Evaluating Message-Passing Abstractions for High-Level Hardware Design

by

Ayana K. Alemayehu

S.B. Electrical Engineering and Computer Science, MIT, 2025.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026

© 2026 Ayana K. Alemayehu. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Ayana K. Alemayehu
Department of Electrical Engineering and Computer Science
May 15, 2026

Certified by: Rachit Nigam
Assistant Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair
Master of Engineering Thesis Committee

Evaluating Message-Passing Abstractions for High-Level Hardware Design

by

Ayana K. Alemayehu

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Developing with low-level Hardware Description Languages (HDLs) like SystemVerilog can be time consuming, error prone, and require careful consideration of the target hardware platform. High Level Synthesis (HLS) toolchains attempt to reduce the burden of this workload through high level abstractions and auto-pipelining. XLS is a HLS library aiming to streamline this development process. However, there are few open source XLS projects that display its capabilities in real world scenarios and evaluate the performance of the produced design. HiSparse, a modular sparse matrix-vector (SpMV) accelerator, serves as a practical implementation target for XLS as it includes dynamic hardware computations often thought too complicated for HLS toolchains. We present a case study of XLS development through the implementation of the HiSparse architecture and its synthesis on a Zynq 7000 series FPGA, identifying general XLS programming methods to optimize performance. We overcome compiler code generation failures through the channel multiplexing technique, and achieve full streaming throughput by splitting modules across blocking frontiers. Our contributions inform future users of the language as well as highlight potential language extensions to improve the design process.

Thesis supervisor: Rachit Nigam
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor, lab, friends and family for their continuous support during my MEng. I am extremely grateful to Chris Leary and Mark Heffernan for taking the time to meet with me and provide support with XLS programming. This work would not at all been possible without their generosity. Finally, I would like to thank the EECS Communication Lab, Undergraduate Office, and all other support systems at MIT who help make such a daunting task approachable.

Contents

List of Figures 9

List of Tables 11

1 Introduction 13

1.1 Motivation . . . 13

1.2 Contributions . . . 14

2 Background and Related Work 15

2.1 XLS Overview . . . 15

2.2 Sparse Matrix Computations . . . 18

2.3 HiSparse Architecture Overview . . . 19

2.3.1 High Level Design . . . 20

2.3.2 Custom Sparse Matrix Format and Memory Layout . . . 22

2.3.3 Key Modules . . . 24

3 Implementation 31

3.1 The Naive Implementation . . . 31

3.2 The Codegen Implementation . . . 36

3.3 The Optimized Implementation . . . 39

3.3.1 Channel Multiplexing . . . 40

3.3.2 Frontier Splitting . . . 41

4 Evaluation 45

4.1 Simulation Setup . . . 45

4.2 Simulation Results . . . 47

4.3 Hardware Setup . . . 49

4.4 Hardware Results . . . 51

5 Conclusion 55

References 59

List of Figures

2.1 Kahn Process Network 18
2.2 Example CSR Matrix 19
2.3 Split Kernel Diagram [1]. 21
2.4 Cluster Diagram [1]. 22
2.5 Hisparse Matrix Format 23
2.6 Matrix Loader 25
2.7 Shuffle 26
2.8 Shuffle Core 27
2.9 Processing Engine Queue [1] 29

3.1 Channel Multiplexing 40
3.2 Blocking Frontier 42

4.1 Cocotb Timing Model Phases [2]. 46
4.2 Benchmark Matrix and Vector 48
4.3 Matrix Loader Waveform Diagram 49
4.4 Block Design 50
4.5 Timing Report 53
4.6 Design Floorplanning 54

List of Tables

4.1 Simulated Cycle Performance 48
4.2 Design Resource Utilization 52

Chapter 1

Introduction

1.1 Motivation

Developing with low-level Hardware Description Languages (HDLs) like SystemVerilog can be time consuming. The efficient translation of an algorithm to hardware is often verbose due to the lack of high level abstractions found in modern software development and the inherent physical constraints within the target platform. The verification of a design is arguably even more difficult. Hardware designs can face a multitude of possible inputs, many modules may be running in parallel, and low-level HDLs typically lack safeguards within the language to prevent common bugs at compilation time. Even working designs still face further issues in readability, where extensive documentation or programmer expertise is needed to understand the intent behind wire declarations and correctly interface with modules.

Efforts exist to improve this workflow and automate some of the typical tasks of a hardware engineer through the introduction of higher level languages and tools. XLS is a higher level synthesis language and library aiming to streamline this development process. It combines software style types, macros, and functional re-use to simplify bit manipulation. It generalizes the common ready-valid interface through its introduction of channels, enabling the programmer to focus on the logic consuming payloads rather than worrying about the validity of the payload itself. When generating SystemVerilog code, XLS handles breaking complicated computations across multiple stages to meet power, performance and area (PPA) requirements while preserving the natural readability of the source code.

However, there are few open source XLS projects that display its capabilities in real world

designs. A concern with higher level HDLs is that their abstractions and compilation process can result in worse performance during deployment compared to handwritten low-level implementations. Some higher level HDLs may simply fail to generate certain designs, such as latency sensitive components where behavior is dependent on the ordering of incoming data. Thus the implementation of a sufficiently large design in XLS that incorporates complicated hardware use cases would provide a useful case study in current XLS development to identify its strengths and weaknesses and give future users a basis to refer to.

1.2 Contributions

HiSparse is a modular sparse matrix-vector (SpMV) accelerator optimized for Field Programmable Gate Array (FPGA) deployment [1]. Originally developed in AMD's Vitis HLS software [3], it displays the capabilities of modern High Level Synthesis (HLS) by efficiently implementing dynamic structures often thought difficult or impossible to do without the fine grain control of traditional low-level HDLs. The implementation of HiSparse in XLS will similarly benchmark XLS' performance and identify the implications of its programming model on the design process. This work implements a variant of the HiSparse architecture [1] in XLS, providing the following contributions:

1. A detailed description of the HiSparse architecture
2. A case study on XLS development and design modifications necessary to achieve optimal performance
3. A publically available accelerator entirely written in XLS

Chapter 2

Background and Related Work

2.1 XLS Overview

XLS stands for Accelerated Hardware Synthesis, and is a development toolchain capable of generating both software and hardware designs from code written in either of its languages: XLScc and DSLX [4]. This flexibility aims to address the growing need of specialized hardware to realize performance gains, enabling users to offload software computations to hardware at their choosing. XLScc is embedded in C++ and DSLX, the language of focus for this work, is their domain specific hardware description language (HDL).

DSLX provides two functional primitives to describe computations: funcs and procs. Funcs are stateless, take in zero or more arguments, and return a value. As an expression-based language, the final expression within a func serves as its output. Funcs can call other funcs within their body.

```
1 fn mul_add(a: u32, b: u32, c: u32) -> u32 {
2     let d = a*b + c;
3     (d)
4 }
```

Listing 2.1: An example func

Procs, standing for Communicating Sequential Processes, are stateful modules capable of communicating with one another using directional channels. They are split into three regions: channel configuration, state, and the activation. The channel configuration region defines

how the proc will communicate with other procs and consists of the channel declarations as well as content within the body of the `config(){}`. The proc's state is initialized within the body of the `init{}` and is retrieved using the variable name specified in `next(){}`. In Figure 2.2, the state is initialized to the unsigned 32 bit integers 0 and 1 that can be accessed as `rolling_sums.0` and `rolling_sums.1` respectively. Finally, the activation of a proc is encoded within the body of the `next(){}` and represents the infinitely recurring logic that defines the behavior of the module.

The model of communication for procs is based upon Kahn Process Networks (KPNs) [5], a graphical model for describing the flow of computations within a distributed network of processes. Kahn Process Networks consist of processes, channels, and tokens. Processes must obtain a token before sending data to downstream processes. This token may be received from upstream channels or spawned by the process itself. As a result, tokens serve as a mechanism to order operations across processes. XLS likewise uses tokens to control when send and receive operations fire within a proc.

Kahn Process Networks, and XLS in turn, model channels as infinite depth FIFO queues with non-blocking sends (i.e. they always succeed) and blocking receives. When lowering processes to hardware, XLS translates these channels into latency insensitive connections with blocking, ready valid interfaces on both the sender and receiver. By adhering to pure KPN semantics, procs avoid timing hazard bugs, where the output of the computation is dependent on the execution time of IO operations.

However, KPN semantics alone do not support the full breadth of possible hardware designs, such as a module that must arbitrate access to a bus between multiple inputs. XLS offers non-KPN operations for these scenarios. The `recv_non_blocking` IO operation only performs a read on a channel when there is a valid message, allowing the rest of the logic within a proc's activation to continue regardless of the status of the channel. Every IO operation also has a conditional variant such as `send_if` or `recv_if_non_blocking`. These IO operations take in a boolean that controls whether the operation fires within an activation. The use of these non-KPN operations re-introduces the possibility for timing hazards.

DSLX generates hardware by lowering the encoded logic into SystemVerilog modules. The compilation process begins by transforming funcs and procs into nodes within their Sea

```
proc mul_add{
    a:      chan<u32> in;
    b:      chan<u32> in;
    c:      chan<u32> in;
    d:      chan<u32> out;
    config(
        a: chan<u32> in, b: chan<u32> in,
        c: chan<u32> in, d: chan<u32> out
    ) {(a, b, c, d)}
    init {u32: 0, u32: 1}
    next (rolling_sums: (u32, u32)){
        let (tok1, av) = recv(token(), a);
        let (tok2, bv) = recv(token(), b);
        let (tok3, cv) = recv(token(), c);
        let tok = join(tok1, tok2, tok3);
        send(tok, d, av*bv + cv);
        let new_sum_zero = rolling_sums.1;
        let new_sum_one = rolling_sums.1 + av*bv + cv;
        // multiple return values supported via tuples
        (new_sum_zero, new_sum_one);
    }
}
```

Listing 2.2: An example proc. join() produces a new token useful for ordering IO operations *after* all the tokens in its parameter list. Procs can hold state across activations.

of Nodes intermediate representation (IR). The computations within these nodes are then scheduled using a variation of a System of Difference Constraints (SDC) scheduler [6] to solve an optimization problem combining the logic's data dependencies with the hardware constraints of the target platform. The process ends with the generation of pipeline stages using an identified schedule, or returns with an error if no schedule is found. The user can further customize the compilation process by providing a custom delay model to inform the SDC scheduling process and adding constraints like target frequency or number of pipeline stages.

A further introduction to DSLX programming can be found on the XLS website.

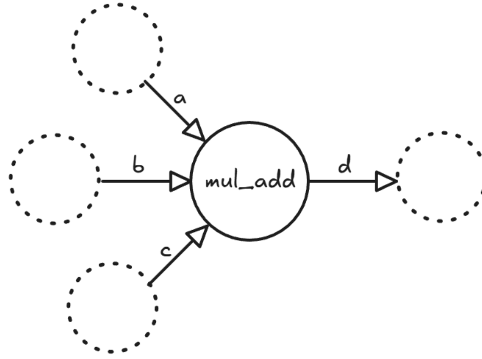


Figure 2.1: KPN depiction of Figure 2.2. Dotted circles represent other procs in the network, arrows represent channels. The mul_add proc would activate only once tokens were received from channels a, b and c.

2.2 Sparse Matrix Computations

Sparse matrix computations are a family of matrix operations where many elements of the input matrix are zeros. The operations are often denoted in shorthand with ”Sp” highlighting which matrix is sparse in the notation, thus SpMSpM and SpMV indicate a sparse matrix-matrix and sparse matrix-vector operation respectively. As the zeros of the sparse matrix can be trivially ignored when computing the result of the operation, traditional dense algorithms waste both compute resources and useful memory bandwidth when applied to sparse workloads. By instead tailoring the software and/or hardware design for these sparse workloads, performance exceeding a dense computation can be achieved.

