Optimizing Serpens SpMV Accelerator on Xilinx Alveo U280

Project Report
EECE 5698 : FPGA in Cloud
Aditya Padamwar
Northeastern University
Email: padamwar.a@northeastern.edu

***Abstract*—This project implements a high-throughput sparse matrix–vector multiply (SPMV) accelerator named Serpens on the Xilinx Alveo U280 platform using 24 HBM-backed sparse channels. The accelerator computes $Y = \alpha AX + \beta Y$ and is adapted from the original TAPA-based Serpens A16 version to a fully Vitis HLS dataflow kernel with expanded parallelism, improved memory scheduling, and completely removed TAPA dependencies.**

I. Project Description

The work focuses on an FPGA/HLS implementation of the Serpens sparse accelerator using the Xilinx Alveo U280 with 24 HBM channels. The design performs sparse matrix–vector multiplication using a windowed edge-list scheduling strategy and fully unrolled processing elements (PEs).

A. Key Components

- Kernel: serpens.h and serpens.cpp define the HLS dataflow kernel with 24 parallel PEs. Each PE streams a windowed edge list, caches X in URAM, multiplies and accumulates edge values, and outputs partial Y. Arbiters merge 3-PE groups (eight groups total) before scaling by alpha and beta and writing packed 512-bit Y values.
- Host runtime: host_xrt.cpp uses XRT to load a sparse matrix in .mtx format and converts it to CSC/CSR to generate and schedule per-PE edge lists then it packs A, X, and Y into 512-bit aligned words and allocates the HBM buffers and launches the Serpens kernel, time execution, and compares FPGA results with a CPU CSR baseline.
- Sparse utilities: sparse_helper.h and mmio.h perform matrix parsing, symmetric expansion, COO sorting, CSR/CSC conversion, CPU SPMV, and generation of 64-bit edge records (14-bit column, 18-bit row, 32-bit value) arranged for multi-PE scheduling.
- Build/configuration: run_hls.tcl drives Vitis HLS for the U280 target with a 6.67 ns clock and exports Serpens.xo. link_config_u280_24ch.ini maps the 24 edge channels and X/Y buffers to specific HBM banks.
- Data generation: spmvgen.py creates random square .mtx matrices at multiple sizes.

II. Code Description

- serpens.h
  - It defines compile time constants for the 24-channel SpMV design:
    - NUM_CH_SPARSE, WINDOW_SIZE, dependency distance.
    - X-vector partitioning and URAM depth.
  - It declares lightweight vector structures (Vec, float_v16, float_v8, float_v2).
  - It provides bit-cast helpers to_float and to_uint for IEEE float and ap_uint reinterpretation.
  - It supplies SIMD helpers vec_add and vec_mul operating on 16-lane floating vectors.
  - It declares the top-level HLS kernel Serpens(...) including 24 edge-list ports, X/Y input buffers, Y output buffer, and scalar arguments (NUM_ITE, NUM_A_LEN, matrix sizes, P_N, alpha_u, beta_u).
- serpens.cpp
  - Implements the full Vitis HLS dataflow kernel.
  - Does packing and unpacking helpers convert between 16 floats and 512-bit AXI words.
  - Input Readers:
    - read_edge_list_ptr, read_X, and read_A fetch instruction streams, X tiles, and per-channel edge blocks.
    - Support repetition factor P_N.
  - PEG_Xvec path:
    - Tiles X into URAM.
    - Streams channel-aligned edge blocks.
    - Multiplies edge values with cached X.
    - Emits MultXVec records with metadata for Y accumulation.
  - Y Accumulation (PUcore_Ymtx / PEG_Yvec):
    - Uses per-PE URAM buffers for partial sums.
    - Applies dependency distance rules.
    - Produces packed 2-float outputs per address.
  - Output Merging:
    - Arbiter_Y merges 3 PE's in round-robin fashion.
    - Merger_Y combines eight arbiter outputs into 16-lane float vectors.

