Stellar: An Automated Design Framework for Dense and Sparse Spatial Accelerators

Hasan Nazim Genc
University of California, Berkeley
hngenc@berkeley.edu

Hansung Kim
University of California, Berkeley
hansung_kim@berkeley.edu

Prashanth Ganesh
University of California, Berkeley
prashanthcganesh108@berkeley.edu

Yakun Sophia Shao
University of California, Berkeley
ysshao@berkeley.edu

***Abstract*—Domain-specific hardware, coupled with co-designed algorithmic optimizations, plays a pivotal role in accelerating both dense and sparse workloads, surpassing the capability of general-purpose platforms. However, the diverse nature of these specialized hardware platforms makes it challenging to systematically implement, evaluate, and compare different solutions.**

To address these shortcomings, we introduce Stellar, a novel accelerator design framework tailored for dense and sparse spatial accelerators. Stellar introduces abstractions that systematically decouple different dimensions of accelerator design, addressing the need for a clear separation of concerns for automated design solutions. This modular approach enhances the clarity and flexibility of the design process, while enabling automated hardware generation for a range of dense and sparse accelerator designs. Stellar outputs synthesizable Verilog implementations of these accelerators, paired with RISC-V programming interfaces. We demonstrate that Stellar-generated accelerators are comparable to hand-written, high-quality hardware designs, enabling effective evaluation, comparison, and design-space exploration for both dense and sparse accelerators.

I. Introduction

In response to diminishing technology scaling trends and the growing computational demands of modern workloads, computer architects have increasingly turned to domain-specific accelerators and co-designed software optimizations for improved energy and area efficiency. However, the expanding landscape of diverse workloads has given rise to a multitude of hardware designs and software co-optimization techniques. These diverse solutions range from fixed-function matrix-multiplication arrays for dense DNNs [10], [12], [17], [23], [25], to co-designed structured sparsity formats that remove low-priority weights or features from large DNN models [1], [22], [32], [40], to accelerators for extremely sparse, highly-imbalanced tensor operations [7], [14], [26], [30], [38], [39].

While this broad spectrum of hardware and software optimizations presents ample opportunities for accelerator and software co-design, it also greatly complicates the analysis, exploration, and design of specialized architectures. Prior work has attempted to address these challenges through methods ranging from ad-hoc hardware design to proposals for well-defined, expressive accelerator taxonomies. Ad-hoc design, a traditional and flexible approach, provides limited opportunities for automated and rapid design space exploration.

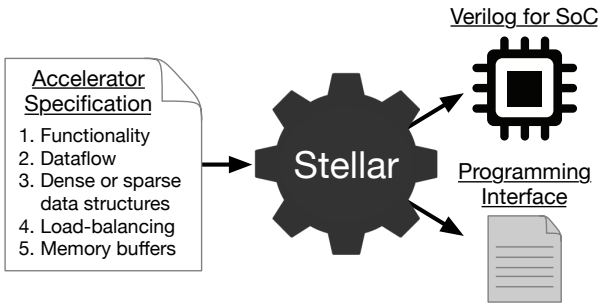


Fig. 1: A simplified illustration of Stellar's design flow, from the user-specified inputs on the left to the Verilog and programming interface outputs on the right.

Conversely, techniques like high-level synthesis enable swift hardware development via direct compilation from software to hardware, but fail to maintain a strong separation of concerns in the hardware design process, making it difficult to explore independent design choices without modifying unrelated parts of an architect's specifications. Recent efforts have introduced expressive abstractions and taxonomies in an attempt to disentangle various components of accelerator design. However, these approaches often fail to generate synthesizable sparse hardware designs, focusing predominantly on higher-level modeling, or they struggle to describe low-level aspects of hardware design that are of interest to architects.

In response to the complexity posed by the diverse landscape of accelerator design, we develop "Stellar", a new accelerator design framework for automated dense and sparse spatial accelerators[1]. Stellar addresses these challenges by introducing three key features: (i) it provides expressive abstractions for the design of *both* dense and sparse accelerators, (ii) it maintains a strong separation of concerns between different design considerations, allowing independent specification and exploration, and (iii) it generates synthesizable Verilog implementations of user-specified hardware, together with RISC-V programming interfaces that can easily be incorporated into users' software applications. Stellar's design flow, summarized in Figure 1, enables the rapid development and generation of

[1]Stellar is open-sourced at https://github.com/hngenc/stellar.

accelerators for both dense and sparse workloads.

Building upon prior taxonomies for dense and sparse accelerators [6], [21], [24], [27], [35], Stellar introduces new abstractions of interest to hardware developers, such as fine-grained load-balancing schemes and pipelining strategies. Users independently express a specialized hardware accelerator's (i) functional behavior, (ii) dataflow, (iii) supported sparsity patterns, (iv) load-balancing strategy, and (v) private memory buffers. They can modify these different design considerations in isolation and observe the subtle interactions between them to determine the best accelerator design choice.

Stellar maps these design specifications to hardware templates, and optimizes them to maximize data reuse and minimize area and wiring congestion. Stellar then outputs synthesizable Verilog implementations of a full SoC, including user-specified accelerators, optional RISC-V host CPUs, and shared memory hierarchies. Our evaluation demonstrates that Stellar-generated hardware implementations perform competitively to hand-designed accelerators, and that they effectively expose various performance bottlenecks caused by either hardware or software design choices, which often cannot be exposed in high-level simulators or models. We provide an end-to-end, unified platform for both dense and sparse accelerator design, enabling systematic evaluation and comparison of diverse architectural design choices.

II. Background and Motivation

The growing diversity of spatial accelerators, tailored for dense and sparse workloads, has spurred the development of design space exploration frameworks to expedite the design process. However, existing accelerator design frameworks often lack the ability to automatically *synthesize* hardware implementations, especially for *sparse* accelerators which require subtle tradeoffs across multiple design dimensions. This section reviews prior work in dense and sparse accelerator designs and motivates the need for an automatic and unified flow for synthesizing both dense and sparse accelerators.

A. Novel Spatial Accelerators

Spatial accelerators, which spatially distribute an array of processing elements (PEs), present a broad spectrum of tradeoffs in performance, programmability, and resource utilization for dense and sparse workloads. The design space for dense accelerators is well understood as they typically differ in the dataflows they support, the quantized or unquantized datatypes they operate on, the functional operations they can perform (e.g., ReLU, GeLU, or other activation functions), or in their resource constraints, e.g. for small accelerators targeting edge devices or for high-performance accelerators in the cloud.

Sparse accelerators differ in even more ways, due to the wide variety of sparsity distributions and the corresponding sparse data formats in sparse workloads. Some sparse accelerators are designed for extremely sparse workloads, where far fewer than 1% of elements are non-zeros [26], [30], [38], [39]. Others target sparse DNNs [1], [28], [29] with matrix densities ranging from 30% to 70%. The vast range of sparsity levels necessitates the adoption of distinct sparse data formats and hardware designs, exposing a substantial design space.

As a result, existing accelerator designs generally differ from each other in *multiple* ways. For example, recent sparse accelerator proposals commonly propose not only new hardware dataflows but also new sparse formats and load balancing strategies [30]. This inherent diversity complicates the process of comparing different accelerators, making it challenging to discern *which* feature contributes to specific improvements or drawbacks. However, such nuanced comparisons are crucial for architects, providing insights into the key principles underlying each design and guiding the selection of optimal solutions for specific workloads.

B. Dense Spatial Accelerator Design Frameworks

To address the complexities of dense spatial accelerators, prior efforts have developed hardware generation frameworks that systematically enumerate the dense design space. These frameworks feature orthogonal design dimensions, e.g., *functionality* to describe an accelerator's expected outputs, *dataflow* to describe data reuse patterns, and *memory buffers* to describe the memory hierarchy. The regularity of dense accelerators within these well-defined design dimensions has enabled the development of frameworks like PolySA [6], AutoSA [33], and Interstellar [36]. These frameworks can automatically synthesize RTL implementations for FPGAs or ASICs, together with application-level APIs as their programming interfaces, while offering a comprehensive separation of design concerns for systematic design space exploration.