Often the storage format of the matrix is optimized to address its sparsity. The Compressed Sparse Row (CSR) format, depicted in Figure 2.2, is a common sparse matrix storage format where a traditional two dimensional array representing the matrix is replaced with three arrays: a value array containing the non-zero matrix elements, a column array containing the column indices of the elements, and a monotonically increasing row array encoding the row indices of the elements. Unlike the column array, the values within the row array do not directly translate to the row indices of the elements. In other words, it is not the same length as the value array. Instead, the values of the row array represent the index ranges of the value array that reside on a particular row. A row array of the form [0, 2, 2, ...] implies that within the value array, elements 0 and 1 reside on the first row, no elements

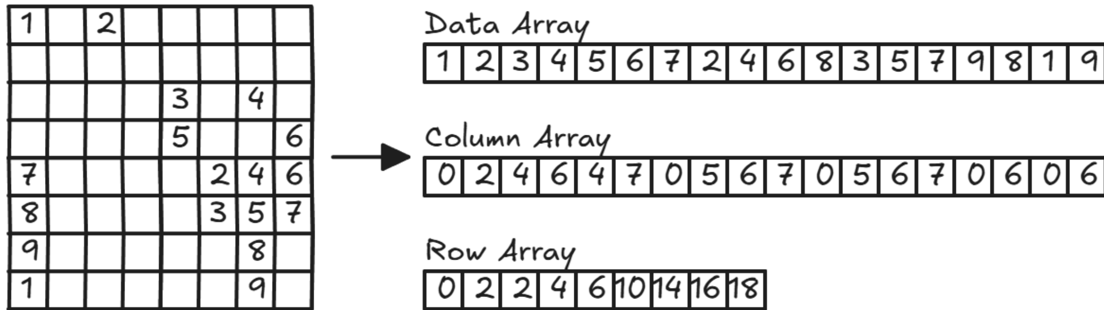


Figure 2.2: Example matrix encoded in CSR format

reside on the second row, and so on. Each successive value in the row array represents the end of the index range of the next row in the matrix. As the name of the CSR format implies, this row array is ideally compressed compared to a simpler direct encoding of the indices when the input matrix holds many empty rows. Its mirror form, Compressed Sparse Column (CSC), performs the same compression on the column indices of the matrix. Finally, simply storing the value array, and the column and row array in their non-compressed direct encoding results in the Coordinate (COO) storage format.

Many current workloads involve sparse matrix computations, and correspondingly there are many options for sparse accelerators within academic and commercial spaces; Introduced in their Ampere architecture, Nvidia's Tensor Cores support full throughput on 2:4 sparse matrices (2:4 implies that for every 4 successive matrix elements, 2 are zeros)[7]. ExTensor, a generalized sparse tensor accelerator, uses intersections to determine not only when to perform operations between scalars, but also whether to operate on larger granularities such as entire tensors [8]. HiSparse, the SpMV accelerator used as a case study for this work, implements its logic in Vitis HLS to show modern high level synthesis capabilities on the often dynamic workloads present in sparse computations [1]. A further analysis of the HiSparse architecture is provided in the following section.

2.3 HiSparse Architecture Overview

HiSparse is a modular FPGA based SpMV accelerator written entirely in Vitis HLS. Expressing dynamic operations such as payload arbitration alongside typical latency insensitive

pipeline synthesis, it broadens the idea of what is possible with high level synthesis and likewise serves as an appropriate case study for identifying the capabilities of a hardware focused language. There are slight variations between the integral and floating point version of the architecture, and this overview will focus on the integral version. However, much of the architecture remains the same for the floating point version.

2.3.1 High Level Design

HiSparse is a multi-kernel architecture capable of spanning across chip die boundaries to optimize the frequency of the synthesized design. The number of kernels is flexible and is typically sized with respect to the target matrix workload and FPGA resources. During the SpMV computation, kernels are synchronized with one another on a per row basis, while internal kernel modules can be synchronized on a finer granularity multiple times per row. Synchronization occurs through the communication of tokens that replace data values within the pipeline. This propagation of synchronization tokens across the pipeline limits the latency of the design, and the synchronization overhead is amortized through larger matrix computations. Figure 2.3 shows an example three kernel architecture. To complete a SpMV computation, the kernels must be re-triggered for each row partition of the input matrix by an external loop.

Within the kernel lies one or more clusters, serving as the smallest mostly complete computation units that compute on a row-by-row basis; Clusters are capable of requesting matrix data, caching input vector data, computing output vector updates and streaming the final row results out. The internal cluster datapath consists of multiple streams running in parallel, each stream computing the results of one or more output vector elements. Streams are numbered successively, and the stream number corresponds to the row a stream will iterate over modulo the number of streams. Thus if there are N streams, the first stream will iterate over the first row of the matrix, then it will iterate over the N+1 row of the matrix, and so on. All clusters have the same number of streams, and the streams of a cluster are grouped into structures called channels at the input of the cluster. Figure 2.4 highlights the modules each cluster contains to perform its computations.

Clusters have localized storage to retain portions of both the input vector and the final

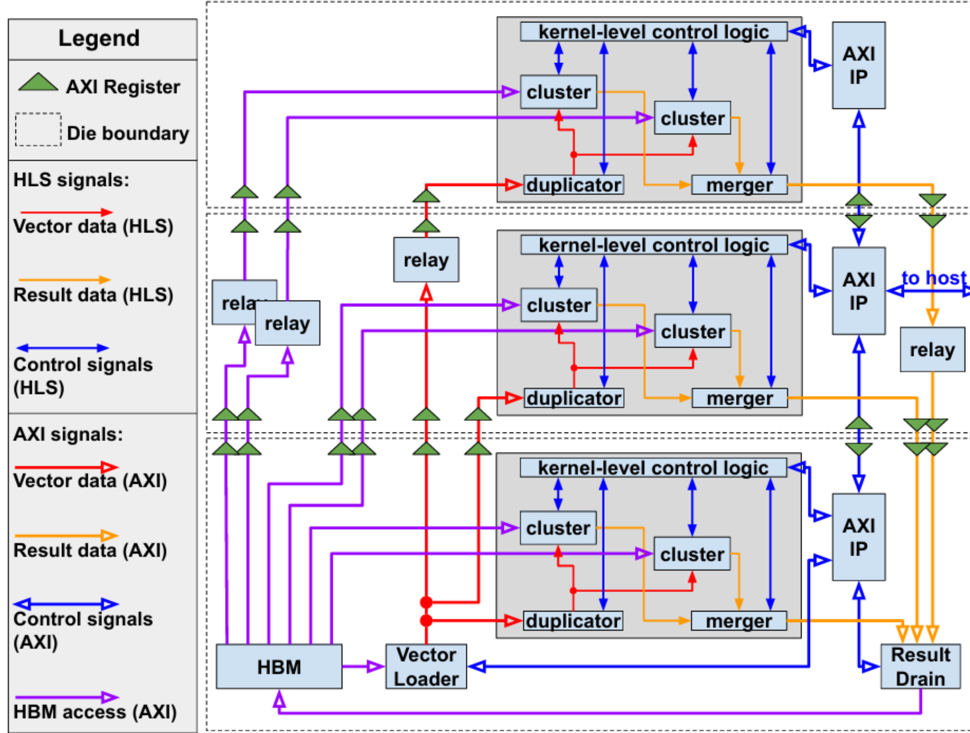


Figure 2.3: Example three kernel HiSparse architecture [1].

output vector. This storage avoids potentially expensive memory requests while processing matrix streams. The storage is split into banks, with each of the cluster's streams owning a particular bank of the cluster's input and output vector storage. All clusters require random access to the same input vector, thus the input vector is duplicated across all clusters and their stream banks. When streaming the final output vector, clusters iterate over their streams' output banks to reconstruct the result.

The size of these banks and the number of streams within a cluster limits the size of the matrix a cluster can process without needing to refresh its storage. The maximum number of columns a cluster can process is equal to the size of a stream's input vector bank multiplied by the number of streams within a cluster. In contrast, the maximum number of rows a cluster can process is equal to the size of a stream's output vector bank multiplied by both the number of streams within a cluster and the number of clusters in the entire design. This discrepancy is due to the input vector being duplicated across all clusters and their stream banks.

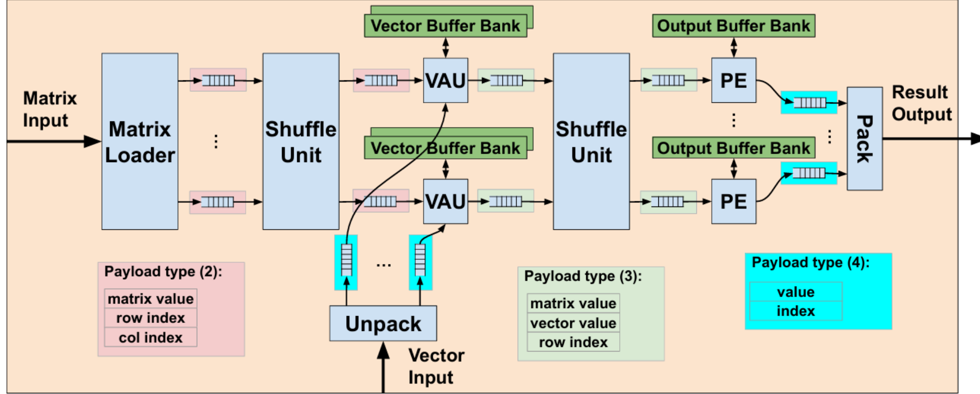


Figure 2.4: Cluster diagram [1].

2.3.2 Custom Sparse Matrix Format and Memory Layout

HiSparse introduces a modified version of the CSR sparse matrix format to address the CSR format's difficulties in streaming data. To reconstruct the row indices of a CSR encoded matrix, the row array must be traversed separately from the data and column arrays, taking note of the index ranges of specific rows. Should the matrix hold many empty rows in succession, the reading of the data and column values must be paused until the traversal of the row array has "caught up" to the next valid row. This potentially irregular access pattern hurts streaming throughput.

HiSparse replaces the functionality of the row array with tokens in the data and column index arrays to communicate row changes. A token consists of a skip row indicator (represented as a -1 in the column index array) paired with an integer in the data array representing the number of rows to skip. The skip row indicator ensures downstream modules do not treat the data and column pair as valid matrix information, and the integer enables a single token to skip an arbitrary number of rows, rather than the element by element traversal necessary in the CSR row array.

A data and column pair comprises a single payload for a single stream. As mentioned in the previous section, clusters are composed of one or more streams grouped into channels at the input. Thus, a single channel payload is composed of one or more stream payloads. These packed channel payloads represent the data type that is read from memory. Consequently, they impose restrictions on the shape of the input matrix; The number of rows of the input

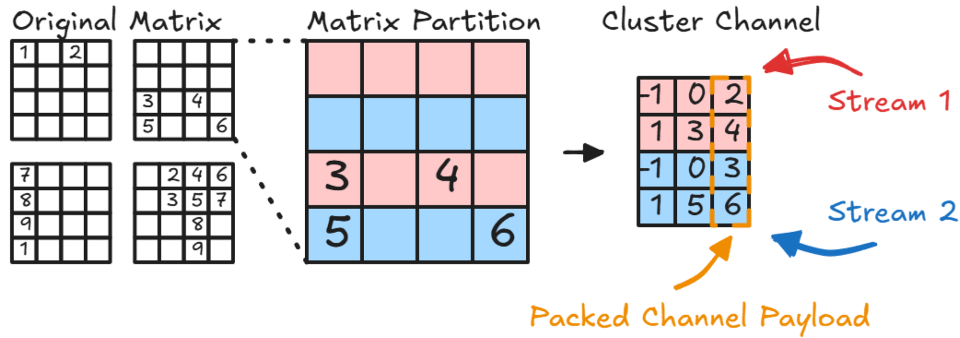


Figure 2.5: The previous CSR example expressed in HiSparse's custom format, using two streams, and vector and output banks of size 2. This results in four total partitions, two column partitions per row partition.

matrix must be cleanly divisible by the total number of streams in the entire design, and the number of columns of the input matrix must be cleanly divisible by the number of streams in a cluster. Both requirements ensure there are no memory out of bounds accesses when all of the clusters simultaneously request the packed payloads of matrix and vector data at any moment. Vector data is packed in a similar manner to matrix stream data, however, there is no use of tokens as the vector is one dimensional.

