Graph Algorithms - BFS and PageRank

Beyond SpMV, two graph algorithms are examined: BFS (Breadth-First Search) and PageRank. Both heavily stress memory bandwidth due to their irregular data traversal patterns. Results are compared against GraphLily [20] using

TABLE IV: Overview of Neural Network Architectures.

Model Name	Sequential Layer Sizes and Activations	Sparsity
`MNIST Classifier`	$784 \times 512_R \times 256_R \times 128_R \times 10_{Sf}$	80%
`JetTagger_5layer`	$16 \times 64_R \times 32_R \times 32_R \times 5_{Sf}$	60%
`JetTagger_3layer`	$10 \times 32_R \times 1_{Sg}$	95%

$A \times B_R \times C_R$ denotes a neural network where A inputs are mapped to B outputs via activation R, which then feed a layer mapping B inputs to C outputs using activation R.
R = ReLU, Sg = Sigmoid, Sf = Softmax.

three randomly selected benchmarks. Comparisons to CPU, GPU, and CHLS are omitted for brevity, as GraphLily already outperforms these baselines. Because GraphLily runs on an Alveo u280 with 30× the memory bandwidth of an Arria 10, only the simulated (*with HBM*) design with 256 heterogeneous partitions is reported to ensure a fair comparison.

Table III shows throughput and bandwidth efficiency. For BFS, on average, the proposed design (*with-HBM‡*) achieves a 1.6× speedup and a 1.8× gain in bandwidth efficiency over GraphLily; for PageRank, it nearly doubles throughput (1.8×) and improves bandwidth efficiency by 2.1×. Overall, these results confirm that the proposed approach adapts well to other irregular workloads and maintains high performance even under bandwidth-intensive conditions.

E. Accelerating Neural Networks

Sparse neural networks retain only a small fraction of their weights after pruning, fixing the nonzero pattern for the lifetime of the model. This invariance invites accelerators that hardwire addresses, pipelines, and reduction trees to the exact structure. Three such workloads are evaluated, listed in Table IV: two JetTagger classifiers from HLS4ML [32] and a sparsified MNIST model [35]. For a fair comparison, both HLS4ML and the proposed flow aim to place the entire network on a single Arria 10, making resource the dominant constraint.

Although HLS4ML also has compile-time knowledge of tensor shapes, its reliance on C-based HLS compilers proves to be a critical limitation. These compilers, tuned for dense arrays, generate resource-inefficient designs for sparse data, causing larger networks to exceed the target FPGA's limits and fail place-and-route (Table II, highlighted in red).

The proposed flow encodes the sparse structure as dependent types and generates pipelines specialized for the underlying sparsity pattern. As shown in Table II, these resulting designs

```
def scaleRow(pair: (DataT, [DataT]_N)) =
  Map(λ (a,b) ⇒ a * b,
     Zip(Replicate(pair._1, N), pair._2))

def rowCompute(row: [DataT]_N) = {
  val scaled = Map(λ pair ⇒ scaleRow(pair),
                Zip(row.vals, Gather2D(B, row.indices)))
  Fold(Add.asFunction(), ZeroVec(N), scaled)
}

// Apply to every sparse row of A to build result C
Map_D(λ row ⇒
   ArithTypeLambda(λ_A i ⇒
      rowCompute(row)), A)
```

Listing 3: Row-wise SpMM expressed in the extended SHIR.

sustain up to 171.2 GOPS on the MNIST classifier, deliver up to 1.8× higher throughput than HLS4ML on JetTagger-5layer, and do so with lower resource footprints. These experiments confirm that a type-directed approach can exploit fixed sparsity patterns where conventional HLS flows fall short, even when both start with identical compile-time information.