However, although these hardware design frameworks can effectively separate the concerns that go into dense accelerator design while generating high-quality RTL, they are challenging to extend to sparse accelerator design. Sparse accelerators introduce new key design considerations such as different sparse data structures and load-balancing techniques, which are often overlooked by frameworks designed exclusively for dense scenarios. In addition, while the application-level programming interface facilitates the offloading of the entire workload to accelerators, the absence of a low-level ISA makes it hard to deploy for sparse workloads due to their irregular nature of computation. Consequently, this inherent limitation significantly narrows their applicability, especially for the growing number of modern workloads that involve irregular sparse computations.

C. Sparse Spatial Accelerator Design Frameworks

While existing design frameworks for dense accelerators provide end-to-end flows from abstract design specifications to RTL generation, frameworks dedicated to *sparse* accelerators have primarily focused on *modeling* and *simulating* sparse hardware designs and are not able to automate the RTL generation process. As such, they still leave significant manual work for hardware designers, and may sometimes fail to expose low-level performance bottlenecks not accounted for in higher-level simulators.

		Prior Dense Frameworks				*Prior Sparse Frameworks*					***Stellar***
		PolySA	**AutoSA**	**Interstellar**	**Tabla**	**Sparseloop**	**TeAAL**	**SAM**	**DSAGen**	**Spatial**	
Design Specifications	*Functionality*	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Dataflow	✓	✓	✓	×	✓	✓	✓	Implicit	Implicit	✓
	Sparse data structures	×	×	×	×	✓	✓	✓	×	×	✓
	Load-balancing	×	×	×	×	×	✓	×	✓	×	✓
	Private memory buffers	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Hardware Outputs	*Simulators*	×	×	×	×	✓	✓	✓	×	×	×
	Synthesizable RTL	✓	✓	✓	✓	×	×	×	✓	✓	✓
Programming Interface	*Application-level*	✓	✓	✓	✓	×	×	×	✓	✓	✓
	ISA-level	×	×	×	×	×	×	×	×	×	✓

TABLE I: A comparison between Stellar and other accelerator design frameworks which attempt to separate different concerns.

Specifically, recent works on modeling and simulating sparse hardware accelerators, such as TeAAL [24], the Sparse Abstract Machine (SAM) [16], and Sparseloop [35], introduce new abstractions for *sparse data formats* and *load balancing*, separated from other design concerns. These works define the functionality of sparse accelerators using syntax such as Einstein summations [11], the scheduling of operations on them using Halide-like loop transformations, and their sparse data formats using extensible abstractions such as fibertrees [31]. While TeAAL supports certain load-balancing schemes by allowing various tensor dimensions to be flattened for balanced distribution to spatial array PEs, it is difficult to describe with this method more sophisticated load-balancing strategies [13], where individual PEs in a spatial array might have greater load-balancing capabilities than others.

Although these frameworks enable rapid specification, evaluation, and simulation of sparse spatial accelerators, none of them *generate* actual RTL implementations. TeAAL and Sparseloop provide simulation and modeling capabilities and are primarily intended for early-stage exploration. SAM, while defining a set of hardware components for mapping dataflow, limits its evaluation to cycle-approximate simulations and CGRAs, lacking actual RTL implementations.

Finally, prior work has proposed hardware generation frameworks which *do* generate sparse accelerator RTL, but these lack the full separation of concerns necessary for effective design-space exploration. For example, DSAGEN [34] generates RTL for both dense and sparse workloads, however, it expects them to be defined as annotated C programs. These C descriptions are limited in their ability to separate out all the different concerns which go into sparse hardware design, so that each can be explored independently, as for the previously mentioned taxonomies. Other works, such as Spatial [19], introduce languages which describe custom hardware as sets of nested-loops, which generalize well to a variety of workloads, but which make it difficult to separate the functionality of an accelerator fully from its scheduling and dataflow, and which also lack higher-level constructions, such as data format specifications, necessary to concisely express sparse workloads.

Unlike prior work, Stellar is an end-to-end tool that not only enables the specification of *both* dense and sparse accelerators with high-level descriptions that separate independent design concerns, but also *generates* efficient and synthesizable Verilog implementations of these high-level specifications. Specifically, Stellar enumerates the sparse accelerator design space along five separate, independent axes: (i) functionality, (ii) dataflow, (iii) sparse data structures, (iv) load-balancing strategy, and (v) private memory buffers. This modular approach facilitates the development of each design consideration in isolation, providing architects with a high degree of separation of concerns during the design process. Subsequently, Stellar maps these specifications to hardware representations and lowers them to synthesizable Verilog implementations, paired with multi-level programming interfaces for users.

III. Specifying Accelerators in Stellar

Stellar is composed of a specification language that describes spatial accelerators, dense or sparse, and a compiler that translates these descriptions into Verilog. The specification language is designed to maximize the architect's separation of concerns when designing an accelerator; each of the five subsections below describes a separate design concern which users can specify and explore independently.

A. Functionality

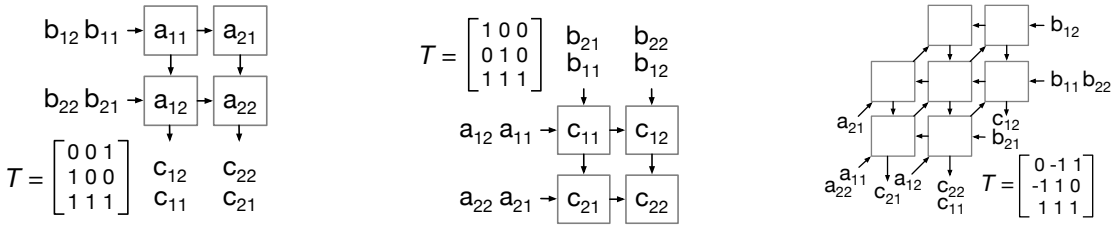
Stellar users specify the functionality of their accelerator with a Halide-like notation which has been used in prior work for dense accelerator design [37]. The functional notation defines various tensor inputs and outputs, and how the outputs are calculated from the inputs.

Consider, for example, Listing 1, which illustrates how a Stellar-user may define the functional behavior of a matrix-multiplication accelerator. We will refer back to this example repeatedly throughout this paper:

Listing 1: Functional specification of a matmul accelerator

```
1  // Inputs
2  a(i, j.lowerBound, k) := A(i, k)
3  b(i.lowerBound, j, k) := B(k, j)
4  c(i, j, k.lowerBound) := 0
5  // Intermediate calculations
6  a(i, j, k) := a(i, j−1, k)
7  b(i, j, k) := b(i−1, j, k)
8  c(i, j, k) := c(i, j, k−1) +
9        a(i, j−1, k) * b(i−1, j, k)
10 // Outputs
11 C(i, j) := c(i, j, k.upperBound)
```

Unlike an iterative for-loop, Stellar's Halide-like notation involves no state-mutations, and makes no assumptions about



(a) Input-stationary dataflow (b) Output-stationary dataflow (c) Hexagonal dataflow

Fig. 2: Examples of space-time-transforms (each named T) and the dense matmul dataflows that result from them.

the order, time, or place of each multiply-accumulate operation within a hardware unit. The i, j, and k indices exist only in what we call the "tensor iteration space," and do not directly correspond to time or space coordinates on a physical hardware accelerator. Neither does the functional notation make assumptions about the sparsity distributions or sparse data formats of the input or output tensors. Sections III-B and III-C describe how we specify such design considerations.

In addition to arithmetic operations, Stellar's functional notation also supports data-dependent accesses to input or output tensors, which are useful for specifying merging and sorting algorithms for sparse workloads. Stellar can therefore be used to construct a full pipeline for both sparse and dense accelerators, including functional units for arithmetic reductions and data-dependent pre- or post-processing.