The input matrix undergoes multiple preprocessing steps to prepare it for iteration. It must be padded to ensure the divisibility requirements are met. The matrix is also split into equally sized partitions, where the dimensions of the partitions are equal to the maximal number of rows and columns a cluster can process. These partition dimensions must also meet the same divisibility requirements imposed on the input matrix. Once the properly sized and partitioned input matrix is converted into the custom format, the resulting streams are grouped into one or more channels. All streams in a channel must have the same length to again prevent malformed packed payloads, thus each stream is padded until it reaches the length of the longest stream in the channel. For each channel, per-partition metadata is stored in the beginning memory addresses preceding the stream data to inform the matrix loading modules during iteration. These channels are finally fed to the clusters for computation, each channel serving a single cluster.

2.3.3 Key Modules

Below is a description of modules performing unique functions within the HiSparse architecture. Unless specified otherwise, IO operations are performed with standard latency insensitive, ready-valid interfaces.

The matrix loader is the first module in the cluster's datapath and is tasked with iterating over an entire row of a channel's contents. To begin iteration, the row partition index, the number of column partitions per row partition, and the total number of partitions in the entire matrix is provided to the matrix loader. The matrix loader then grabs two successive metadata payloads from the channel, using the row partition index and the current column partition as an offset into the metadata region. The first payload indicates where in the channel's memory a partition's matrix data starts. The second payload describes the length of each stream in said partition.

After grabbing the metadata for a partition, the matrix loader sends a Start of Data (SOD) command payload to each of its output streams to synchronize downstream modules with the beginning of a new partition. Afterwards, the matrix loader will continuously read packed payloads from the channel, restore the row indices based upon the observed tokens and the stream numbers, and send the unpacked, fully specified matrix payloads to the correct streams in the cluster's datapath.

Once the matrix loader reads all of the channel payloads of a given partition, it will send an End of Data (EOD) command payload to its output streams. Should it have completed the final column partition of the entire row, it will also send an End of Stream (EOS) command payload. Thus, the matrix loader is also tasked with the creation of these stream command payloads for the rest of the cluster datapath to synchronize their computations with. Figure 2.6 provides an example of the matrix loader processing a partition from Figure 2.5.

Following the matrix loader is the shuffle unit. The shuffle unit is used twice within a cluster's architecture, as noted by Figure 2.4. Its job is twofold:

- To resolve conflicts between streams, where conflicts are determined by shared indices across different stream payloads.

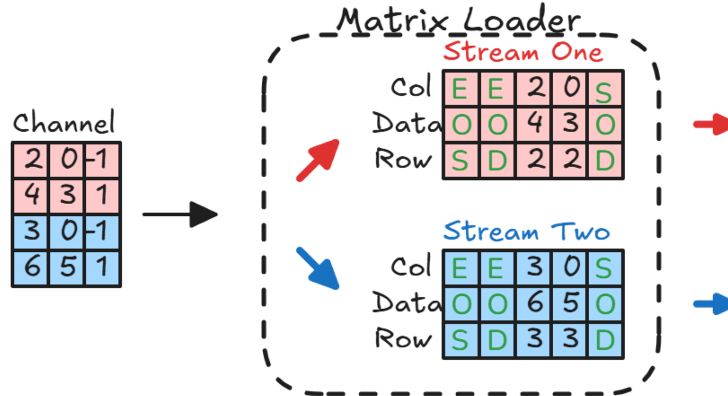


Figure 2.6: Matrix loader output after processing the final column partition of the first row in Figure 2.5. Note the creation of the SOD, EOD and EOS stream commands.

- To synchronize streams commands with one another.

For the first shuffle unit, conflicts occur when two or more streams share the same column index within a given packed payload, as these streams would conflict on the same input vector bank. For the second shuffle unit, conflicts occur when two or more streams share the same row index, as they would again conflict on the same output vector bank. The shuffle unit internally consists of the toplevel shuffle module, a shuffle core module, an arbiter, and a crossbar module.

At the very top, the shuffle module serves as a synchronization wrapper for the rest of the shuffle unit. It first synchronizes all streams with respect to the SOD payload, consuming the command and pausing streams until all are observed to have sent SOD. The shuffle module then sends a SOD command across all streams on its output. Afterwards, the shuffle module diverts all stream inputs and outputs to the shuffle core, waiting for the core to finish arbitrating the streams' contents before proceeding. Once the core exits, the shuffle module will again repeat this synchronization and shuffle core diversion process until it observes the EOS command across all input streams. After observing the EOS command, it will again send the command payload across all streams of its output and finish its computation. Non blocking reads are used during the synchronization process to ensure no stream's payload is dropped when reading another stream's payload. Figure 2.7 depicts the functionality of the shuffle module as a state machine diagram.

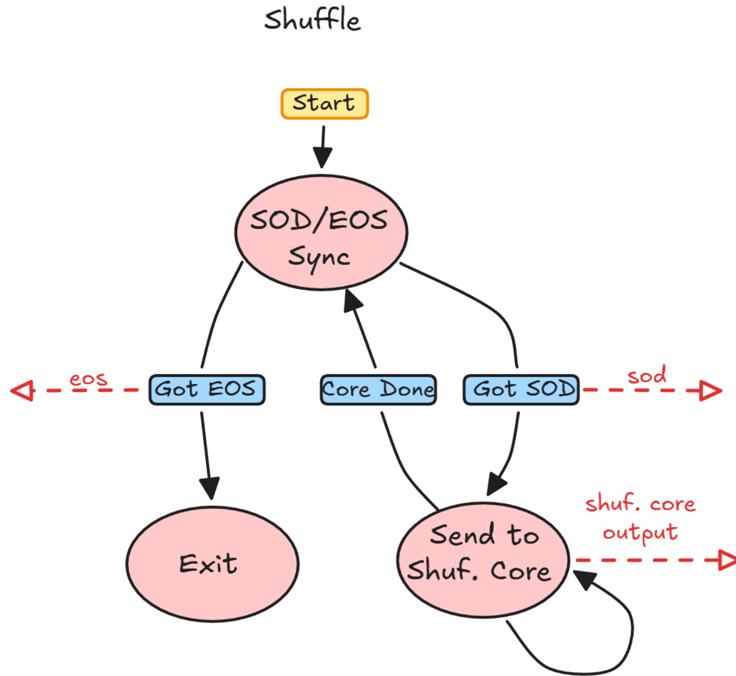


Figure 2.7: Shuffle module state machine diagram. Writes to the output streams are indicated in red.

The shuffle core module performs the stream pausing and payload resending functions required to resolve conflicts identified by the arbiter. It wraps the arbiter and crossbar modules, feeding the arbiter from its input streams and driving its output streams with the crossbar. For each stream, the shuffle core pipeline begins by sampling the arbiter's outputs to determine if it should resend a payload. If it should, it avoids reading from the input stream and instead sends the arbiter payload back into the arbiter's inputs. Otherwise, it performs a non-blocking read on the input stream and sends the payload directly into the arbiter. If the shuffle core resends a payload for one or more streams, the ready signal for the corresponding input streams may be deasserted and can stall the matrix loader and overall pipeline from the resulting backpressure. Consequently, non-blocking reads are used to ensure progress in such a scenario. When the arbiter grants a payload to an output stream, the shuffle core will route the payload through the crossbar, which holds an all to all connection to the output streams of the shuffle core. The shuffle core will stop reading from an input stream once EOD is observed, sending zeroed payloads instead into the arbiter.

Once all input streams have sent EOD, the shuffle core will run for a set number of

flushing cycles to ensure all of the valid payloads within the arbiter have been sent to the output streams. If right before the EOD payload, a payload with every stream conflicting on the same index was sent into the arbiter, then a full "flush" of the arbiter's stages must be done every time a stream payload was resent. Thus, the number of flushing cycles is determined by multiplying the number of pipeline stages of the arbiter with the number of streams in a cluster. Figure 2.8 depicts the functionality of the shuffle core module as a state machine diagram.

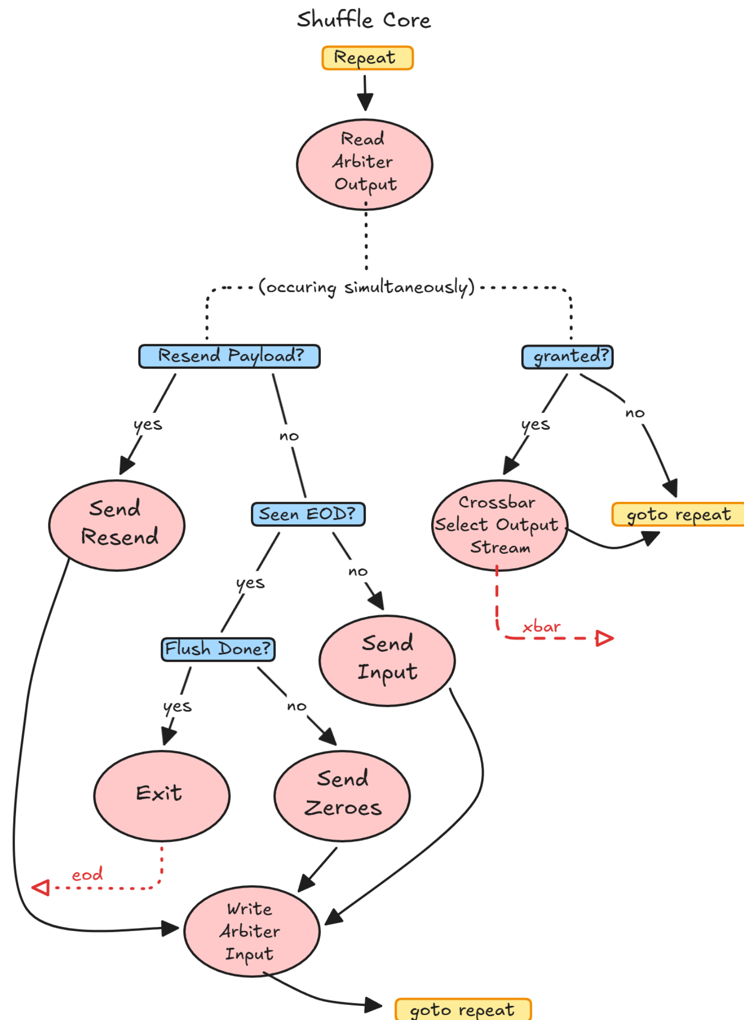


Figure 2.8: Shuffle core module state machine diagram. Writes to the output streams are indicated in red.

The remaining module of note in the shuffle unit is the arbiter. The arbiter is a stateless function that is reinvoked by the shuffle core. It loops through the input streams, identifies

the index conflicts defined previously, and chooses which streams are able to route to the output through a priority queue. The arbiter also intakes an offset to control which stream has the top priority for fairness. On its output, it provides the routing information for the crossbar to ensure the correct bank is used downstream for each input payload. The arbiter performs multiple looping calculations and comparisons over the input streams, thus its logic is broken into multiple stages when generated into verilog to improve the frequency of the overall design.