VIII. Discussion

This work introduces a functional, type-driven methodology for generating specialized hardware tailored to fixed-sparsity workloads. The compositional primitives offered by the IR naturally generalize to other sparse dataflows. For instance, in the row-wise SpMM example detailed in Lst. 3, each row of the sparse matrix A is streamed individually. The kernel gathers corresponding dense rows of B based on the non-zero column indices, scales these rows by their respective values, and accumulates the intermediate results using a `Fold` operation, updating the current output row $C_{(i,:)}$. This exemplifies the flexibility of constructing fundamentally distinct hardware pipelines through the composition of core primitives.

Furthermore, future efforts can extend this approach along two key directions to broaden its applicability. Currently, the provided primitives focus exclusively on sparse-input, dense-output kernels. Incorporating support for sparse-sparse operations, which produce dynamically structured outputs, could be achieved through dependent pairs [29]. Such an enhancement would enable the type system to precisely track and automatically generate hardware for evolving sparse structures, essential in operations such as SpMSpV or graph additions.

To support fully dynamic sparsity at runtime, the proposed approach could be extended to configure a programmable sparse hardware overlay. Achieving this would entail implementing dependent pair primitives [29] directly in hardware, allowing runtime metadata (such as row lengths) to stream into on-chip memory alongside data. This metadata would dynamically configure the processing elements and interconnections for each sparse tensor encountered at runtime without recompilation. Such a design would trade the peak efficiency characteristic of fully specialized hardware with the generality required for dynamic workloads, effectively bridging the gap between application-specific and reconfigurable accelerators.

IX. Related Work

Extensive research has addressed automatic generation of dense accelerators. Frameworks and languages such as Spatial [23], AnyHLS [36], and Lift-HLS [22] offer abstractions that generate hardware descriptions (e.g., Chisel RTL [37], C code) and enable optimizations like pipeline scheduling and memory banking. However, these methods primarily assume dense, static data structures, offering limited support for the irregularity inherent to sparse computations.

Prior work in generic sparse accelerator generation typically targets CGRAs (Coarse-Grained Reconfigurable Architectures) [8]–[21]. Many of these systems feature sophisticated high-level abstractions, such as the SAM compiler for the Onyx CGRA [38], the C/C++ compiler for the SPU [39], and DSLs like Stellar [33] and Spatial-based Capstan/Stardust [34], [40], to map computations to reconfigurable hardware templates. However, the flexibility to support a *domain* of sparse applications inherently limits the degree of specialization possible for any single computational flow, creating an efficiency trade-off. In contrast, this work synthesizes fully custom, ultra-specialized architectures directly from a high-level description of an individual sparse application.

High-level APIs such as GraphBLAS [5] provide a powerful abstraction for expressing graph algorithms using sparse linear algebra. While GraphBLAS provides a standard API, it does not define a compilation pathway for synthesizing hardware accelerators, creating a gap between algorithm specification and hardware implementation. The presented approach introduces a functional IR to potentially bridge this gap, serving as a compilation target for such APIs rather than a new user-facing abstraction. This could provide a key advantage over methods like GraphLily [20], enabling holistic, end-to-end optimization by synthesizing hardware for entire compositions of operations rather than isolated kernels.

Recent works have explored dependent typing to represent sparse data structures more expressively [28], [29]. These efforts, primarily intended for CPUs and GPUs, introduce dependent types for sparse computations but offer no support for synthesizing specialized hardware accelerators.

X. Conclusion

This paper presents a functional HLS approach that enables the concise specification of a wide variety of sparse algorithms. By integrating dependent types and specialized hardware primitives, the approach efficiently handles irregular data and automates specialized accelerator generation. Experimental results indicate that, for SpMV benchmarks, the proposed design achieves a throughput improvement of 2.6× over HiSparse on balanced datasets and 1.7 × over HiSpMV-16 on imbalanced datasets. Concurrently, it reduces the bandwidth efficiency gap to 40% and 30%, significantly outperforming HiSparse and HiSpMV, which have gaps of 80% and 70%, respectively. For graph algorithms, throughput and bandwidth efficiency improvements of up to 1.8× and 2.1× are observed. In neural network inference, the approach delivers up to 171.2 GOPS of throughput while significantly reducing resource usage over HLS4ML. Overall, these findings confirm that the proposed approach offers a robust and scalable solution for rapidly designing high-performance specialized sparse accelerators.