B. Dataflow

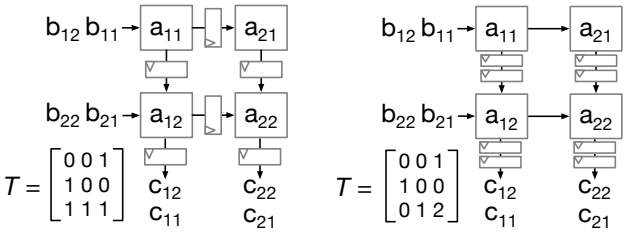
Users define the dataflow of their accelerator by specifying a linear transformation (represented by an invertible matrix) from the tensor iteration space described in Section III-A to physical space and time coordinates on a spatial array. Following the example of prior work [37], we call this linear transformation a "space-time transform". For example, for the matmul example in Listing 1, the space-time transform would be T in the equation below:

$$T \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} x \\ y \\ t \end{bmatrix} \quad (1)$$

where T is a 3×3 invertible matrix, x and y are space coordinates, and t is a timestep. Every input, output, and intermediate MAC operation in the matmul in Listing 1 is mapped by T to a specific place and time on a two-dimensional physical spatial array with x rows and y columns. For example, if T is the identity matrix, then a MAC that takes place when $i = 1$, $j = 2$, and $k = 3$ would be mapped to the PE at position $(x = 1, y = 2)$ (i.e. row-1 and column-2 in the spatial array), and would occur when the time-step, t, equals 3.

Figure 2 illustrates various space-time transforms for matmuls and the spatial arrays that result from them. Note that by simply changing numerical values in the T matrix, users can create a wide variety of spatial arrays, including input-stationary, output-stationary, and hexagonal [4] designs.

Each of these space-time-transforms represents a separate *dataflow*, covering a *superset* of the dataflows proposed by some other dataflow-classification schemes which are instead



(a) (b)

Fig. 3: Different pipelining strategies for the input-stationary matmul accelerator in Figure 2a.

defined in terms of which tensor iterators are spatially or temporally unrolled [36], or by which inputs or outputs remain stationary during execution [5]. For example, a dataflow-classification scheme which only allows users to decide which iterators to spatially unroll could only produce 3D spatial arrays if the user wanted to unroll all three indices (i, j, and k) in Listing 1. Stellar can express 3D arrays as well, but it can also express more niche spatial arrays that such dataflow-classification schemes cannot, such as the hexagonal array in Figure 2c which spatially unrolls all three indices onto a *2D plane*, yielding shorter wires which are easier to route.

Stellar's dataflow specifications also give hardware designers more fine-grained control over lower-level hardware design decisions, such as the number of pipeline registers to place across different axes of the spatial array. Figure 3 illustrates how changing individual values in the lowest row (the $time$ axis) of the dataflow-specification matrix T creates designs that are more or less aggressively pipelined.

Stellar uses the dataflow specified by the user's space-time transform to construct "baseline" dense accelerators which maximize PE-to-PE data re-use, as in Figure 2. Later, Section IV-B describes how these baseline spatial arrays are modified to skip zero-values in sparse workloads, based on the sparsity specifications given next in Section III-C.

C. Sparse Data Structures

For sparse accelerators, the sparse data structures of the input and output tensors are expressed in Stellar in terms of which iterators in the tensor iteration space may be "skipped" and under which conditions they may be skipped. For example, consider the following sparsity structures we define for the matmul example introduced in Listing 1:

Listing 2: Specifying sparse data structures in Stellar

```
// A*B=C where A and B are CSC/CSR
Skip i when A(i,k) == 0
Skip j when B(k,j) == 0
// A*B=C where A is diagonal
Skip i and k when i != k
// A*B=C where rows of A may be all 0
Skip k when A(i,->) == 0
```

Note that in Listing 2, we do not specify how exactly the tensors are stored in memory, or what metadata is associated with them; these details are irrelevant to the spatial array design. Listing 2 only specifies which tensor elements are skipped; e.g. whether we skip elements along rows, as in the CSR format, or along columns, as in the CSC format. By contrast, Section III-E describes how users specify how tensors are actually stored and encoded in memory.

Once a sparsity structure is specified, Stellar determines which PE-to-PE connections in the baseline dense spatial array (as illustrated in Figure 2) are no longer *guaranteed* to transmit useful non-zero values in every single cycle. Under the assumption that these PE-to-PE connections are unlikely to carry useful data (as when the total non-zero density of a tensor is very low), Stellar removes these PE-to-PE connections and replaces them with IO connections that access the input- or output-tensors directly from outer register files.

For example, Figure 4 illustrates how the input-stationary matmul array in Figure 2a will look after the user specifies that the input-B matrix has the CSR format, causing Stellar to remove the vertical PE-to-PE connections which were previously being used to accumulate partial sums.

However, for some forms of structured sparsity, PE-to-PE connections should still retained even if the data they carry will not be used every cycle. For example, Figure 5 illustrates a Stellar-generated spatial array implementing NVIDIA's A100 structured sparsity scheme [1], where two out of every four adjacent DNN weights are zeros. To support such sparsity structures, Stellar provides the `OptimisticSkip` keyword, which, unlike the `Skip` keyword, does not remove PE-to-PE connections, but replaces them with wires that carry small bundles of potentially useful data, rather than scalar values.

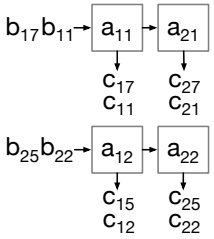


Fig. 4: The input-stationary matmul array from Figure 2a after the B-matrix is specified as a sparse CSR matrix.

D. Load-Balancing

Spatial array workloads are oftentimes extremely imbalanced, causing some PEs to idle while other PEs are performing useful arithmetic operations [13], [14]. Stellar allows users to specify whether they want computations that would normally take place in certain regions of the tensor iteration space to be shifted towards other "target" iterations if the target iterations would be idle otherwise. For example, consider the following load-balancing strategy for a sparse matmul described using Stellar's notation:

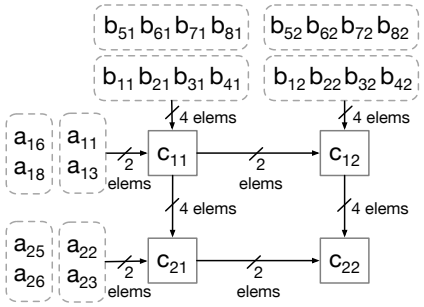
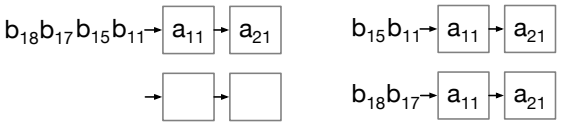


Fig. 5: The output-stationary matrix from Figure 2b when the A-matrix conforms to the A100 2:4 sparsity format [1].



(a) Without load-balancing (b) With load-balancing

Fig. 6: The sparse matmul array from Figure 4, executing an imbalanced B-matrix with and without load-balancing.

Listing 3: A simple load-balancing scheme in Stellar

```
Shift (/*i = */ N -> 2*N, j, k) to
      (/*i = */ 0 -> N, j, k+1)
```

where the tensor iterators, i, j, and k, are the same as those introduced for the matmul in Listing 1. For any iterator value $k = K$, if the target matmul iterations, where $0 \leq i < N$, are *all* idle due to a workload imbalance, then Stellar will shift future work that has not yet begun where $k = K + 1$ and $N \leq i < 2N$ onto the idle PEs.

Based on the exact dataflow specified, this might mean, for example, that *only* direct adjacent rows of the spatial array can share work, as illustrated in Figure 6. More flexible load-balancing schemes can share work across broader sets of PEs, but, as detailed later in Section IV-B, they may require Stellar to generate hardware with greater area and wiring congestion.