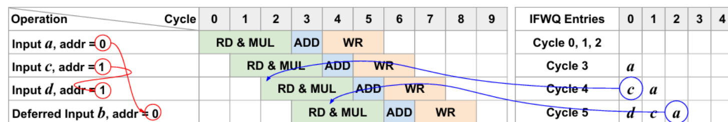
The transmission of input vector data is handled by multiple modules within the HiSparse design. As mentioned previously, vector data is duplicated across all clusters within a kernel. The vector loader handles this duplication by sending partitions of the input vector that correspond with the matrix partitions. Simiar to the matrix loader, it pairs these vector partitions with SOD, EOD and EOS command payloads.

On the receiving end of the vector loader's stream and the first shuffle unit's output is the vector buffer access unit. There is one vector buffer access unit per stream in a cluster. It is tasked with both managing the input vector bank for its corresponding stream, and replacing the column index in incoming COO formatted payloads with the associated vector value. At the beginning of every matrix partition calculation, the vector buffer access unit must replace the stale bank data with the correct information provided by the vector loader stream *before* handling incoming shuffle unit payloads. As partitions could be entirely empty, this repeated bank filling and flushing may add to the latency of a cluster's calculation. Thus, the banks in each vector buffer access unit are double buffered to help hide the latency of this bank maintenance.

The final module critical to a cluster's SpMV calculation is the processing engine. It holds many similarities to the vector buffer access unit, handling a single stream within a cluster and maintaining an output vector bank during the computation. The processing engine receives payloads from the second shuffle unit that contain the row, matrix, and vector value, and performs the multiply accumulate operation for the entire row partition. During its accumulation, the processing engine must grab the up to date output vector value from its bank. However, since latencies of FPGA memories are often one or more cycles, this up to date value may still be in flight should the cluster's pipeline be operating at a sufficiently

low initiation interval. This problem is a type of Read After Write (RAW) hazard, and is mitigated by the processing engine maintaining an in-flight write queue (IFWQ). Figure 2.9 shows this queue in action. As depicted, the queue maintains the full throughput of the pipeline while resolving the accumulation's data dependencies. The depth of the queue is the combined read and write latency of the memory technology backing the output vector bank. Each cycle new data is inserted into the queue using non-blocking reads done on the input stream to ensure the queue's data is cycle accurate with respect to the current in flight writes.

(a) Using registers.



(b) Using load-store forwarding — Red arrows indicate the RAW dependencies. Blue arrows indicate the data forwarding to resolve dependencies.

Figure 2.9: RAW hazard mitigation enabled by the processing engine's in-flight write queue [1].

Upon getting the EOS command payload, the processing engine streams the result out to further cluster and kernel result collection modules. This concludes the SpMV operation within the HiSparse architecture.

Chapter 3

Implementation

Implementing a correct and streaming optimized HiSparse variant in XLS involved multiple approaches due to existing bugs in the compiling toolchain and code generation peculiarities that limited throughput in certain scenarios. Split into three attempts, the first encapsulates a ”naive” approach that purely translates HiSparse's logic into XLS and fails to generate verilog. The second attempt addresses the code generation failures of the first through a conservative design framework. The third and final attempt modifies the HiSparse architecture to reduce the throughput penalties of the conservative framework, leading to a full streaming throughput design.

3.1 The Naive Implementation

The first attempt involved identifying the XLS features necessary to translate HiSparse's logic. At a high level, procs were used in almost all module implementations due to the prevalence of latency insensitive AXI4-Stream connections in the original Vitis HLS implementation. These stream connections can be directly expressed through XLS' streaming channels. Procs also easily supported a state machine structure within their activation to reason about the initialization, streaming, and cleanup behavior of each module, grouping operations together under a single thematic state. Figure 3.1 shows an abridged version of the first matrix loader implementation.

Only a single match arm within the state machine is firing in a given activation, thus

each state is self contained. Regarding the IO operations themselves, both regular sends and receives as well as conditional and nonblocking variants were used depending on the original modules behavior. Conditional sends and receives also reduced the number of states necessary by providing greater control over the active codepaths within a state. To ensure IO operations were ordered across multiple activations, cross-activation tokens[4] (tokens that are included within a proc's state) were used as a basis for all IO operations.

```
pub proc matrix_loader<STREAMS: u32>{
    addr:                       chan<u32> out;
    payload:                    chan<uN[64][STREAMS]> in;
    output:                     chan<uN[64]>[STREAMS] out;

    // ...

    next (state: ...) {
        let new_state =
        match (state.0) {
            u32: 0 => { // inital state, read partition metadata
                let new_tok = send(state.10, addr, state.1);
                let (new_tok, metadata_one) = recv(state.10, payload);
                let new_tok, send(state.10, addr, state.1 + u32: 1);
                let (new_tok, metadata_two) = recv(state.10, payload);
                // update state...
            },
            u32: 1 => { // streaming state, read custom matrix format
  and output COO matrix packets
                let new_tok = send(state.10, addr, state.1);
                let (new_tok, custom_matrix_format_payload) =
  recv(state.10, payload);
                // process custom matrix format...
                for (stream_idx) : (u32) in u32:0..STREAMS {
                    let new_tok = send(state.10, output[stream_idx],
  COO_output_payload);
                } // ...
                // update state...
            },
            u32: 2 => { // send EOS, finish
                let new_tok = send(state.10, payload, EOS);
                // update state...
            }
        };
        (new_state)
    }
}
```

Listing 3.1: Initial design of the matrix loader. The initial state, streaming state, and cleanup state reside in separate match arms

Each module was tested both individually for correctness and in larger scale integration tests with a sample sparse matrix. These tests were performed using XLS' test proc framework, a special type of proc capable of being invoked at a software level, i.e., without the generation and simulation of verilog. The final, end-to-end test involved a single kernel,

single cluster, two stream architecture computing the correct SpMV result between an 8x8 matrix and an 8 element vector. This architecture and computation will serve as the base for all future implementation attempts.

After achieving the correct result in the end-to-end test, the generation of verilog (referred to as codegen) and verification of its correctness followed. However, this first XLS implementation failed to generate code due to multiple compiler errors present at the time of the development of the HiSparse XLS variant. The first limiting bug involved the flowing of tokens through match arms as indicated in Figure 3.2.

```
pub proc example {
    addr:       chan<u32> out;
    pld:        chan<u32> in;
    config(addr: chan<u32> out, pld: chan<u32> in) { (addr, pld) }
    init{(u32: 0, token())}
    next (state: (u32, token)) {
        match (state.0) {
          u32:0 => {
            let new_tok = send(state.1, addr, u32: 0);
            let (new_tok, _) = recv(new_tok, pld);
            (u32: 1, new_tok)
          },
          u32:1 => {
            let new_tok = send(state.1, addr, u32: 0);
            let (new_tok, _) = recv(new_tok, pld);
            (u32: 2, new_tok)
          },
          _ => {
            state
          },
        }
    }
}
```

Listing 3.2: Example proc that failed to codegen due to tokens flowing through match arm results.

XLS is an expression oriented language with block style syntax, where values are returned at the ends of the blocks. Each match arm in Figure 3.2 is a block delineated by the curly brackets, and the return values of the block include a token due to the IO operations present. As mentioned previously, this token must be preserved to update the cross-activation token held in the proc's state to ensure IO operations are ordered across activations. Thus, IO

operations had to be separated from the block representing the state to avoid flowing the tokens through match arms and preserve the tokens.

The second limiting bug involved multiple IO operations per channel per activation. An activation is a single iteration of the logic contained in the `next(){}` structure of a proc. If the logic involves multiple IO operations on a single channel (for example, multiple sends on a `addr: chan<u32> out;` channel), a fifo must be described backing the channel in question. This fifo could be used when multiple IO operations contend for the same channel when activations are overlapped with one another, or when downstream connections exert backpressure on a streaming channel and pending operations must be queued until the downstream connection is ready again. However, the mechanism to describe FIFOs for these channels did not work, and the compiler continuously errored due to the missing fifo instantiation. Figure 3.3 provides an example proc incapable of codegen due to this bug.

```
1  pub proc example {
2      addr:        chan<u32> out;
3      pld:         chan<u32> in;
4      config(addr: chan<u32> out, pld: chan<u32> in) { (addr, pld) }
5      init{(u32: 0, token())}
6      next (state: (u32, token)) {
7        let new_state =
8        match (state.0) {
9          u32: 0 => { u32: 1 },
10         u32: 1 => { u32: 2},
11         _ => {state.0}
12       };
13       let new_tok = send_if(state.1, addr, state.0 == u32: 0, u32: 0);
14       let (new_tok, _) = recv_if(new_tok, pld, state.0 == u32: 0, u32:
   0);
15       let new_tok = send_if(new_tok, addr, state.0 == u32: 1, u32: 0);
16       let (new_tok, _) = recv_if(new_tok, pld, state.0 == u32: 1, u32:
   0);
17       (new_state, new_tok)
18     }
19 }
```

Listing 3.3: Example proc that failed to codegen due to multiple operations per channel per activation. Note this proc is free of the previous tokens flowing through match arms bug.

As a consequence of this bug, state machines had to be modified to ensure only one IO operation on a channel was performed in the entire activation.

The final bug affecting the development of the HiSparse variant was the inability to codegen parametric proc networks. Procs can instantiate other procs when being configured, and this quality was particularly useful given the nested, modular structure of the HiSparse architecture where kernels could contain multiple clusters, clusters contained multiple modules, and so on. However, this feature failed to work when cross activation tokens were present in the procs being instantiated, which encompassed practically all modules within the design. Though not changing the design of the HiSparse XLS variant, this bug greatly slowed down development as codegen'd XLS procs had to be manually connected through hand-written verilog toplevel modules and other similar plumbing.

3.2 The Codegen Implementation

Taking all of the previous limitations into account, a framework was developed to avoid the aforementioned bugs and guarantee codegen capabilities. Figure 3.4 shows the previous abridged matrix loader modified to conform to this new framework.

```
pub proc matrix_loader<STREAMS: u32>{
    // ...
    next (state: ...) {
        // Stage 1: boolean generation
        let (tx_addr, tx_out, rx_pld) =
        match (state.0) {
            u32: 0 => {(true, false, false)},
            u32: 1 => {(false, false, true)},
            u32: 2 => {(false, true, false)},
            _ => {(false, false, false)}
        }
        // Stage 2: Send payload generation
        let (addr_pld, out_pld) =
        match (state.0) {
            u32: 0 => {(state.1, zero!<uN[STREAMS][96]>())},
            u32: 1 => {(u32: 0, state.2)},
            _ => {u32: 0, zero!<uN[STREAMS][96]>()}
        }
        // Stage 3: IO firing
        let t1 = send_if(state.2, addr, tx_addr, addr_pld);
        let t2 = send_if(state.2, output, tx_out, out_pld);
        let t3 = recv_if(state.2, payload, rx_pld,
   zero!<uN[STREAMS][64]>());
        let new_tok = join(t1, t2, t3);
        // Stage 4: State update
        let new_state =
        match (state.0) {
            // state transition logic...
        };
    }
}
```

Listing 3.4: First codegen design of the matrix loader. The stages of a substate are indicated in comments.

In the conservative framework, all proc state machines were refactored to ensure at most one direction of IO operation was performed per activation. Thus if a state within a proc's state machine originally performed (in order) a send, a receive, and two subsequent sends, these IO operations were later grouped into three smaller ”sub-states” of send, receive, and the final two sends. If IO operations shared the same direction and original token, i.e., they were meant to occur simultaneously, then these IO operations resided in the same sub-state. Alternating send and receive patterns multiplied the number of states within a module, and in Figure 3.4, the initialization state was broken into 4 sub-states as a result

of this transformation. Using this approach allowed for a single conditional IO operation per channel that only needed to be ordered with respect to the cross-activation token in the proc's state.