  - Post-processing:
    * FloatvMultConst applies alpha or beta scaling.
    * FloatvAddFloatv computes AX + Y.
    * write_Y stores packed vectors back to HBM.
  - It also contains drain utilities for pipeline balancing.
  - The top-level Serpens kernel instantiates 24 PE pipelines, binds AXI masters to HBM banks, configures AXI-Lite controls, and enables top-level dataflow parallelism.
- host_xrt.cpp
  - Provides host runtime, preprocessing, and result validation.
  - Parses command line arguments: ¡xclbin¿ ¡matrix.mtx¿ [rp_time] [alpha] [beta] with defaults 0.85, -2.06, and 1.
  - Matrix processing:
    * Reads MatrixMarket input using read_suitsparse_matrix.
    * Converts CSC to CSR.
    * Creates reference X and Y vectors.
  - Edge-list generation:
    * generate_edge_list_for_all_PEs partitions edges across 24 PEs.
    * edge_list_64bit packs edges into 64-bit records (row, col, value).
    * All lists are padded to 1024-aligned boundaries.
  - Packs X and Y into 512-bit aligned buffers.
  - Runs CPU CSR SPMV for golden reference.
  - XRT runtime:
    * Loads xclbin and constructs kernel handle.
    * Allocates Buffer Objects for edge pointers, 24 channels, X, Y, and output buffers.
    * Maps, fills, and synchronizes host-to-device transfers.
    * Launches kernel and synchronizes results back.
  - Unpacks Y and checks for accuracy and throughput (in GFLOPS).
- mmio.h
  - Provides matrix parsing utilities.
  - Handles banners, typecodes, sizes, and format validation.
  - Supports real, integer, pattern, and complex matrices in coordinate or array formats.
  - Includes helpers such as mm_strdup.
- sparse_helper.h
  - Contains sparse preprocessing utilities and data structures including rcv triples, edge structs, CSR/CSC enums.
  - Sorting helpers for row-major and column-major orders.
  - Matrix loading:
    * mm_init_read and load_S_matrix handle symmetric expansion and pattern values.
    * read_suitsparse_matrix constructs sorted CSR or CSC structures.
  - CPU reference kernel: cpu_spmv_CSR.
  - Edge scheduling:
    * generate_edge_list_for_one_PE enforces dependency spacing.
    * generate_edge_list_for_all_PEs partitions edges into windowed blocks and balances channels.
    * edge_list_64bit packs edges into 24 HBM channels using sentinel padding.
- link_config_u280_24ch.ini
  - Vitis connectivity configuration mapping AXI masters to U280 HBM banks.
  - Assigns edge pointer buffers, 24 channel streams, and X/Y/Y_out buffers to dedicated HBM banks.
  - Ensures host and kernel use identical bank placement.

III. Optimization Strategies

The Serpens accelerator uses a variety of hardware and software optimizations that enable high throughput sparse matrix–vector multiplications on the U280, these optimizations include parallelism, dataflow, locality, memory layout, scheduling, and host-side preprocessing.

A. Parallel Sparse Processing

The kernel exploits twenty-four-way sparse parallelism by initiating twenty-four fully unrolled processing element pipelines. Each PE is connected to a dedicated AXI master interface which is mapped to a separate HBM bank through interface bundle assignments and the link_config_u280_24ch.ini connectivity script which allows all HBM channels to operate concurrently making sure that every PE receives its own edge stream without conflicts.

B. Streaming Dataflow Execution

A top-level dataflow region enables concurrent execution of all major pipeline components, including edge-list readers, X loaders, compute PEs, arbiters, mergers, scaling units, and writers. All internal loops are pipelined with II = 1 which allows each stage to accept new data every cycle. This streaming organization prevents back pressure and ensures full throughput.

C. Locality and Bandwidth Optimization

Each PE caches its assigned X-tile segments in URAM using storage binding and array partitioning. This eliminates repeated HBM reads and supports cycle-by-cycle access during computation. Similarly, partial Y accumulators are maintained in URAM which stores two floating-point values per 64-bit entry to match the update granularity of the compute units. These local memories enable sustained II = 1 operation on the compute pipeline.