More fine-grained and sophisticated load-balancing schemes can also be specified. Listing 4 shows an example load-balancing scheme where only iterations corresponding to a small subset of PEs will take work from other PEs:

Listing 4: Very flexible load-balancing for a limited set of PEs.

```
Shift (i,j,k) to (/*i=*/0, /*j=*/0->4, k)
```

E. Private Memory Buffers

To generate scratchpad memories and private memory buffers for their accelerators, Stellar users must specify the specific dense or sparse data formats they will support. To specify such data formats, we use the fibertree notation from

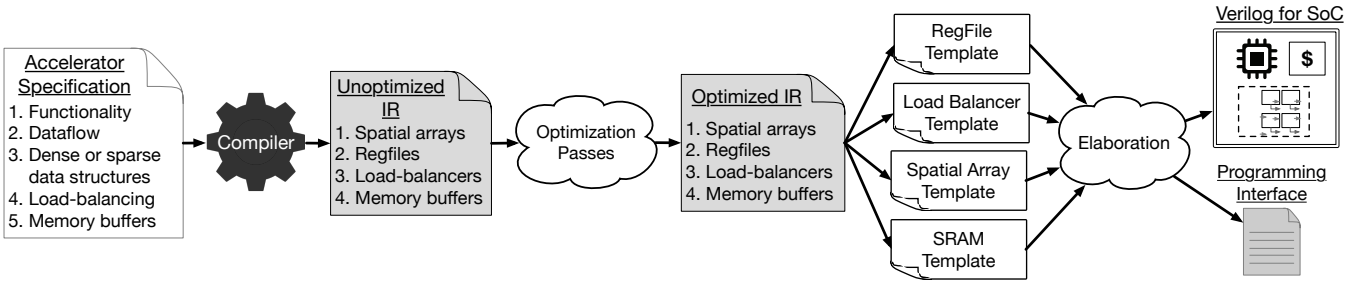


Fig. 7: An overview of the hardware generation process for Stellar, from the initial architectural specification, to the unoptimized and optimized IRs, to the final Verilog and programming interface outputs.

prior work [31], where users specify a different dense/sparse format for every axis (i.e. dimension) of a tensor. For example, the CSR format would be specified by setting the outer-axis of a two-dimensional matrix to the *Dense* uncompressed format, while the innermost axis would be *Compressed*, composed of a list of coordinates and data. Stellar supports other formats as well, such as *Bitvectors* and *Linked-Lists*; by composing these formats to different dimensions of a tensor, a wide variety of unique sparse tensor formats can be defined.

IV. Hardware Generation in Stellar

After an accelerator's functionality, dataflow, sparse data structures, load-balancing schemes, and private memory parameters are specified in Stellar, our compiler elaborates these into an IR which represents a set of spatial arrays, register files, SRAMs, and load-balancers. These are optimized based on data-access patterns that can be determined at elaboration time, and the optimized IR is mapped to a set of Chisel [3] templates which are lowered into Verilog. Figure 7 illustrates this full hardware generation process.

A. Architectural Overview

Stellar-generated accelerators are composed of spatial arrays, register files, private memory buffers, (optional) load-balancers, and a DMA. Figure 8 shows the overall hardware architecture of an example Stellar-generated accelerator that performs sparse matrix multiplications and merges the scattered partial sums into merged matrix results.

Spatial arrays perform compute operations such as matrix-multiplications or the merging of scattered output results. These spatial arrays read and write their input and output tensors to register files, which may themselves be populated by or emptied into larger private memory buffers. Load-balancers monitor the regfile inputs and outputs to determine whether PEs will be idle or over-utilized. Finally, a DMA transfers tensors between off-chip DRAM or outer caches and the accelerator's private memory buffers. The following subsections describe how the aforementioned hardware components are generated and optimized by Stellar.

B. Spatial Arrays

Stellar initially constructs *dense* spatial arrays, based on the functional description of the accelerator and its dataflow. The PEs of these spatial arrays will request inputs or issue outputs when their physical coordinates and current time-step correspond to the indices these inputs/outputs are supposed to occur at. For example, the PE output on line 11 of Listing 1 occurs whenever the tensor iterator k is equal to `k.upperBound`. By multiplying a PE's space-time coordinates, (x, y, t) as in Equation 1, by the inverse of the space-time transform, T^{-1}, we can find the corresponding value of k; if it is `k.upperBound`, then the PE output will be issued.

Figure 9a illustrates how an example spatial array is initially represented in the Stellar compiler's internal IR, based purely on the functional description in Listing 1, before the dataflow or sparsity specifications are applied to it. Stellar refers to this IR as an `IterationSpace`, in which every `Point` corresponds to a different set of values for the tensor iterators (i, j, k). Furthermore, the `IterationSpace` includes a set of `Point2PointConns` (point-to-point connections) describing data dependencies between different points, and a set of `IOConns` (IO connections) representing input- or output-requests to external register files (described later in Section IV-D). Finally, every `Point` has a set of `Assignments` representing the arithmetic operations it must perform.

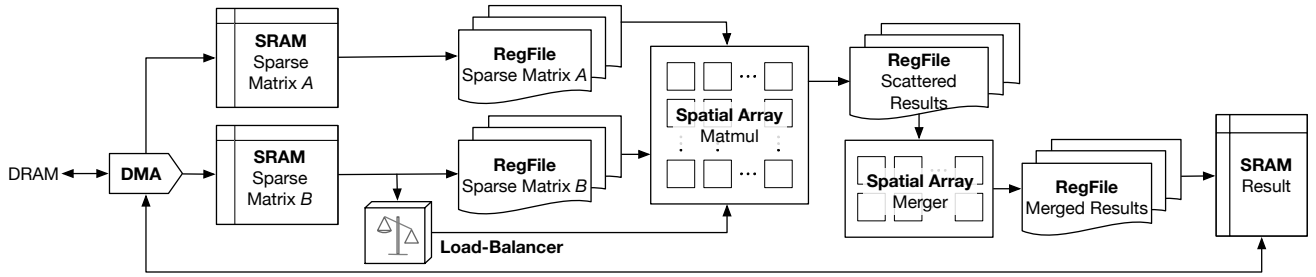
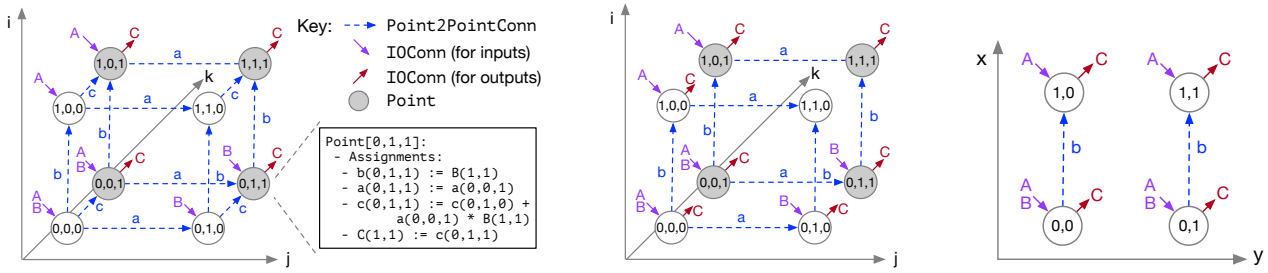


Fig. 8: Hardware architecture overview for an example sparse matrix-multiplication accelerator.



(a) After functional description. (b) After sparsity pruning. (c) After dataflow.

Fig. 9: The internal representation, called an `IterationSpace` for an accelerator as it is transformed from a purely functional description to a physically realizable two-dimensional spatial array.

The baseline, dense spatial array that is initially constructed by Stellar is modified based on the user's sparsity structure specifications. Dense spatial arrays typically achieve high data reuse by sharing data through PE-to-PE connections. However, sparse workloads often have much less data reuse, rendering many of these PE-to-PE connections obsolete. Stellar will remove the PE-to-PE connections which are no longer guaranteed to carry useful data between PEs, and replace them with direct connections to outer regfiles, as seen by the change between Figure 2a and Figure 4.

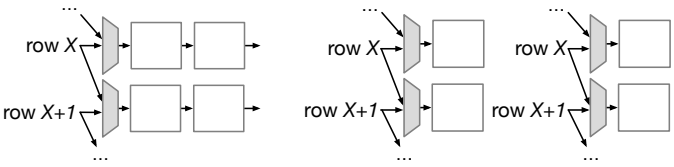
For an example of how Stellar determines which connections to remove, consider the accumulation of partial sums on lines 8-9 of Listing 1. We see that a PE at point (i, j, k) in the tensor iteration space computes the multiply-accumulate-sum, $c(i, j, k)$, based on $c(i, j, k-1)$, which means that the "difference vector" [37] for the variable c is $(\Delta i = 0, \Delta j = 0, \Delta k = 1)$. Multiplying the input-stationary space-time-transform T in Figure 2a by the difference vector yields the spacetime difference vector $(\Delta x = 1, \Delta y = 0, \Delta t = 1)$, which indicates that the partial sums travel vertically down the spatial array every time-step.