The rest of the framework works around this restriction to progress through the original state machine logic. The logic of a sub-state is spread apart into four sequential stages (also numbered in Figure 3.4):

1. The generation of the booleans to trigger the conditional IO operations
2. The generation of send payloads
3. The firing of the conditional IO operations
4. The determination of future state based upon current state and received payloads

With this conservative framework, the first successful codegen of the HiSparse XLS variant was achieved. Though some modifications had to be made to the state variables stored in each proc, particularly to carry over received values from previous sub-states to future sub-states, the implementation of this framework across all HiSparse XLS modules was straightforward.

Apart from the conservative framework, skid buffers also had to be attached to the arbiter due to a combinational loop across the ready signals of the shuffle unit. The shuffle core both drives the arbiter inputs and samples the arbiter outputs. As a result, its determination of its ready signal naturally created a combinational loop when connected to the arbiter. XLS' codegen flow provides the option of attaching skid buffers on the inputs and outputs of a module, which solved this problem.

However, due to the multiplication of states as a result of the IO operation restriction, the throughput of the overall design suffered greatly. The initiation interval (II) of a proc, specified during the codegen process, refers to the number of cycles that can elapse between two proc activations ignoring stalls [4]. If each state could execute in its entirety within a single cycle, full throughput would be achieved when performing codegen with an II of 1. However, many states exercised the same fundamental execution path of sending an address, receiving a payload, performing a computation on the received data, and sending

the computed result downstream. The IO operation restriction ensured this execution path must be split into three separate sub-states, consequently requiring at least three cycles to complete the original state and accept a new value. Since cross-activation tokens were used to serialize IO operations across activations, this three cycle latency cannot be hidden by overlapping multiple activations with one another.

Thus modules like the conservative matrix loader in Figure 3.4 were throughput limited to one value per three cycles, whereas other modules had further limitations. For example, the shuffler core must receive data from the arbiter, *conditionally* receive data from the input stream (determined by the previous arbiter payload), send data into the arbiter, and finally send data to the output stream, leading to a minimal latency of four cycles for each new value. Given the shuffle module wraps the shuffle core's inputs and outputs, the entire shuffle units performance was further degraded for the same reasons.

Working around these performance limitations, a more optimized framework was developed with consultation from the language creators to achieve full streaming throughput.

3.3 The Optimized Implementation

When optimizing the codegen implementation, the goal was to achieve true full throughput during streaming by ensuring a new value was accepted on each cycle for every module when possible. Most HiSparse modules contained an initialization phase, streaming phase, and cleanup phase for each column partition. Since the streaming phase occupied the vast majority of the modules runtime, its optimization would result in the greatest performance gains and subsequently was the priority. Likewise, any techniques developed to optimize the streaming phase are likely generally applicable and can inform how to optimize the other parts of the design.

Two main techniques, Channel Multiplexing and Frontier Splitting, were used to achieve full throughput during steady state streaming.

3.3.1 Channel Multiplexing

As mentioned previously, part of the impetus for the conservative framework was to avoid the codegen failures present when procs encoded multiple IO operations on a particular channel within a single activation. The vast majority of the non codegenable scenarios involved conflicts between the initialization, streaming, and cleanup phases for a module. These scenarios were split into separate stages within a state machine and consequently conflicted on the IO channels, failing to codegen due to the inability to instantiate a fifo.

However, despite the logic in a proc's `next(){}` structure encoding multiple IO operations per channel, the IO operations resided in separate stages of the state machine and at any given activation only a single IO operation could occur on the channel. Using this fact, multiple channels were created that inherently referred to the same channel, and a multiplexing proc combined these channel references into a unified channel for both sends and receives. It must be noted that this workaround is very similar to a future update provided in the XLS compilation flow that was not available at the time of development [4].

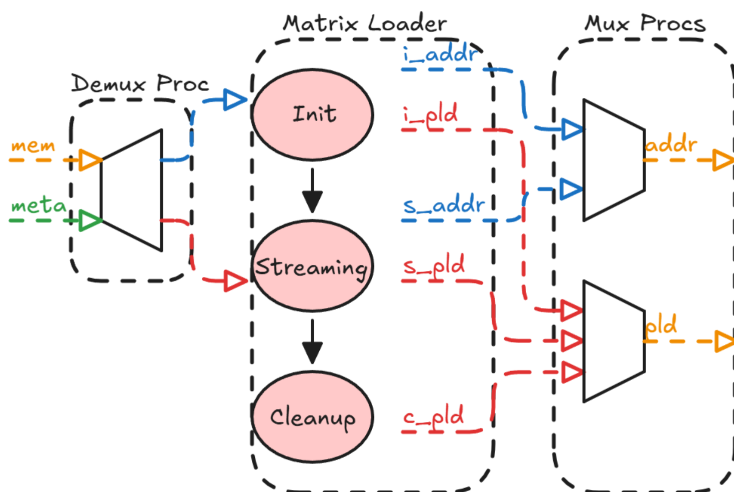


Figure 3.1: Example of channel multiplexing on the matrix loader.

Figure 3.1 displays this approach on the matrix loader. It reads metadata at the beginning of a column partition, streams the partition's contents, and sends EOD and EOS payloads at the end of the column partition. To route the incoming memory payloads to the correct channels on the matrix loader, metadata indicating which address channel sent the

request is pipelined alongside the address. This technique was used for the matrix loader, vector buffer access unit, and processing engine to reduce the complexity of the state machines and in turn lower the activations necessary during their streaming phases.

3.3.2 Frontier Splitting

Apart from the previously mentioned compiler bugs leading to decreased throughput, the current XLS compilation flow introduces pipeline stages for each send->receive execution path identified in a proc's activation. Should there be multiple send and receive operations in succession, such as during the initial phase within Figure 2.6, each send->receive pair will introduce a cycle of latency that again cannot be overlapped with other operations due to the cross-activation token serializing IO operations across activations. This send->receive execution path will be referred to as a blocking frontier, and the technique for optimizing this path will be referred to as frontier splitting.

Frontier splitting involves creating two procs for each identified blocking frontier. Figure 3.2 displays this technique for the channel multiplexed matrix loader. During streaming, the matrix loader will send a memory request, receive a payload, and send a modified payload downstream. Though the channel multiplexing technique ensures this path executes in a single activation (as the initialization, streaming and cleanup channel references are separate), a pipeline stage will be inserted during codegen between the sending of the memory request and receiving of the payload. Using frontier splitting, this path is split into two procs, one responsible for the creation of the memory requests, labelled ml_send, and another responsible for the handling of received payloads and sending of computed results downstream, labelled ml_recv.

Splitting originally monolithic procs into independent send and receive procs introduces added complexity. If state information managed by the send proc is needed by the receive proc, this state information must be pipelined alongside the send proc's requests. For the matrix loader, this involved attaching metadata to memory requests that inform the receive proc on when to send SOD, EOD and EOS command payloads. However, communication from the receive proc to the send proc is more difficult as it moves against the primary direction of the architecture. For timing sensitive state updates, it may not be possible to

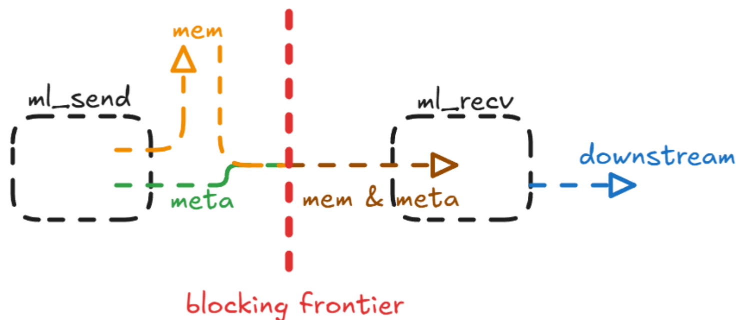


Figure 3.2: Diagram of streaming optimized matrix loader using frontier splitting technique. ml_send pipelines metadata alongside memory requests, and ml_recv computes updates based upon the combined metadata and memory data.

efficiently propagate information upwards. For example, the processing engine's streaming state consists of receiving incoming matrix and vector information, sending a bank read request, receiving the memory payload and sending the accumulated result downstream. Each blocking frontier was wrapped by send and receive procs, referred to as `pe_send` and `pe_recv` respectively. This enabled full streaming throughput. However, if the received memory payload influenced the sending of future read requests, the required receive from the `pe_send` proc would introduce another blocking frontier and lower the throughput of the `pe_send` proc.

Luckily, only the shuffle module held a dependency of this form. The shuffle module sampled the shuffle core's outputs to determine when the current column partition had finished. Once the partition completed, the shuffle module would simply read the next incoming matrix payload to figure out if the SOD or EOS command payload should be sent. This behavior was replaced through a combination of modifying the shuffle core and arbiter behavior to support passthrough of command payloads, and the introduction of stream syncing procs to replace the original syncing behavior of the shuffle module. Through these changes, the shuffle module could be removed and this cross-proc data dependency avoided, enabling full streaming throughput for the shuffle module.

Using the above techniques, the following architecture changes were made to the HiSparse architecture:

1. **Matrix Loader**

   (a) Split into two procs: ml_send and ml_recv. ml_send sent address requests to external memory, ml_recv processed the returned memory payload.

   (b) Metadata and streaming versions of the addr and payload channels were created, with associated mux and demux procs.

2. **Shuffle Unit**

   (a) Shuffle unit reduced to shuffler core and arbiter (from original shuffle outer module, shuffle core, and arbiter)

   (b) Arbiter was modified to pass through commands to enable deletion of the shuffle outer module.

   (c) Sync proc introduced to sync command tokens in stream, separating that behavior from the shuffle unit.

3. **Vector Buffer Access Unit**

   (a) Split original vector buffer access unit into two procs: vba_send and vba_recv. vba_send sent input vector address requests and vba_recv paired the returned value with the original payload.

   (b) Loading and streaming versions of the addr and payload channels were created, with associated mux and demux procs.

4. **Processing Engine**

   (a) Split original processing engine into two procs: pe_send and pe_recv. pe_send requested the value of the output vector buffer that corresponds to a given input payload, and pe_recv performs the accumulation and writeback into the output vector buffer.

   (b) Clearing, streaming and result reading versions of the addr and payload channels were created, with associated mux and demux procs.

Chapter 4

Evaluation

The evaluation of the HiSparse XLS Implementation was done in multiple phases. As mentioned previously, some testing for correctness was first done before codegen through the use of XLS' test proc infrastructure. After codegen was achieved, simulation of the generated verilog was done with handwritten top-level files and driven by Cocotb [2] coroutines with Verilator [9] as the backend simulator. Surfer [10], an open-source waveform viewer, was used to visually analyze the performance of the design. The Cocotb-Verilator-Surfer evaluation flow was used from the first codegen achieved until full throughput streaming was realized. Finally, a small verilog driver was written alongside BRAM wrappers for memory requests to prepare the design for deployment onto an FPGA. For all evaluation phases, the single kernel, single cluster, two stream HiSparse architecture was used with input vector and output vector bank sizes of 2.

4.1 Simulation Setup

Input matrices, described in JSON files with metadata, were read by custom python classes to create iterables supporting HiSparse's memory layout and custom sparse matrix format. Similar iterable creation was done on the input vectors to support the packed data format. Cocotb drivers, i.e. python coroutines monitoring verilog wires for a particular module, were created with customizable latency for the matrix loader, vector loader, vector buffer access units, and processing engines to simulate a memory servicing their requests. The backing

store for the drivers could be the aforementioned matrix and vector iterables for the matrix loader and vector loader drivers, or simple empty arrays to reflect the banks of the vector buffer access units and processing engines.