D. SIMD and Wide-Word Packing

All X, Y, and edge data move through 512-bit AXI interfaces for maximum bus utilization. The kernel unpacks and repacks sixteen-lane floating-point vectors using dedicated helpers, and edges are constructed as eight 64-bit packed entries per cycle. Host-side preprocessing aligns all buffers to 1024-byte boundaries to facilitate long, burst-friendly HBM transactions.

E. Edge Scheduling and Hazard Avoidance

The preprocessing routines in sparse_helper.h schedule edges per PE and per window which enforces dependency distances to preserve correctness during streamed accumulation. All PEs receive balanced window lengths to prevent pipeline starvation. Edges are encoded into compact 64-bit records and mapped deterministically to the twenty-four channels, ensuring hazard-free execution without runtime reordering.

F. Compute and Merge Pipeline Organization

Each PE processes up to eight edges per cycle, multiplies values with cached X entries, and accumulates local results. Groups of three PEs feed into 3:1 arbiters, and the outputs of eight arbiters are combined in an eight-way merger to form contiguous sixteen-element vectors. This organization ensures that downstream scaling and addition stages remain fully utilized.

G. Loop and Array-Level Optimizations

Inner loops are extensively unrolled across lanes and PEs, enabling parallel operations within each cycle. Array partitioning provides multiple simultaneous read and write ports, supporting the bandwidth required for vector operations, accumulator accesses, and URAM-stored X tiles.

H. In-Stream Scaling and Output Computation

Alpha and beta scaling, as well as the final AX+Y addition, are applied directly on the output stream prior to writing results to HBM. This avoids additional memory passes and minimizes overall latency.

I. Host Runtime Efficiency

The host program precomputes all edge lists, performs 512-bit packing, and allocates XRT buffer objects individually for each HBM bank. Explicit synchronization and alignment ensure minimal data-movement overhead and guarantee that the runtime matches the layout expected by the kernel. Launch overhead is reduced by batching preprocessing steps and avoiding redundant transfers.

IV. Code Source and Adaptation

The original Serpens accelerator was obtained from the public repository at https://github.com/linghaosong/Serpens. The provided implementation used the TAPA programming model and targeted a 16-channel HBM configuration. This project used the A16 version of the codebase as the foundation. All TAPA dependencies were removed, and the design was rewritten in pure Vitis HLS while retaining the pipeline structure. The architecture was expanded to support 24 HBM channels on the Alveo U280, including updates to edge scheduling, memory mapping, vector datapaths, and kernel interfaces, enabling higher parallelism without the TAPA runtime.

V. Integrated Hardware–Software Architecture

The Serpens accelerator is implemented through a coordinated combination of FPGA-side HLS kernels and software-side preprocessing and runtime management. Together, these components form a complete execution pipeline from Matrix-Market input to validated numerical output.

A. FPGA Hardware Kernel

The hardware implementation is defined in serpens.h and serpens.cpp, which together describe the Serpens dataflow kernel for sparse matrix–vector multiplication on the U280. The kernel incorporates streaming loaders that fetch windowed edge lists and X tiles directly from HBM, followed by twenty-four parallel processing element pipelines. Each pipeline performs URAM-based caching of the corresponding X segment, executes multiply–accumulate operations for edges assigned to that PE, and generates partial contributions to Y. Arbiter and merger stages combine the outputs of the PEs into aligned 16-lane vector blocks. These vectors pass through alpha and beta scaling units before being written back into HBM through 512-bit AXI ports. The build artifacts Serpens.xo and Serpens.xclbin encapsulate the synthesized kernel, while link_config_u280_24ch.ini specifies the memory connectivity required to map all sparse channels and vector buffers to distinct HBM banks.

B. Software Runtime and Preprocessing

The software infrastructure is centered around host_xrt.cpp, which serves as the XRT-based launcher and validator. The host program loads a MatrixMarket file, parses and converts it into CSR and CSC forms, constructs per-PE edge lists according to the accelerator's scheduling model, and packs the sparse matrix, X, and Y buffers into 512-bit aligned memory blocks matching the kernel's interface. These buffers are allocated in device HBM, transferred from host memory, and used to initialize kernel execution. Upon completion, the output Y vector is copied back, unpacked into floating-point values, and compared against a CPU CSR baseline.