Now, suppose that B is in the CSR format, as in Listing 5:

Listing 5: Making the B-matrix CSR

```
1  Skip j when B(k,j) == 0
```

In the CSR sparse format, finding the j-coordinate would require a series of indirect lookups. Stellar abstracts these lookups away by expressing the "expanded" j-coordinate as some arbitrary function f whose inputs are k and the *compressed* j-coordinate: $j_{expanded} = f(k, j_{compresssed})$. Therefore, the difference vector for c now becomes $(\Delta i = 0, \Delta j_{expanded} = f(k, j_{compressed}) - f(k-1, j_{compressed}), \Delta k = 1)$. Because the j-component depends on indirect data lookups and can no longer be simplified into a scalar constant, Stellar can no longer assume that the partial sums will be unconditionally accumulated vertically across the spatial array. These vertical PE-to-PE connections are therefore removed, yielding the matmul array in Figure 4 with fewer PE-to-PE connections but more ports to outer register files. Figure 9b illustrates how the spatial array's `IterationSpace`, appears after its `Point2PointConns` are pruned based on the equations described above.



(a) Balancing across entire rows (b) Balancing across each PE

Fig. 10: More or less flexible load-balancing strategies.

Load-balancing schemes can also affect the design of spatial arrays. Figure 10b illustrates a load-balancing strategy where *any* PE within a row can operate on data that would otherwise have been sent to the upper row, if that PE would have otherwise been idle. Each PE can independently be redistributed work from the above row, and therefore it might no longer receive useful data along its horizontal PE-to-PE connections. Therefore, since horizontal PE-to-PE connections might no longer transmit the necessary inputs, Stellar must replace them with connections to outer regfiles. Contrast this to Figure 10a, where load balancing operates at the granularity of an entire row of PEs, preserving the horizontal connections.

After `Point2PointConns` have been pruned and replaced with `IOConns` based on the sparsity and load-balancing specifications, the dataflow space-time transform is applied to generate a final `IterationSpace`, as in Figure 9c. Each `Point` in this new `IterationSpace` represents a different PE in the final generated spatial array. Multiple `Points` in Figure 9b may map to the same `Point` in Figure 9c if they represent different timesteps for the same PE.

Finally, every `Point` in Figure 9c is mapped to a Chisel template of a PE, shown in Figure 11. Every `Assignment` of the `Point`, such as a multiply-accumulate operation, is translated to Chisel in the "User-Defined Logic" block. How-

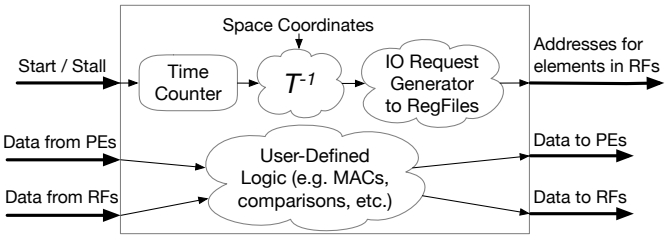


Fig. 11: The architecture for a Stellar PE.

ever, a PE may perform different operations in different time-steps; for example, the variable c is initialized to 0 on line 4 of Listing 1, but is thereafter accumulated every subsequent cycle on line 8. To calculate *which* operation to perform at a specific time-step, every PE includes a "Time Counter" register; when concatenated to the physical coordinates of the PE, a space-time vector, (x, y, t), can be generated and multiplied by the inverse of the space-time transform, T^{-1} to generate the original tensor iterators (i, j, k). Input- and output-requests to outer register files are generated in the "IO Request Generator."

C. Private Memory Buffers

As described in Section III-E, Stellar users specify the dense or sparse data formats, capacities, and bandwidths their private memory buffers will support. The data formats are defined using the fibertree notation, where different dense/sparse formats are defined for every axis (i.e. dimension) of a tensor.

Stellar then generates multiple pipeline stages — one for each axis of the dense or sparse tensors that the buffer stores — which read/write requests made by programmers pass through. *Dense* axes generate simple address generators, while *Compressed*, *Bitvector*, or *Linked-List* axes may require indirect lookups to SRAMs which store metadata to determine the final data addresses to read or write to. Figure 12 shows example pipeline stages generated for a private memory buffer holding tensors in the block-CSR [9] format.

For every read or write request, the programmer must specify at runtime the address, length, and strides for each axis. However, users can also optionally hardcode certain read/write request parameters before hardware generation begins in order to help Stellar make optimizations to both the memory buffers and For example, Listing 6 shows how a Stellar user hardcodes certain parameters to specify that a dense tensor memory buffer will always produce 4×4 dense matrices, as illustrated in Figure 13a. The address-generators in the memory buffer can be simplified based on these hardcoded parameters. More importantly, however, since the order in which the matrix elements are generated is now known to Stellar's compiler ahead of runtime in Figure 13a, important optimizations can be made to register files as described next in Section IV-D.

Listing 6: Hardcoding memory buffer read parameters

```
1  def hardCoded(x: MemPipeline) = Map(
2    x.read_req.spans(0) -> 4.U,
3    x.read_req.spans(1) -> 4.U,
4    x.read_req.data_strides(0) -> 1.U,
5    x.read_req.data_strides(1) -> 4.U)
```

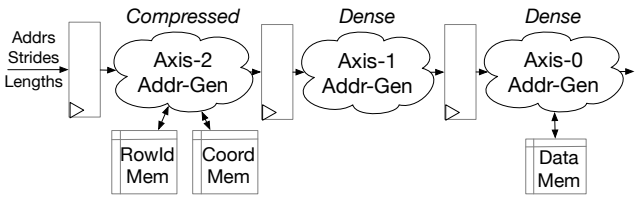
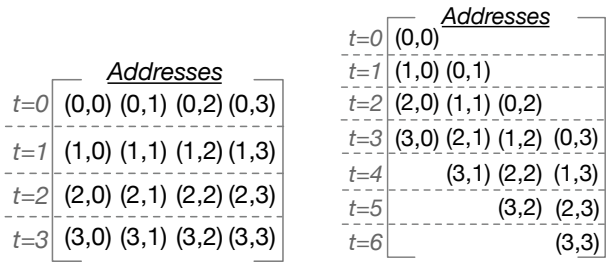


Fig. 12: The read/write pipeline stages for a private memory buffer holding block-CSR [9] tensors.



(a) Generated by memory buffer. (b) Consumed by spatial array.

Fig. 13: The hardcoded order of the addresses of the elements generated by a memory buffer or consumed by a spatial array,

D. Register Files

All spatial arrays in Stellar read from and write into *register files* (regfiles). For the matmul example in Listing 1, every input and output variable (A, B and C) must be stored in a separate register file before being accessed by the spatial array.

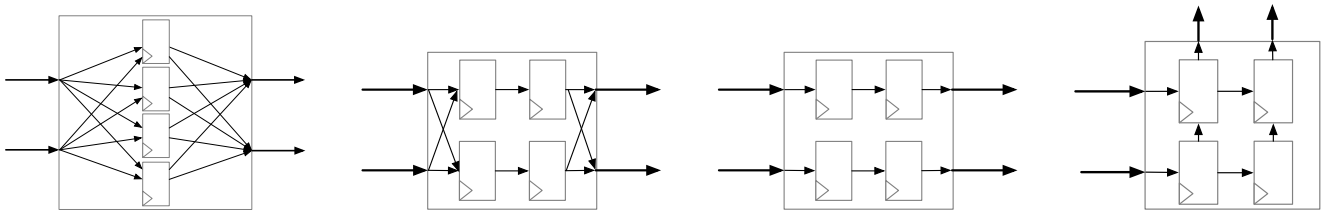
However, the default, baseline register file design in Stellar is quite expensive, as illustrated in Figure 14a. Every input port and output port has access to *all* entries in the register file simultaneously, and outputs are performed by searching the coordinates of all entries.

The baseline register file design is expensive because Stellar's functional specifications (described in Section III-A) are highly flexible and support indirect accesses whose coordinates may not be known until runtime. The baseline design therefore functions as a worst-case fallback for spatial arrays with complicated and unpredictable regfile access patterns. Fortunately, for most accelerators, Stellar's optimization passes can enormously reduce these overheads.