When the previously monolithic proc implementations were split across blocking frontiers into send and receive procs, any ready backpressure from the receive proc must be routed upwards towards the send proc to ensure the send proc pauses iteration. Combinational ready connections were simulated within the Cocotb drivers to simulate this backpressure, which also required the ordering of reads and writes within the Cocotb driver code to ensure correctness with regards to Cocotb's timing model [2]. Figure 4.1 displays the phases of a given Cocotb simulation time step in the timing model. To ensure a write observed up to date data, an arbitrary number of HDL Evaluation -> Values Changed loops were waited upon before sampling wire values, while other writes and reads executed after a lesser number of iterations of said loop. This difference in loop iterations established an order between wire reads and writes and ensured wire connections settled into correct values.

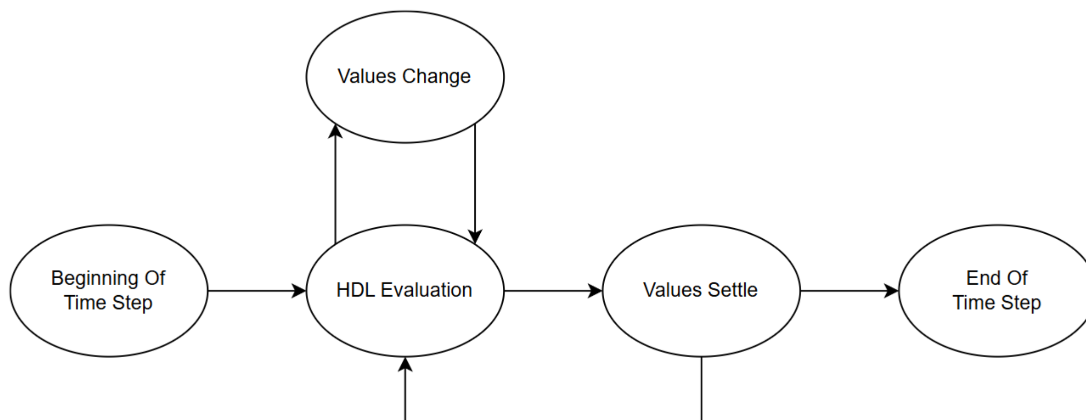


Figure 4.1: Phases of a time step in Cocotb's Timing Model [2].

Whenever a receive proc exerts backpressure on upstream send procs, multiple in-flight memory requests, the number determined by the latency of the simulated memory, must be saved to ensure values are not dropped once the receive proc is ready again. This backpressure was managed using coroutines alongside Cocotb's event triggers. Each transaction fired its own coroutine that only progressed when downstream modules were ready and ensured

results returned in order.

Finally, to reduce the number of manual inputs necessary to codegen each XLS module, a Makefile was developed that generated the modules of the XLS HiSparse implementation with preset codegen flags.

4.2 Simulation Results

Table 4.1 displays the number of cycles used to calculate the SpMV result on the benchmark matrix and vector displayed in Figure 4.2. The first version, the Unoptimized Conservative, took 2058 cycles to compute the SpMV result. Through minor codegen tweaks such as generating with a target II of 1, its optimized variant achieved 897 cycles. This was the upper bound of the conservative framework, and the following versions required the architectural changes mentioned previously to achieve better performance. The Optimized Matrix Loader version split the matrix loader into send and receive procs, however, it resulted in no speed increase from the Optimized Conservative. Through analyzing the waveform generated during the cycle simulation, the unoptimized shuffle unit bottlenecked the performance of the design in both scenarios. Correspondingly, once the shuffle unit was optimized, a 70% speed increase was observed. The shuffle unit is used twice within a cluster to manage both input vector and output vector bank conflicts. It also had the most complicated datapath out of all modules, requiring many activations for each new input value due to the multiple states necessary for communication between the shuffle module, shuffle core and arbiter. Thus, much of this speed increase can be attributed to the optimized shuffle unit.

In between the two shuffle units is the vector buffer access unit, whose optimization saved 136 cycles. The vector buffer access unit only had a single blocking frontier and was a relatively simple state machine originally, explaining the modest speed increase. Finally, the optimization of the processing engine resulted in the final 155 cycles saved and represents the full streaming throughput design. Like the shuffle unit, the processing engine required multiple activations for each incoming payload due to the multiple states necessary for its accumulation. Thus its streaming optimized variant saved a few more cycles than the vector buffer access unit.

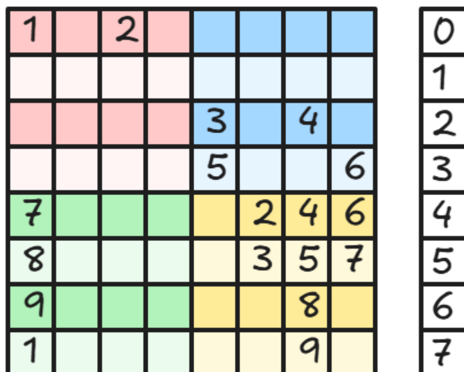


Figure 4.2: Matrix and vector used to benchmark each codegen capable HiSparse version. The four column partitions are highlighted in different colors, and each stream is its own shade of the partition's color.

Version	Cycle Count	Percent Speedup from Previous
Unoptimized Conservative	2058	n/a
Optimized Conservative	897	129%
Optimized Matrix Loader	897	0%
Optimized Shuffle Unit	528	70%
Optimized Vector Buffer Access Unit	392	35%
Optimized Processing Engine	237	65%

Table 4.1: Simulated cycle performance of each codgen capable HiSparse XLS design. Versions are introduced in order of development and include the optimizations of the previous versions if applicable.

For a M by N matrix, there are M multiplies and $M-1$ additions for each output vector element calculated in a dense matrix-vector multiplication, thus the equation for the number of operations is $N*(M+(M-1)) = N*(2*M-1)$. Using this formula, a simple implementation that performed a dense matrix-vector multiplication on the benchmark matrix and took one cycle per operation would take 120 cycles. Thus despite the HiSparse design achieving full throughput during streaming, the unoptimized initialization and cleanup-phases as well as general overhead of the HiSparse modules contributed enough latency for the SpMV calculation to exceed the cycle count of the simple, dense-assuming implementation. To realize the efficiency gains of the SpMV architecture, a larger 32x32 matrix that was 99% sparse was ran on the same single kernel, single cluster two stream architecture and achieved a runtime of 727 cycles. This is 177% faster than the simple, dense-assuming runtime of 2016

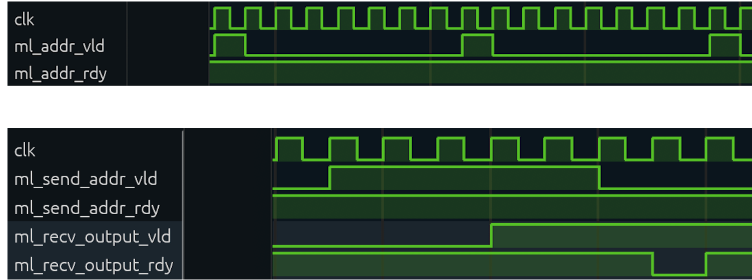


Figure 4.3: Comparison of the streaming performance between the monolithic and split-proc versions of the matrix loader. At the top, each valid address is separated by multiple unused cycles due to the conservative state machine as well as the generated pipeline stages from the blocking frontier. At the bottom, multiple addresses are communicated without empty cycles in between, highlighting the full throughput streaming performance of the optimized matrix loader.

cycles using the above formula, displaying the greater efficiency of the HiSparse design.

4.3 Hardware Setup

An AMD Pynq-Z2 served as the target board and its Zynq-7000 SoC the target chip for synthesizing the design. Vivado 2024.2 [11] was used to generate the bitstream and hardware overlay files to initialize the design in the Processing System's (PS) Jupyter Notebook environment provided within the Pynq ecosystem. A wavlink AC1200 router was used to set up a LAN network to communicate between the Pynq and host machine.

Some verilog was written outside of the XLS environment to prepare the design for synthesis. A verilog testbench was created to drive the cluster in a similar manner as the Cocotb simulation and record the results and cycle latency of the SpMV calculation. Likewise, all memory requests were converted to support the Block RAM (BRAM) interface [12] using intermediate wrappers. As metadata must be pipelined alongside memory requests due to the frontier splitting optimization technique, these wrappers included skid buffers to both ensure values were not dropped whenever downstream modules exhibited backpressure, and that values were always associated with their correct metadata. Xilinx Coefficient files (COE) [13] were used to prepopulate the BRAMs with the input matrix and vector data using HiSparse's sparse format.

To ensure the design was running properly, AXI GPIO IP blocks [14] were included in

the block diagram and used to sample the values of the output vector and cycle count wire of the design. These wires were read on the PS side of the Zynq using python function calls.

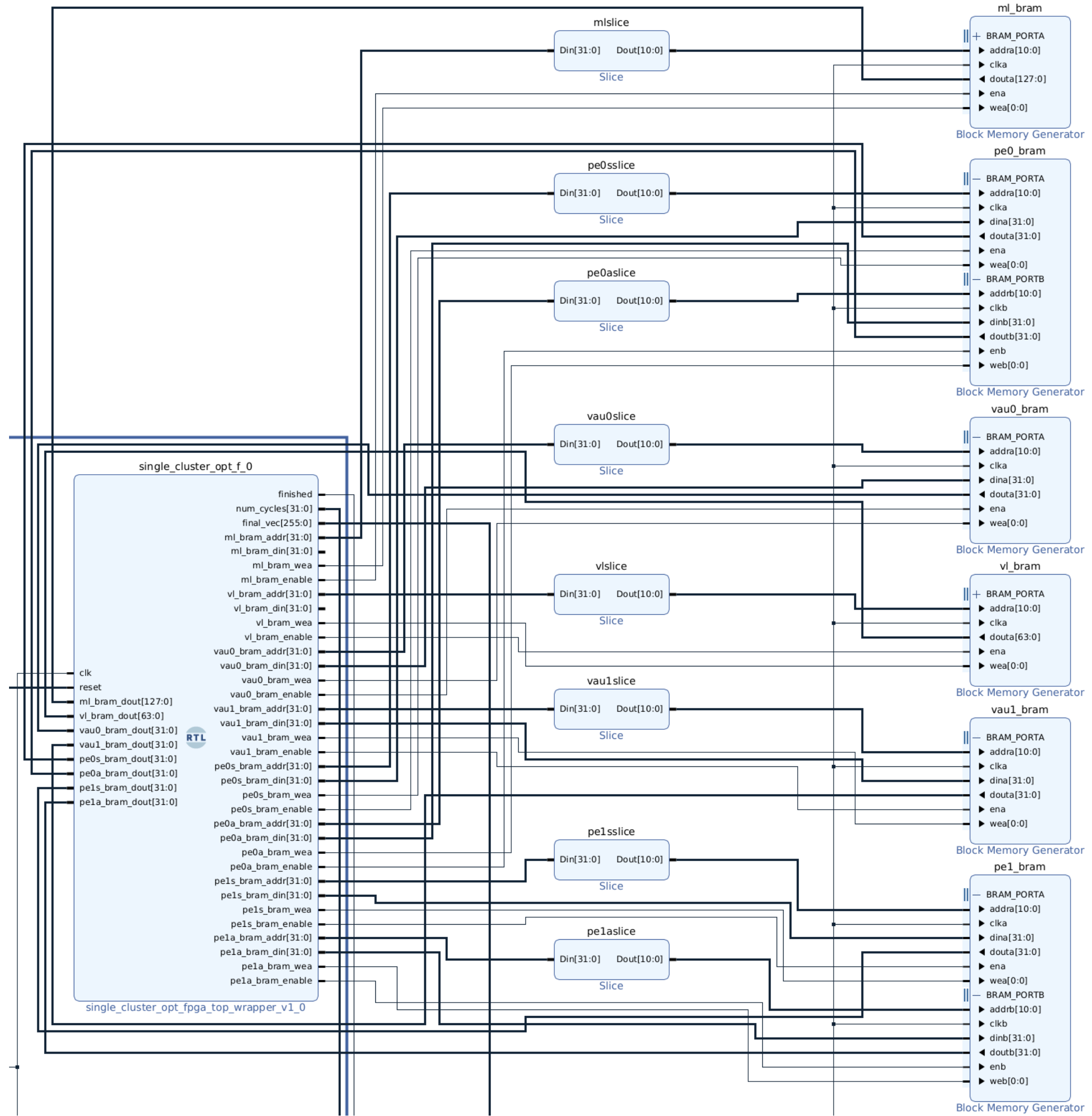


Figure 4.4: A portion of the block design containing the HiSparse RTL module and its related BRAMs. Note the processing engine's dual port nature, as pipelined accumulations require concurrent read and writes.