The utility modules sparse_helper.h and mmio.h perform the supporting operations needed for preprocessing, including MatrixMarket parsing, COO sorting, CSR and CSC construction, CPU SPMV computation, and the encoding of edges into the 64-bit format expected by the hardware. For generating additional test matrices, spmvgen.py creates synthetic MatrixMarket files with controllable sparsity patterns suitable for performance stress-testing.

Together, the hardware and software components form a tightly integrated pipeline in which preprocessing shapes the sparse matrix into a hardware-friendly format, the FPGA kernel executes the core computation in a highly parallel HBM-driven dataflow, and the runtime verifies correctness and captures performance metrics.

C. Inputs and Validation

- spmvgen.py creates random square MatrixMarket files at four scales (50k, 100k, 500k, 1M). For each size it chooses a fixed NNZ-per-column (70, 80, 90, 100 respectively), writes the header, then for every column randomly samples that many row indices and emits entries with value 1.0. Output filenames follow `matrix_<N>_<N>_<totalNnz>.mtx`.
- Validation: the host computes a CPU reference SPMV result, runs the FPGA kernel on the same A/X/Y, unpacks the device output, and compares element-wise with a small relative tolerance (1e-4), reporting mismatch count and pass/fail.

VI. Results

This section summarizes the performance and resource utilization of the Serpens accelerator on the Alveo U280. The evaluation compares CPU baseline performance, FPGA kernel execution time, throughput, and post-implementation resource usage. All experiments were conducted using synthetic sparse matrices of varying sizes.

A. Execution Time Comparison

Figure 1 illustrates the execution time of the CPU CSR baseline and the FPGA kernel across four matrix sizes. The FPGA achieves significantly lower execution time, particularly for large matrices, due to parallelism enabled by twenty-four HBM-backed sparse channels.

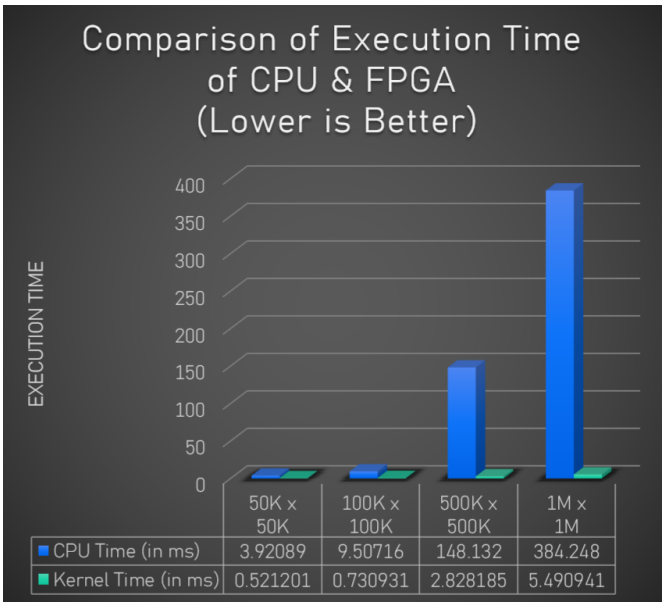

Fig. 1. Comparison of execution time between CPU and FPGA (lower is better).

B. Throughput Comparison

Figure 2 shows the throughput measured in GFLOPS. While CPU throughput drops significantly as matrix size increases, the FPGA kernel maintains high throughput due to efficient streaming, URAM caching, and parallel PE execution.