ACKNOWLEDGMENTS

We thank Christof Schlaak for implementing the core SHIR project, which this work extended. This research was supported in part by a scholarship from the Fonds de Recherche du Quebec, Nature et Technologies (FRQNT) under Award #336324; the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [Grant RGPIN-2020-05889]; and the Canada CIFAR AI Chairs Program.

REFERENCES

[1] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40. DOI: 10.1145/3079856.3080254.

[2] J. Kepner and J. R. Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*, ser. Software, environments, tools. SIAM, 2011, vol. 22. URL: http://dblp.uni-trier.de/db/books/collections/KG2011.html

[3] A. Canning, G. Galli, F. Mauri, A. De Vita, and R. Car, "O(n) tight-binding molecular dynamics on massively parallel computers: an orbital decomposition approach," *Computer Physics Communications*, vol. 94, no. 2, pp. 89–102, 1996. DOI: https://doi.org/10.1016/0010-4655(96)00009-4. URL: https://www.sciencedirect.com/science/article/pii/0010465596000094

[4] N. Nayak, T. O. Odemuyiwa, S. Ugare, C. Fletcher, M. Pellauer, and J. Emer, "Teaal: A declarative framework for modeling sparse tensor accelerators," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1255–1270. DOI: 10.1145/3613424.3623791. URL: https://doi.org/10.1145/3613424.3623791

[5] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9. DOI: 10.1109/HPEC.2016.Kepner16GraphBLAS.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[7] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 363–372. DOI: 10.1109/ICDM.2008.89.

[8] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "pal18outerspace: An outer product based sparse matrix multiplication accelerator," pp. 724–736, 2018. DOI: 10.1109/HPCA.2018.00067.

[9] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274, 2020.

[10] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 75–84. DOI: 10.1145/3020078.3021745. URL: https://doi.org/10.1145/3020078.3021745

[11] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, "Alrescha: A lightweight reconfigurable sparse-computation accelerator," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 249–260. DOI: 10.1109/HPCA47549.2020.00029.

[12] C. Y. Lin, Z. Zhang, N. Wong, and H. K.-H. So, "Design space exploration for sparse matrix-matrix multiplication on fpgas," in *2010 International Conference on Field-Programmable Technology*, 2010, pp. 369–372. DOI: 10.1109/FPT.2010.5681425.

[13] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning sparse matrix row permutations for efficient spmm on gpu architectures," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 48–58. DOI: 10.1109/ISPASS51385.2021.00016.

[14] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format," *SIGPLAN Not.*, vol. 51, no. 8, Feb. 2016. DOI: 10.1145/3016078.2851190. URL: https://doi.org/10.1145/3016078.2851190

[15] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Sparse systolic tensor array for efficient cnn hardware acceleration," 2020.

[16] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Transactions on Circuits and Systems for Video Technology*, p. 1–1, 2020. DOI: 10.1109/tcsvt.2019.2911674. URL: http://dx.doi.org/10.1109/TCSVT.2019.2911674

[17] E.-J. Im and K. Yelick, "Optimizing sparse matrix vector multiplication on smps," 06 2001.

[18] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 36–43. DOI: 10.1109/FCCM.2014.23.

[19] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 54–64. DOI: 10.1145/3490422.3502368. URL: https://doi.org/10.1145/3490422.3502368

[20] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643582.