4.4 Hardware Results

The target frequency during synthesis was 100 Mhz. The design originally failed to meet this timing requirement and required two modifications along the critical path. The first involved a result draining module that performed a 32 bit modulo to accumulate results across all kernels and clusters. Since the design is a single kernel, single cluster variant with two streams, the bit width of the modulo's inputs could be easily reduced to meet timing without any practical loss of performance. However, the second offending path involved the processing engines' multiply accumulate. Figure 4.5 displays a setup timing report of an example offending path.

Analyzing the path's timing, it arrives at the clock edge at 14.830ns while its required time is 12.749ns, consequently overshooting the requirement by 2.081ns. A substantial amount of the timing delay is the result of high fanout clock nets. However, in the middle of the timing report lies two DSP48E1 blocks [15] in succession. The DSP48E1 is a primitive used within the Programming Logic (PL) fabric to perform the multiply accumulate operation. As the processing engine performs a multiply accumulate, it is reasonable that Vivado inferred its use. However, the DSP48E1 accepts a 25 bit and 18 bit twos complement input respectively for its multiplication operation. The original processing engine implementation performed a 32bitx32bit signed multiplication ideally within a cycle (under full throughput streaming operation). To support a higher bitwidth on both ports, multiple DSP48E1 blocks must be connected together in a hierarchical fashion, and consequently two of these DSPs end up being connected in succession. These successive DSPs contributed 6.53ns of delay, with the second DSP contributing 2.438ns. Thus allowing the matrix and vector elements to be 32 bit led to negative slack violations, and the removal of the second DSP would make the design achieve the timing requirements.

```
pub proc pe_recv {
    // ...
    let incr = ((spld.matrix_val as s18) * (spld.vector_val as s18))
   as s32;
    // ...
}
```

Listing 4.1: Single line modification necessary to acheive 100Mhz. Multiply inputs wrapped with s18 casts. Accumulation with the 32bit increment is still supported as the DSP48E1 supports up to 48 bit accumulation

Reducing the bitwidth of the matrix and vector elements to 18 bits resulted in the design achieving the timing constraints and producing the correct output as it could fit in a single DSP48E1. Figure 4.1 displays this simple modification. Though technically one of the operands can be 25bits and still fit within the DSP, XLS did not support asymmetrical bitwidths during multiplication, hence both operands are 18 bits. This modification affects performance by hurting the dynamic range of the design, though it enables its operation at 100 Mhz on the Zynq-7000 SoC. An alternative approach of breaking the 32bit multiply into parallel, smaller bit multiplications fails to meet timing due to an inability to introduce registers between intermediate results using XLS' current syntax.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	11786	0	0	53200	22.15
LUT as Logic	11574	0	0	53200	21.76
LUT as Memory	212	0	0	17400	1.22
LUT as Distributed RAM	18	0	–	–	–
LUT as Shift Register	194	0	–	–	–
Slice Registers	22380	0	0	106400	21.03
Register as Flip Flop	22380	0	0	106400	21.03
Register as Latch	0	0	0	106400	0.00
F7 Muxes	3	0	0	26600	0.01
F8 Muxes	1	0	0	13300	< 0.01

Table 4.2: Resource utilization summary from synthesis. LUTs for logic and registers used 42.79% of all slices on the Zynq-7000, while only 1% of LUTs were used as memories since the benchmarked matrix was so small. This resource utilization seems to support another cluster, kernel or more streams, though that will limit the available LUTs to be used as BRAMS and consequently the maximal input matrix and vector size.

Source Clock Path

Delay Type	Incr (ns)	Path (ns)	Location	Cell Pin	Cell	Netlist Resources
(clock clk_fpga_0 rise edge)	(r) 0.000	0.000				
PS7	(r) 0.000	0.000	Site: PS7_X0Y0	FCLKCLK[0]	PS7_i (PS7)	design_1_i/processing_system7_0/inst/PS7_i/FCLKCLK[0]
net (fo=1, routed)	1.193	1.193				design_1_i/processing_system7_0/inst/FCLK_CLK_unbuffered[0]
			Site: BU...RL_X0Y16	I	buffer_fclk_clk_0.FCLK_CLK_0_BUFG (BUFG)	design_1_i/processing_system7_0/inst/buffer_fclk_clk_0.FCLK_CLK_0_BUFG/I
BUFG (Prop_bufg_I_O)	(r) 0.101	1.294	Site: BU...RL_X0Y16	O	buffer_fclk_clk_0.FCLK_CLK_0_BUFG (BUFG)	design_1_i/processing_system7_0/inst/buffer_fclk_clk_0.FCLK_CLK_0_BUFG/O
net (fo=23482, routed)	1.634	2.928				design_1_i/single_cluster_opt_f_0/inst/dut/dut/pe1_recv/clk
FDRE			Site: SLICE_X35Y73	C	___state_0_reg (FDRE)	design_1_i/single_cluster_opt_f_0/inst/dut/dut/pe1_recv/___state_0_reg/C

Data Path

Delay Type	Incr (ns)	Path (ns)	Location	Cell Pin	Cell	Netlist Resources
FDRE (Prop_fdre_C_Q)	(r) 0.456	3.384	Site: SLICE_X35Y73	Q	___state_0_reg (FDRE)	design_1_i/single_cluster_...e1_recv/___state_0_reg/(
net (fo=335, routed)	0.537	3.921				design_1_i/single_cluster_.../dut/pe1_recv/___state_(
			Site: SLICE_X34Y74	I0	smul32b_32b_x_32b_return__0_i_4__0 (LUT2)	design_1_i/single_cluster_...x_32b_return__0_i_4__0/I(
LUT2 (Prop_lut2_I0_O)	(r) 0.124	4.045	Site: SLICE_X34Y74	O	smul32b_32b_x_32b_return__0_i_4__0 (LUT2)	design_1_i/single_cluster..._x_32b_return__0_i_4__0/O
net (fo=2, routed)	0.596	4.641				design_1_i/single_cluster_.../dut/dut/pe1_recv/lhs[13]
			Site: DSP48_X2Y29	A[13]	smul32b_32b_x_32b_return__0 (DSP48E1)	design_1_i/single_cluster...2b_x_32b_return__0/A[13]
DSP48E1 (Prop_dsp..._A[13]_PCOUT[47])	(r) 4.036	8.677	Site: DSP48_X2Y29	PCOUT[47]	smul32b_32b_x_32b_return__0 (DSP48E1)	design_1_i/single_cluster..._32b_return__0/PCOUT[47]
net (fo=1, routed)	0.056	8.733				design_1_i/single_cluster...2b_x_32b_return__0_n_106
			Site: DSP48_X2Y30	PCIN[47]	smul32b_32b_x_32b_return__1 (DSP48E1)	design_1_i/single_cluster..._x_32b_return__1/PCIN[47]
DSP48E1 (Prop_dsp48e1_PCIN[47]_P[4])	(r) 1.518	10.251	Site: DSP48_X2Y30	P[4]	smul32b_32b_x_32b_return__1 (DSP48E1)	design_1_i/single_cluster..._32b_x_32b_return__1/P[4
net (fo=2, routed)	0.920	11.171				design_1_i/single_cluster_...ut/pe1_recv/p_1_in__0[21
			Site: SLICE_X35Y74	I0	__t__accumulation_addr_reg[87]_i_9__0 (LUT2)	design_1_i/single_cluster_...ion_addr_reg[87]_i_9__0/I
LUT2 (Prop_lut2_I0_O)	(r) 0.124	11.295	Site: SLICE_X35Y74	O	__t__accumulation_addr_reg[87]_i_9__0 (LUT2)	design_1_i/single_cluster_...ion_addr_reg[87]_i_9__0/(
net (fo=1, routed)	0.000	11.295				design_1_i/single_cluster_...n_addr_reg[87]_i_9__0_n_
			Site: SLICE_X35Y74	S[1]	__t__accumulation_addr_reg_reg[87]_i_2__0 (CARRY4)	design_1_i/single_cluster_...dr_reg_reg[87]_i_2__0/S[1
CARRY4 (Prop_carry4_S[1]_CO[3])	(r) 0.550	11.845	Site: SLICE_X35Y74	CO[3]	__t__accumulation_addr_reg_reg[87]_i_2__0 (CARRY4)	design_1_i/single_cluster_..._reg_reg[87]_i_2__0/CO[3
net (fo=1, routed)	0.009	11.854				design_1_i/single_cluster_...dr_reg_reg[87]_i_2__0_n_
			Site: SLICE_X35Y75	CI	__t__accumulation_addr_reg_reg[91]_i_2__0 (CARRY4)	design_1_i/single_cluster_...addr_reg_reg[91]_i_2__0/
CARRY4 (Prop_carry4_CI_O[1])	(r) 0.334	12.188	Site: SLICE_X35Y75	O[1]	__t__accumulation_addr_reg_reg[91]_i_2__0 (CARRY4)	design_1_i/single_cluster_...dr_reg_reg[91]_i_2__0/O[
net (fo=2, routed)	0.521	12.709				design_1_i/single_cluster..._32b_x_32b_return__2[25]
			Site: SLICE_X32Y75	I5	__t__accumulation_addr_reg[91]_i_5__0 (LUT6)	design_1_i/single_cluster_...ion_addr_reg[91]_i_5__0/I
LUT6 (Prop_lut6_I5_O)	(r) 0.303	13.012	Site: SLICE_X32Y75	O	__t__accumulation_addr_reg[91]_i_5__0 (LUT6)	design_1_i/single_cluster_...ion_addr_reg[91]_i_5__0/(
net (fo=1, routed)	0.000	13.012				design_1_i/single_cluster_...n_addr_reg[91]_i_5__0_n_
			Site: SLICE_X32Y75	S[1]	__t__accumulation_addr_reg_reg[91]_i_1__0 (CARRY4)	design_1_i/single_cluster_...dr_reg_reg[91]_i_1__0/S[1
CARRY4 (Prop_carry4_S[1]_CO[3])	(r) 0.533	13.545	Site: SLICE_X32Y75	CO[3]	__t__accumulation_addr_reg_reg[91]_i_1__0 (CARRY4)	design_1_i/single_cluster_..._reg_reg[91]_i_1__0/CO[3
net (fo=1, routed)	0.000	13.545				design_1_i/single_cluster_...dr_reg_reg[91]_i_1__0_n_
			Site: SLICE_X32Y76	CI	__t__accumulation_addr_reg_reg[95]_i_1__0 (CARRY4)	design_1_i/single_cluster_...addr_reg_reg[95]_i_1__0/
CARRY4 (Prop_carry4_CI_O[3])	(r) 0.331	13.876	Site: SLICE_X32Y76	O[3]	__t__accumulation_addr_reg_reg[95]_i_1__0 (CARRY4)	design_1_i/single_cluster_...dr_reg_reg[95]_i_1__0/O[
net (fo=2, routed)	0.647	14.523				design_1_i/single_cluster...dut/pe1_recv/add_714[31]
			Site: SLICE_X33Y76	I2	___state_2[0][31]_i_1__0 (LUT3)	design_1_i/single_cluster_...__state_2[0][31]_i_1__0/I2
LUT3 (Prop_lut3_I2_O)	(r) 0.307	14.830	Site: SLICE_X33Y76	O	___state_2[0][31]_i_1__0 (LUT3)	design_1_i/single_cluster_...__state_2[0][31]_i_1__0/O
net (fo=1, routed)	0.000	14.830				design_1_i/single_cluster.../one_hot_sel_780[0]_7[31
FDRE			Site: SLICE_X33Y76	D	___state_2_reg[0][31] (FDRE)	design_1_i/single_cluster_...v/___state_2_reg[0][31]/
Arrival Time		14.830				