For example, to reduce the overhead of each individual input or output port, optimization passes might determine that it is sufficient for inputs and outputs to occur only at the *edges* of a regfile as in Figure 14b. Further optimizations can narrow the number of elements that need to be searched even further, as in Figure 14c, where each output port observes only a single element of the register file, to produce a simple feed-forward array of shift registers. By selecting *which* edges to designate as entry and exit points for the regfile, Stellar regfiles can even perform various data layout transformations, such as transpositions, as illustrated by Figure 14d.

To make all these optimizations, Stellar observes the order in which inputs are requested and outputs are produced by a spatial array depending on its dataflow, or by a private memory buffer based on its hardcoded parameters.

For example, consider a register file which buffers inputs in between a dense memory buffer storing the dense matrix B from Listing 1, and a spatial array with the dataflow in Figure 2b. Based on the `IOConns` in the spatial array's `IterationSpace`, described above in Section IV-B, Stellar's compiler can determine that elements of B are consumed



(a) Unoptimized baseline regfile (b) Inputs/outputs only at *edges* (c) Each I/O port connecting to only a single regfile entry at edges (d) Regfile tranposing inputs

Fig. 14: Various register files generated by Stellar, with more or less aggressive optimizations. Observe that when input/output ports can only connect to regfile *edges*, elements must travel through the regfile entries so they can reach the output ports.

by the spatial array in the order shown in Figure 13b; furthermore, if the memory buffer's read parameters are hardcoded as in Listing 6, Stellar's compiler can determine that elements of B exit the memory buffer in the order shown previously in Figure 13a. Stellar's compiler includes a number of hand-written optimization passes; one of these passes checks if the inputs to a regfile always enter it in the exact same order that they exit it. If so, that regfile's entry and exit options are set such that it matches the efficient feedfordward regfile shown in Figure 14c.

Similar optimization passes exist for other spatial arrays with indirect accesses, or regfiles which perform transpositions. Stellar's compiler performs such optimization passes sequentially, checking if progressively less efficient regfiles can be generated, until finally falling back on the baseline, default regfile when no potential optimizations can be found.

E. Load Balancers

To support load-balancing, Stellar generates load-balancer modules which monitor regfile inputs to determine whether the PEs that read from those regfiles have enough inputs available to do useful work, or whether they will be idle.

Once the load-balancers determine that work should be redistributed between PEs, they calculate *space-time biases* to apply at runtime to the space-time transforms of the PEs to which work will be redistributed. A space-time bias is a vector addition, as seen in Equation 2, which modifies the unbalanced space-time transform in Equation 1 from Section III-B:

$$T\left(\begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) = \begin{bmatrix} x \\ y \\ t \end{bmatrix} \tag{2}$$

where b_1, b_2, and b_3 are scalar offsets which are calculated based on the user's load-balancing specification. When the space-time bias gives the PEs new space-time coordinates at runtime, they behave as if they were other PEs in other parts of the spatial array, allowing them to take some of their workload.

F. Limitations

Although Stellar's front-end specification language and backend hardware generator support a wide variety of sparse and dense accelerator designs, the existing tool is limited in its ability to express or generate certain sophisticated cache hierarchies found in prior work [38], [39]. Stellar's private memory buffers are explicitly managed by the programmer; however some accelerators benefit most from hardware-managed caches with unusual eviction policies [39]. Fortunately, this limitation is mitigated to a degree by Stellar's integration with the Chipyard [2] framework, which can provision Stellar-generated SoCs with large L2 caches which can be shared by both CPUs and accelerators, although support for custom eviction or prefetching policies are left for future work.

For spatial arrays on the other hand, Stellar's dataflow descriptions (Section III-B) can currently only express *affine* transformations, and cannot express recursive or hierarchical transformations such as tree-reductions. However, our functionality specification language (Section III-A) is still general enough that such compute structures can be manually implemented by Stellar's users, though at the cost of blurring the separation of concerns between the functional behavior of an accelerator and the scheduling of operations spatially or temporally on it. For example, we were able with Stellar to express the complex hierarchical mergers described in SpArch [39]. After synthesis, we found that these mergers consumed 13× the area of simpler, non-hierachical mergers from OuterSPACE [26]. Therefore, even for recursive operations which don't map easily to Stellar's dataflow abstractions, our experience indicates that such designs can still be specified by the user and explored for area or performance tradeoffs.

V. Programming Interface

Stellar-generated accelerators are programmed using custom RISC-V instructions, summarized in Table II. All instructions revolve around data transfers from one memory unit to another; for example, from DRAM to a private memory buffer, or from a memory buffer to a register file. Spatial arrays begin execution as soon as their input register files are filled.

When moving data from a source memory to a destination memory, users set certain values, such as addresses, strides, and fibertree axis types [31] for the source and destination. For example, to move a sparse CSR matrix from DRAM into a private memory buffer, programmers specify the addresses of the matrix's data and metadata arrays in DRAM, as well as the address within the accelerator's private memory that the data will be copied to. To illustrate, Listing 7 shows two code snippets in C using Stellar's ISA: one moving a dense matrix from DRAM into a private memory buffer called `SRAM_A`, and another which moves a CSR matrix into `SRAM_B`.

Opcode	Rs1[19:16]	Rs1[15:0]	Rs2
`set_address`	For src, dst, or both	Axis	DRAM/SRAM address, or regfile
`set_span`	For src, dst, or both	Axis	Number of elements to move
`set_data_stride`	For src, dst, or both	Axis	Stride
`set_metadata_stride`	For src, dst, or both	Axis and metadata type (e.g. `ROW_ID` or `COORD`)	Stride
`set_axis_type`	For src, dst, or both	Axis	"Dense", "Compressed", etc.
`set_constant`	N/A	ID of scalar or boolean constant to set: e.g. `should_trail_reads`, `should_interleave`, etc.	True/false if boolean, scalar integer otherwise

TABLE II: A representative subset of the commands in Stellar's 64-bit RISC-V ISA

Listing 7: Moving matrices from DRAM into local memory.

```
// Moving in dense matrix
float matrix_A[DIM][DIM];

set_src_and_dst(DRAM, SRAM_A);

set_data_addr(FOR_SRC, matrix_A);

for (int axis = 0, axis < 2; axis++) {
  set_span(FOR_BOTH, axis, DIM);
  set_axis(FOR_BOTH, axis, DENSE);
}

set_stride(FOR_BOTH, /*addr-gen-axis=*/0, 1);
set_stride(FOR_BOTH, /*addr-gen-axis=*/1, DIM);

stellar_issue();

// Moving in CSR matrix
float matrix_B_data[DATA_SIZE];
int matrix_B_coords[DATA_SIZE];
int matrix_B_row_ids[N_ROWS];

set_src_and_dst(DRAM, SRAM_B);

set_data_addr(FOR_SRC, matrix_B);

set_metadata_addr(FOR_SRC, /*axis=*/0, ROW_ID,
  matrix_B_row_ids);
set_metadata_addr(FOR_SRC, /*axis=*/0, COORDS,
  matrix_B_coords);

set_span(FOR_BOTH, /*axis=*/0, ENTIRE_AXIS);
set_span(FOR_BOTH, /*axis=*/1, N_ROWS);

set_stride(FOR_BOTH, /*addr-gen-axis=*/0, 1);
set_metadata_stride(FOR_BOTH,
  /*addr-gen-axis=*/0, /*axis=*/0,
  COORDS, 1);
set_metadata_stride(FOR_BOTH,
  /*addr-gen-axis=*/1, /*axis=*/0,
  ROW_IDS, 1);

set_axis(FOR_BOTH, /*axis=*/0, COMPRESSED);
set_axis(FOR_BOTH, /*axis=*/1, DENSE);

stellar_issue();
```

VI. Evaluation

Stellar allows users to efficiently express state-of-the-art accelerator designs, and then automatically synthesizes RTL for them comparable in performance and area consumption to hand-designed accelerators. By generating actual hardware, Stellar also enables architects to make insights into performance bottlenecks which are caused by low-level interactions between hardware, memory layouts, and data distributions which are not always visible in higher-level simulators.