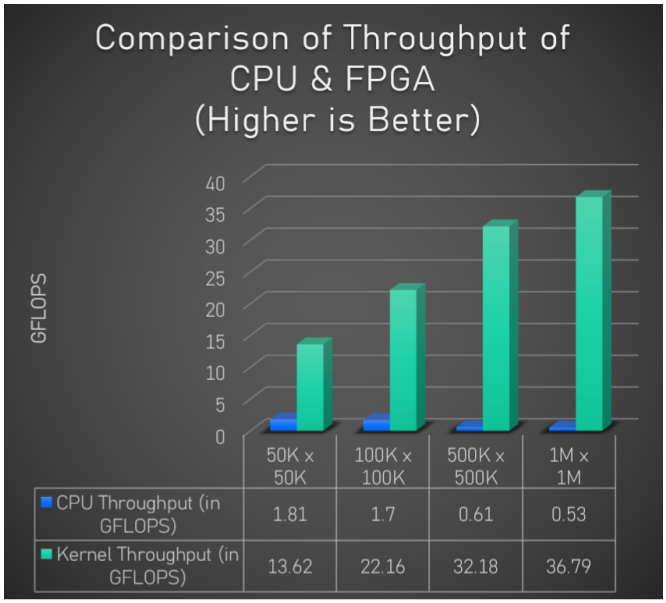

Fig. 2. Comparison of throughput of CPU and FPGA (higher is better).

C. Resource Utilization

Table VI-C presents resource usage from the U280 implementation. LUT and FF usage remains modest relative to device capacity. DSP usage is low because the design relies primarily on LUT-based floating-point operations. BRAM consumption remains under 30 percent, while URAM is fully utilized due to per-PE X-vector caching and accumulation structures.

Resource	Used	Available	Utilization (%)
LUT	368344	1303680	28.25
FF	521693	2607360	20.01
DSP	1358	9024	15.05
BRAM	595	2016	29.51
URAM	960	960	100.00

Table VI-C. FPGA resource utilization on the Alveo U280.

D. Power Metrics of the FPGA

Power measurements were collected using the Xilinx xbutil electrical monitor while executing the largest test case (1M by 1M matrix with ninety million nonzeros). The U280 reported an average board power of approximately 32.9 W, which is significantly below its 225 W maximum power envelope. No thermal or power warnings were observed during kernel execution.

The primary FPGA rail (VCCINT) drew roughly 10.68 A at 0.85 V, reflecting the activity of the compute pipelines and memory interfaces. Auxiliary rails such as the PCIe and 12 V inputs remained lightly loaded, indicating that on-chip logic and HBM controllers dominate overall power consumption.

These measurements demonstrate that the accelerator delivers high throughput while maintaining relatively low power draw, emphasizing the efficiency benefits of a deeply streamed, parallel sparse compute architecture.

E. Comparison Against CPU and GPU Implementations

In addition to CPU comparisons, performance was evaluated against a custom CUDA SpMV kernel running on an RTX 4070 Mobile GPU at a 130 W overclocked configuration. The CUDA implementation employed warp-level reduction strategies, 2D tiling, and vectorized memory accesses tailored for the benchmark's sparsity pattern. It achieved a runtime of 3.079 ms corresponding to approximately 58.5 GFLOPS.

The FPGA implementation, operating at roughly 33 W and a lower clock frequency, achieved 36–37 GFLOPS through twenty-four parallel HBM-backed sparse processing elements. Although its raw throughput is lower than the GPU, the power efficiency is substantially higher. The FPGA provides deterministic streaming behavior, customizable dataflow parallelism, and explicit control over memory architecture, all of which contribute to sustained performance at low power cost.

The GPU excels in peak throughput due to its large number of SIMD cores and high-frequency operation, but requires a significantly higher power budget. The FPGA, by contrast, offers a better balance of energy efficiency, architectural specialization, and scalability for sparse workloads.

VII. Scaling Challenges and Lessons Learned

Scaling the Serpens accelerator beyond its original configuration revealed several architectural and physical limitations that directly influenced achievable throughput, clock frequency, and overall system stability. Although the U280 provides thirty-two HBM channels, practical scaling was constrained by routing congestion, timing closure difficulty, and per-channel memory capacity.