[21] M. B. Rajashekar, X. Tian, and Z. Fang, "Hispmv: Hybrid row distribution and vector buffering for imbalanced spmv acceleration on fpgas," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 154–164. DOI: 10.1145/3626202.3637557. URL: https://doi.org/10.1145/3626202.3637557

[22] M. Kristien, B. Bodin, M. Steuwer, and C. Dubach, "High-level synthesis of functional patterns with lift," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ser. ARRAY 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 35–45. DOI: 10.1145/3315454.3329957. URL: https://doi.org/10.1145/3315454.3329957

[23] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," *SIGPLAN Not.*, vol. 53, no. 4, p. 296–311, Jun. 2018. DOI: 10.1145/3296979.3192379. URL: https://doi.org/10.1145/3296979.3192379

[24] D. Durst, M. Feldman, D. Huff, D. Akeley, R. Daly, G. L. Bernstein, M. Patrignani, K. Fatahalian, and P. Hanrahan, "Type-directed scheduling of streaming accelerators," p. 408–422, 2020. DOI: 10.1145/3385412.3385983. URL: https://doi.org/10.1145/3385412.3385983

[25] C. Schlaak, T.-H. Juang, and C. Dubach, "Memory-aware functional ir for higher-level synthesis of accelerators," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 2, jan 2022. DOI: 10.1145/3501768. URL: https://doi.org/10.1145/3501768

[26] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish, S. Orgrenci-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," 2021.

[27] V. Isaac–Chassande, A. Evans, Y. Durand, and F. Rousseau, "Dedicated hardware accelerators for processing of sparse matrices and vectors: A survey," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, Feb. 2024. DOI: 10.1145/3640542. URL: https://doi.org/10.1145/3640542

[28] F. Pizzuti, M. Steuwer, and C. Dubach, "Generating fast sparse matrix vector multiplication from a high level generic functional ir," in *Proceed-*

ings of the 29th International Conference on Compiler Construction, ser. CC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 85–95. DOI: 10.1145/3377555.3377896. URL: https://doi.org/10.1145/3377555.3377896

[29] ——, "Generating high performance code for irregular data structures using dependent types," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, ser. FHPNC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 37–49. DOI: 10.1145/3471873.3472977. URL: https://doi.org/10.1145/3471873.3472977

[30] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. DOI: 10.1145/2049662.2049663. URL: https://doi.org/10.1145/2049662.2049663

[31] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015.

[32] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, jul 2018. DOI: 10.1088/1748-0221/13/07/P07027. URL: https://dx.doi.org/10.1088/1748-0221/13/07/P07027

[33] H. N. Genc, H. Kim, P. Ganesh, and Y. S. Shao, "Stellar: An automated design framework for dense and sparse spatial accelerators," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 409–422. DOI: 10.1109/MICRO61859.2024.00038.

[34] O. Hsu, A. Rucker, T. Zhao, V. Desai, K. Olukotun, and F. Kjolstad, "Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 628–643. DOI: 10.1145/3696443.3708918. URL: https://doi.org/10.1145/3696443.3708918

[35] Y. LeCun and C. Cortes, "The mnist database of handwritten digits," 2005. URL: https://api.semanticscholar.org/CorpusID:60282629

[36] M. A. Özkan, A. Pérard-Gayot, R. Membarth, P. Slusallek, R. Leißa, S. Hack, J. Teich, and F. Hannig, "Anyhls: High-level synthesis with partial evaluation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3202–3214, 2020. DOI: 10.1109/TCAD.2020.3012172.

[37] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," pp. 1212–1221, 2012. DOI: 10.1145/2228360.2228584.

[38] K. Koul, M. Strange, J. Melchert, A. Carsello, Y. Mei, O. Hsu, T. Kong, P.-H. Chen, H. Ke, K. Zhang, Q. Liu, G. Nyengele, A. Balasingam, J. Adivarahan, R. Sharma, Z. Xie, C. Torng, J. Emer, F. Kjolstad, M. Horowitz, and P. Raina, "Onyx: A programmable accelerator for sparse tensor algebra," in *2024 IEEE Hot Chips 36 Symposium (HCS)*, 2024, pp. 1–91. DOI: 10.1109/HCS61935.2024.10665150.

[39] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019. DOI: 10.1145/3352460.3358276. URL: https://doi.org/10.1145/3352460.3358276

[40] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector rda for sparsity," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1022–1035. DOI: 10.1145/3466752.3480047. URL: https://doi.org/10.1145/3466752.3480047