A. Methodology

To demonstrate that Stellar-generated designs are competitive with hand-written implementations, we generate two DNN accelerators from prior work: a dense DNN accelerator modeled after Gemmini [12], which performs convolutions and 8-bit quantized matrix multiplications with a 16×16 weight-stationary systolic array, and SCNN [28], which targets convolutional networks which have been pruned for unstructured weight and activation sparsity. Using cycle-accurate simulators [18], we compare the performance of both the hand-written and Stellar-generated implementations on the DNN workloads they were originally evaluated on in prior work: an end-to-end ResNet50 [15] inference for Gemmini, and AlexNet [20] for SCNN. For area and frequency comparisons, we synthesize designs using the ASAP7 PDK, and we evaluate energy consumption on Joules using Intel 22nm.

B. Performance and Area Overheads

The Stellar-generated Gemmini accelerator achieved 90% of the utilization of the handwritten Gemmini accelerator when both were synthesized to 500 MHz, as shown by Figure 16a. However, the Stellar-generated accelerator was successfully synthesized at up to 1 GHz, while the handwritten Gemmini could only reach 700 MHz. The handwritten Gemmini includes complicated, centralized loop-unrollers, whose address generators failed to meet timing at higher frequencies; Stellar's more distributed memory-buffer address-generators were more scalable.

Furthermore, the Stellar-generated Gemmini accelerator only consumed 13% more area than the hand-designed accelerator when both were synthesized to 500 MHz, as shown in Table III, demonstrating that Stellar's support for sparse accelerators does not compromise the competitiveness and efficiency of the dense accelerators that it generates. The area overhead for the matmul array comes partially from the larger amount of internal state in a Stellar-generated PE (such as the "time" register in Figure 11), compared to handwritten Gemmini PEs which have no internal counters. Furthermore, Stellar-generated spatial arrays include global signals that start and stall all PEs simultaneously. While this is useful for many workloads, it is not needed in Gemmini-like workloads where the memory buffers consuming partial sums from the matmul array will always be ready to consume spatial array outputs. These long global signals add further area overhead.

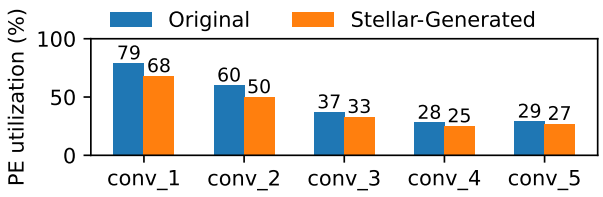
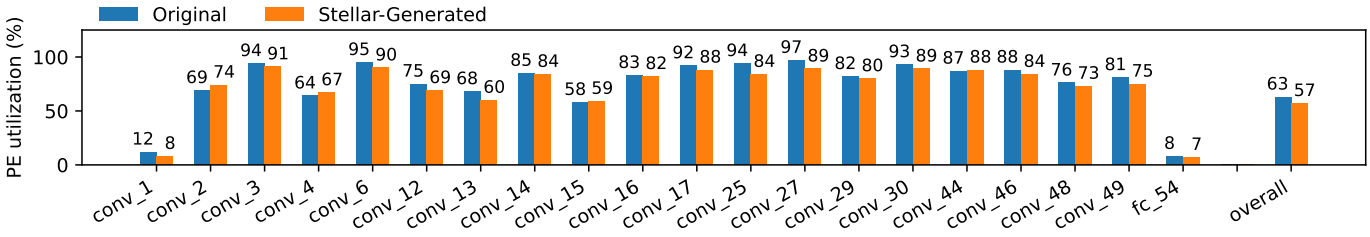
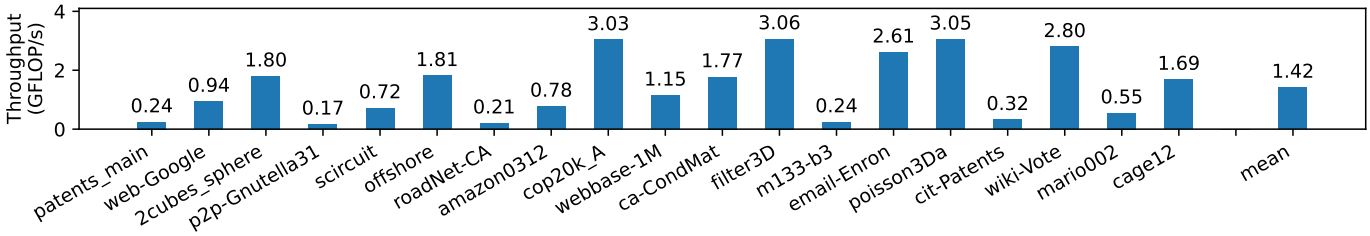


Fig. 15: PE utilization of SCNN on AlexNet.



(a) The PE utilization of both the handwritten and Stellar-generated Gemmini accelerators on ResNet50.



(b) Throughput of the initial Stellar-generated OuterSpace accelerator on SparseSuite.

Fig. 16: The performance of Stellar-generated accelerators on dense and sparse workloads.

	Original		Stellar-Generated	
	Area (μm^2)	Area (%)	Area (μm^2)	Area (%)
Matmul array	334K	10%	420K	11%
SRAMs	2,225K	68%	2,247K	61%
Regfiles	25K	1%	104K	3%
Loop unrollers	259K	8%	482K	13%
Dma	102K	3%	109K	3%
Host CPU	337K	10%	337K	9%
Total	3,282K	*100%*	3,699K	*100%*

TABLE III: Area comparison between Gemmini accelerators.

Stellar's power overhead ranges from 7% at best to 30% at worst compared to the handwritten Gemmini on various layers of ResNet50, as illustrated in Figure 17 when both were synthesized to 500 MHz with the Intel 22nm node.

Finally, as illustrated in Figure 15, the Stellar-generated SCNN achieved 83%-94% of the hand-designed accelerator's reported performance when executing a sparse, pruned model of AlexNet. SCNN has a sophisticated design including a four-dimensional PE topology, conversions back-and-forth between dense and sparse data formats, and a heavily distributed memory system, but Stellar's abstractions easily covered this design space while Stellar's hardware generation flow delivered efficient and programmable RTL.

C. Identifying Sparse Accelerator Performance Bottlenecks

In addition to generating performant and efficient accelerators, Stellar also enables architects to discover performance bottlenecks which are only visible on real-world hardware implementations. To illustrate, we generate a sparse matrix-multiplication accelerator based upon OuterSPACE from prior work [26], and we show how Stellar can provide insights into how scattered accesses to pointers in DRAM can limit total performance for certain irregular workloads.

Our Stellar-generated accelerator initially achieved an average throughput of only 1.42 GFLOP/s while performing the highly-sparse matmuls that OuterSPACE was originally evaluated on [8]; as shown in Figure 16b. OuterSPACE's paper

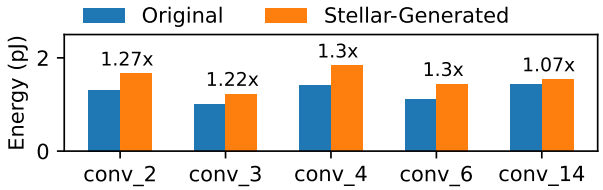


Fig. 17: Energy consumed per MAC on layers of ResNet50.

does not report performance for individual matmuls, but the average throughput reported – 2.9 GFLOP/s – was noticeably higher than that of the initial Stellar-generated one.

The Stellar-generated accelerator's performance was severely impacted by scattered accesses made to DRAM when reading and writing partial sums. OuterSPACE stores partial sums as small contiguous vectors scattered through DRAM. To access a partial sum vector, the pointer to that vector must first be read non-contiguously from DRAM. Despite comprising less than 10% of the total memory traffic in a typical matmul, we found that accesses to these pointers initially posed a severe memory bottleneck for the accelerator, due to control-dependencies imposed by the pointer accesses. Any inefficiency in their reads or writes caused further latency-sensitive stalls in the accelerator's DMA.

This issue is compounded by simpler DMAs, such as Stellar's default DMA, which can only make *one* new memory load/store request per cycle. Such DMAs are sufficient for accelerators such as SCNN. However, when accessing scattered *pointers*, one read request can only return a single scalar pointer, causing costly stalls and underutilized bandwidth.