Destination Clock Path

Delay Type	Incr (ns)	Path (ns)	Location	Cell Pin	Cell	Netlist Resources
(clock clk_fpga_0 rise edge)	(r) 10.000	10.000				
PS7	(r) 0.000	10.000	Site: PS7_X0Y0	FCLKCLK[0]	PS7_i (PS7)	design_1_i/processing_system7_0/inst/PS7_i/FCLKCLK[0]
net (fo=1, routed)	1.088	11.088				design_1_i/processing_system7_0/inst/FCLK_CLK_unbuffered[0]
			Site: BU...RL_X0Y16	I	buffer_fclk_clk_0.FCLK_CLK_0_BUFG (BUFG)	design_1_i/processing_system7_0/inst/buffer_fclk_clk_0.FCLK_CLK_0_BUFG/I
BUFG (Prop_bufg_I_O)	(r) 0.091	11.179	Site: BU...RL_X0Y16	O	buffer_fclk_clk_0.FCLK_CLK_0_BUFG (BUFG)	design_1_i/processing_system7_0/inst/buffer_fclk_clk_0.FCLK_CLK_0_BUFG/O
net (fo=23482, routed)	1.464	12.643				design_1_i/single_cluster_opt_f_0/inst/dut/dut/pe1_recv/clk
FDRE			Site: SLICE_X33Y76	C	___state_2_reg[0][31] (FDRE)	design_1_i/single_cluster_opt_f_0/ins...dut/pe1_recv/___state_2_reg[0][31]/
clock pessimism	0.229	12.872				
clock uncertainty	-0.154	12.718				
FDRE (Setup_fdre_C_D)	0.031	12.749	Site: SLICE_X33Y76		___state_2_reg[0][31] (FDRE)	design_1_i/single_cluster_opt_f_0/ins...t/dut/pe1_recv/___state_2_reg[0][31
Required Time		12.749				

Figure 4.5: The timing report of the processing engine critical path.

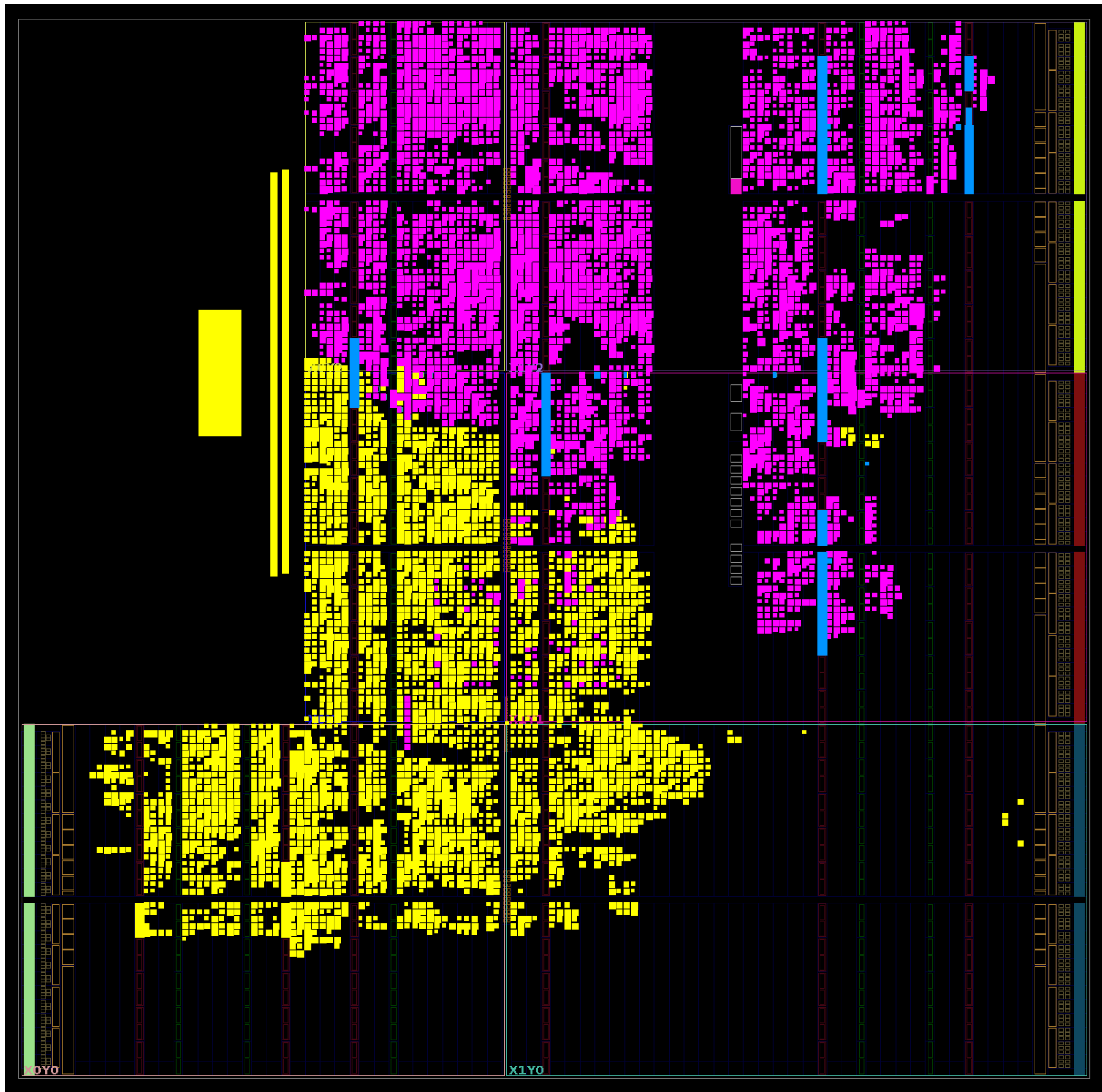


Figure 4.6: Synthesized design floorplanning. Blocks highlighted in purple represent the logic encoding the HiSparse architecture. Blocks highlighted in blue represent the BRAMs used by the HiSparse modules. All other logic (processing system, GPIO, etc) is highlighted in yellow.

Chapter 5

Conclusion

We present a case study of XLS development through the implementation of the HiSparse SpMV FPGA architecture. Due to a mismatch between the built-in XLS test framework and the codegen capabilities of the toolchain, we developed a conservative approach to avoid the related compilation issues and guarantee working hardware. Furthermore, we optimize the conservative approach using Channel Multiplexing and Frontier Splitting techniques, improving streaming throughput for all critical modules. Finally, we identify a shortcoming when attempting to synthesize the design on the Zynq 7000 at 100 Mhz. Unable to pipeline the multiplication operation across multiple cycles, we reduced the bitwidth of the multiply operation within the Processing Engine to maintain the target frequency and throughput performance, sacrificing dynamic range in the process.

Development in XLS provided noticeable benefits over typical low-level Hardware Description Languages. The latency insensitive, typed channels in procs replaced needless plumbing code while also enabling iteration on the communication protocol itself. Libraries of funcs described the message and its getters and setters, serving as the centralized authority for a particular payload. Furthermore, using tokens to reasoning about individual streams of computations within an activation was a more natural approach to handle the parallel nature of hardware designs. Rather than juggle the parallel executions within a module, tokens allowed for the explicit focus on a single chain of events while the compiler handled the overlapping of computations necessary to reach performance targets. Finally, the general ecosystem of macros, functions and bit-manipulation operators provided many

ways to approach a problem.

Despite the shortcomings of the simulation framework, the capacity for quick confidence testing within the source language sped up implementation as it avoided switching domains. Further development on the test framework to emulate the parallelism of hardware, and also ensure parity with the codegen capabilities of the design, would improve this feature. Overall the aforementioned compilation problems and resulting Channel Multiplexing and Frontier Splitting techniques reflect the current state of the compiler and do not imply fundamental limitations of the language's design. The XLS toolchain is constantly under development, and recent features such as channel strictness attributes likely mitigate previous issues [4].

Re-introducing scheduling control of operations to the user as a potential language extension may also improve the toolchain's capabilities. Syntax in Figure 5.1 represents such an extension inspired by Filament HDL [16]. The `cycles_after()` function would intake a token and a number of cycles to wait after the token is received, producing a new token to constrain future operations. Such an extension could potentially address the codegen failures of the full 32 bit processing engine mentioned previously by specifying when intermediate outputs should be staged in the pipeline.

```
1  pub proc adder {
2      //...
3
4      next( st: ()) {
5          let (tok_a, data_a) = recv(token(), a);
6          let (tok_b, data_b) = recv(token(), b);
7          let tok = join(tok_a, tok_b);
8          let tok_2 = cycles_after(tok, 2);
9          let sum@<tok, tok_2> = data_a + data_b;
10         send(tok_2, c, sum);
11     }
12 }
```

Listing 5.1: Reintroducing scheduling control to the user by extending the capabilities of tokens

XLS represents a push towards higher level HDLs aiming to streamline the hardware development process. Such innovation within language design improves access to the potential performance gains of custom hardware, providing us with better tools to tackle the

computing challenges of the current age.

References

[1] 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*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 54–64. ISBN: 9781450391498. DOI: 10.1145/3490422.3502368. URL: https://doi.org/10.1145/3490422.3502368.

[2] F. Foundation. *Cocotb*. 2026. URL: https://www.cocotb.org/.

[3] Xilinx. *AMD Vitis HLS*. 2026. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[4] G. O. Source. *XLS: Accelerated HW Synthesis*. 2026. URL: https://google.github.io/xls/.

[5] G. Kahn. "The Semantics of a Simple Language for Parallel Programming." In: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*. Ed. by J. L. Rosenfeld. North-Holland, 1974, pp. 471–475.

[6] Z. Zhang and B. Liu. "SDC-based modulo scheduling for pipeline synthesis." In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013, pp. 211–218. DOI: 10.1109/ICCAD.2013.6691121.

[7] NVIDIA. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. 2021. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[8] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher. "ExTensor: An Accelerator for Sparse Tensor Algebra." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 319–333. ISBN: 9781450369381. DOI: 10.1145/3352460.3358275. URL: https://doi.org/10.1145/3352460.3358275.

[9] W. Snyder. *Verilator*. 2026. URL: https://www.veripool.org/verilator/.

[10] L. University. *Surfer*. 2026. URL: https://surfer-project.org/.

[11] I. (Advanced Micro Devices. *AMD Vivado Design Suite*. 2026. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html.

[12] Xilinx. *Block Memory Generator v8.4*. 2026. URL: https://docs.amd.com/v/u/en-US/pg058-blk-mem-gen.

[13] Xilinx. *Vivado Design Suite User Guide: Designing with IP (UG896)*. 2026. URL: https://docs.amd.com/r/en-US/ug896-vivado-ip/Using-a-COE-File.

[14] Xilinx. *AXI GPIO LogiCORE IP Product Guide (PG144)*. 2026. URL: https://docs.amd.com/r/en-US/pg144-axi-gpio.

[15] Xilinx. *Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953)*. 2026. URL: https://docs.amd.com/r/en-US/ug953-vivado-7series-libraries/DSP48E1.

[16] R. Nigam, P. H. Azevedo de Amorim, and A. Sampson. "Modular Hardware Design with Timeline Types." In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591234. URL: https://doi.org/10.1145/3591234.