A. Impact of Congestion on Timing

A primary challenge encountered during scaling was routing congestion across the programmable logic fabric. While logic resources such as LUTs, flip-flops, and DSPs remained well below device limits, the interconnect fabric became saturated as additional processing elements and memory interfaces were instantiated. As a result, timing closure at target frequencies such as 250 MHz became unattainable; even modest increases in compute parallelism caused significant negative slack. The final design operated reliably at approximately 150 MHz, which provided stable timing while still enabling high throughput through extensive parallelism.

B. Scaling Limits of HBM Connectivity

Despite the presence of thirty-two physical HBM channels, only twenty-four channels could be used effectively in this design. This mirrors the behavior reported in the original Serpens work, but the limitation was more pronounced here due to the absence of the TAPA programming framework, which provides automated task-level parallelism and improved routing structure. Without TAPA, routing complexity increased, crossbar pressure intensified, and scaling the remaining channels resulted in unroutable designs or timing failures. This demonstrated that scaling is limited not only by available hardware channels but also by achievable physical routing patterns.

C. HBM Capacity Constraints

Each HBM channel on the U280 provides approximately 256 MB of storage. Although the total device capacity is high, per-channel capacity becomes a critical constraint in sparse workloads. Depending on sparsity and distribution, a channel may fill its allocated memory region before all required edges for its matrix window can be assigned. This restricts scalability for very large matrices and forces the scheduling system to either rebalance loads or limit matrix dimensions. Thus, per-channel capacity, not total HBM capacity, becomes the bottleneck in multi-channel sparse workloads.

D. Routing, Parallelism, and the Absence of TAPA

In contrast to the original Serpens A16 implementation, which used TAPA to enforce clean task boundaries and simplify wiring between components, this work reimplemented the accelerator entirely in Vitis HLS. While this approach eliminated dependencies and gave full control over the structural design, it significantly increased routing burden. TAPA's graph-based scheduling and automatic interconnect reduction were absent, so manually unrolled pipelines and AXI interface bundles produced long and dense wiring paths. This contributed to timing failures when attempting to use more than twenty-four channels or target higher clock frequencies.

E. Potential Improvements

Several improvements could help increase performance, scalability, and clock frequency:

- By the end of the semester I would like to implement Tiling. Tiling strategies could be introduced so that only subregions of the matrix are resident per channel at any given time. However, the small per-channel HBM capacity still limits how effectively large matrices can be partitioned.
- Using a device with larger total HBM capacity or larger per-channel capacity would support substantially larger matrices and reduce pressure on the edge scheduler.
- Reintroducing TAPA would likely improve routability and help achieve higher clock frequencies by structuring the communication graph more efficiently.

These observations indicate that practical performance is determined not only by compute resources but also by the physical limits of routing, interconnect topology, and per-channel memory distribution. Effective scaling thus requires co-design across architecture, compilation strategy, and memory layout.

References

[1] A. Akopyan, H. Fu, et al., "Serpens: A Sparse Linear Algebra Accelerator with Decoupled Memory Access and Execution," arXiv:2111.12555, 2021. Available: https://arxiv.org/abs/2111.12555

[2] UCLA VAST Lab, "Serpens: Sparse Linear Algebra Accelerator," GitHub Repository, Available: https://github.com/UCLA-VAST/Serpens

[3] A. Akopyan, H. Fu, et al., "Serpens: A Sparse Linear Algebra Accelerator with Decoupled Memory Access and Execution," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022.

[4] A. Li, S. Chen, et al., "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication," in *47th International Symposium on Computer Architecture (ISCA)*, 2020.
[5] S. Yang, et al., "GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs," in *SC20: International Conference for High Performance Computing*, 2020.
[6] AMD Xilinx, "Alveo U280 Data Center Accelerator Card: Product Documentation," Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html
[7] AMD Xilinx, "Vitis High-Level Synthesis User Guide, Version 2023.2," Available: https://docs.xilinx.com/
[8] AMD Xilinx, "Xilinx Runtime (XRT) Software Stack Documentation," Available: https://xilinx.github.io/XRT/
[9] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.
[10] R. Boisvert et al., "Matrix Market Exchange Formats," NIST Technical Report, 1997. Available: https://math.nist.gov/MatrixMarket/