Fortunately, Stellar's rapid generation of real RTL makes it easy for users to explore and evaluate such design complexities without hand-writing complicated, high-fidelity simulators. Furthermore, Stellar's strong separation of concerns enabled us to mitigate this bottleneck with only minor modifications to the DMA, without any changes to the memory buffers or spatial arrays. We updated the DMA to generate up to 16 independent

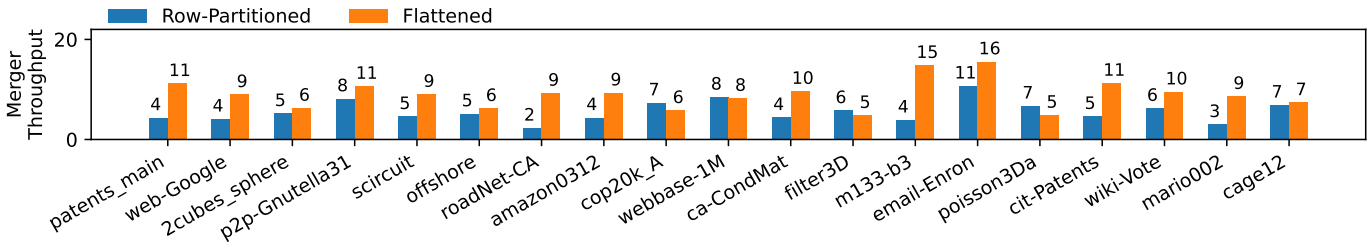
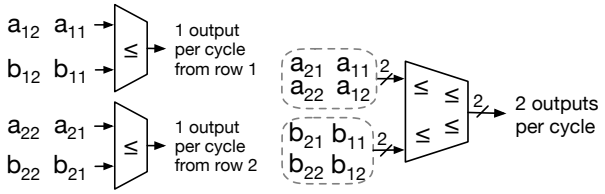


Fig. 18: The number of merged elements generated every cycle by both row-partitioned and flattened mergers when merging partial matrices with SpArch's proposed execution order [39].



(a) Row-partitioned mergers. (b) Flattened merger.

Fig. 19: Spatial arrays that merge scattered partial matrices.

DRAM read requests per cycle without changing total DRAM bandwidth, increasing our throughput to 2.1 GFLOP/s.

D. Area and Performance Tradeoffs in Sparse Mergers

Stellar's generation of real RTL also enables architects to investigate performance, area, and hardware efficiency tradeoffs which cannot be explored by more abstract, higher-level simulators. To illustrate, in this section, we show how Stellar can be used to significantly reduce the area of partial matrix mergers without compromising their performance on a variety of sparse matrix multiplications.

Prior work [26], [38], [39] introduces various spatial arrays that *merge* the scattered partial matrices produced by sparse matrix multiplier arrays. Some works, such as GAMMA [38], merge each row of a partial matrix on a different PE, each generating one element every cycle as shown in Figure 19a. Other accelerators, such as SpArch [39], do not partition merging tasks in this way, but instead flatten the different rows in a partial matrix into a single contiguous fiber, and pop *multiple* elements from this fiber every cycle, as in Figure 19b.

The mergers which operate on different rows separately, such as with GAMMA, take up far less area than the ones which flatten partial matrices, such as the one used in SpArch. For example, SpArch's mergers consume over 60% of it's area, with 128 64-bit comparators used for a maximum throughput of 16 elements per cycle, while GAMMA-like mergers, when synthesized with Stellar, consume 13× less area. However, the cheaper, row-partitioned mergers are more sensitive to imbalances in the lengths of different rows of the partial matrices. SpArch's loop execution order generates many small partial matrices which can have highly imbalanced row-lengths, causing severe underutilization on GAMMA-like mergers.

The original SpArch work [39] does not explore the potential performance and area tradeoffs of using cheaper, row-partitioned mergers at the cost of greater sensitivity to row-length imbalances. However, for accelerators with severe area or resource constraints, this trade-off may be worthwhile.

To investigate, we generated row-partitioned, low-area mergers for SpArch with a maximum throughput of 32, and compared their performance to the more expensive flattened mergers from the handwritten accelerator, which have a smaller maximum throughput of 16 but more comparators in total in the mergers. As shown in Figure 18, the row-partitioned mergers achieve at least 80% of the flattened merger's performance on over a third of the SuiteSPARSE matrices that SpArch was tested on in its original publication. In fact, on four of the matrices, the smaller, row-partitioned merger performed *better* than the larger, flattened merger from the original SpArch work, due to the row-partitioned merger's greater maximum theoretical throughput. Architects who face area constraints and who expect that the matrices they merge will be similar to `poisson3Da` or `cop20k_A`, may prefer the row-partitioned mergers when building accelerators that merge matrices in the same order that SpArch does.

As noted previously in Section IV-F, SpArch's flattened mergers, illustrated in Figure 19b, are not the best fit for Stellar's dataflow specification language. However, despite this limitation, Stellar's functionality specification language was still generalizable enough to enable SpArch's flattened mergers to be implemented so that they could be compared to the simpler, row-partitioned mergers more commonly used in accelerators such as GAMMA. Furthermore, Stellar's compiler was capable of generating the RTL for the memory buffers, regfiles, DMAs, and programming interfaces necessary to run these matrix merging and sorting workloads without writing custom Verilog for hardware components or testbenches.

VII. Conclusion

Stellar enables the rapid design, exploration, and generation of both dense and sparse spatial accelerators, by allowing architects to cleanly separate the different concerns that go into designing an accelerator, and then generating synthesizable RTL implementations and software interfaces which are comparable to hand-written designs from prior work. Our codebase, including our frontend specification language, compiler and optimization passes, and software libraries, are open-sourced. Stellar is also fully compatible with the Chipyard [2] chip design framework, enabling users to integrate their designs into complete, programmable SoCs.

REFERENCES

[1] "NVIDIA A100 Tensor Core GPU Architecture V1.0," NVIDIA, Tech. Rep.

[2] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.

[3] 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," in *DAC*, 2012.

[4] M. Bekakos, I. Milovanović, E. Milovanović, T. Tokić, and M. Stojčev, "Hexagonal systolic arrays for matrix multiplication," in *Highly parallel computaions: algorithms and applications*, 2001, pp. 175–209.

[5] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 44, no. 3, pp. 367–379, 2016.

[6] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[7] 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, p. 924–939. [Online]. Available: https://doi.org/10.1145/3352460.3358276

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

[9] J. Dongarra, "Block Compressed Row Storage (BCRS)," https://netlib.org/linalg/html_templates/node93.html, 1995, accessed: 2023-11-21.

[10] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *ISCA*, 2015.

[11] A. Einstein *et al.*, "The foundation of the general theory of relativity," *Annalen Phys*, vol. 49, no. 7, pp. 769–822, 1916.

[12] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[13] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. Herbordt, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," 2020.

[14] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture*, 2021, pp. 1051–1063.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[16] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 710–726.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017.

[18] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *ISCA*, 2018.

[19] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis *et al.*, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 296–311.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[21] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang, Y. Zhang, J. Cong, N. George, J. Alvarez, C. Hughes, and P. Dubey, "Susy: A programming model for productive construction of high-performance systolic arrays on fpgas," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415644

[22] L. Liu, Z. Qu, Z. Chen, Y. Ding, and Y. Xie, "Transformer acceleration with dynamic sparse attention," 2021.

[23] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "VTA: An Open Hardware-Software Stack for Deep Learning," *CoRR*, 2018.

[24] N. Nayak, T. O. Odemuyiwa, S. Ugare, C. Fletcher, M. Pellauer, and J. Emer, "Teaal: A declarative framework for modeling sparse tensor accelerators," *arXiv preprint arXiv:2304.07931*, 2023.

[25] NVIDIA, "Nvidia deep learning accelerator," http://nvdla.org/, 2019, accessed: 2019-08-1.

[26] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[27] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[28] 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," *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.

[29] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[30] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 766–780.

[31] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. Springer, 2020.

[32] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 97–110.

[33] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 93–104.

[34] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 268–281.

[35] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 232–234.

[36] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 369–383.

[37] E. Zehendner, "Systolic systems," in *Algorithms of Informatics*, 2005, vol. 2, ch. 12.

[38] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 687–701.

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

[40] A. Zhou, Y. Ma, J. Zhu, J. Liu, Z. Zhang, K. Yuan, W. Sun, and H. Li, "Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch," 2021.