Streamed Multi-Format Sparse Matrix Vector Multiplication for FPGA

Spencer Smith
School of Computer Science
University of Oklahoma
Norman, United States
spencer.smith-1@ou.edu

Richard Veras
School of Computer Science
University of Oklahoma
Norman, United States
richard.m.veras@ou.edu

***Abstract*—Sparse Matrix Vector Multiplication (SpMV) is an important and fundamental operation used in high-performance computing. However, the potential for irregular sparsity as well as unique memory constraints have led to the development and use of many disparate storage formats. We propose a new 'meta' format that could take advantage of any number of previous formats, while also taking advantage of the single matrix access nature of SpMV. In addition, we implement this using Vitis HLS for FPGA. In this approach, a much larger matrix is decomposed into smaller submatrix blocks of uniform size but of arbitrary existing formats and streamed in to be processed on FPGA. On FPGA, each block contains metadata that allows dispatch to format-specific computation modules. Preliminary tests for matrices up to 2048x2048 demonstrate functionality and efficient use of memory and resources. We chose FPGA, specifically HLS for its accessibility, to explore potential future optimizations in parallelizing individual matrix block computations.**

***Index Terms*—FPGA, SpMV, Sparse Matrix, HLS**

I. Introduction and Background

SpMV is used in many different practical applications today, all manners of scientific computing, machine learning, image processing, and more. In the case of SpMV where, on most modern machine architectures and in most implementations, the problem is memory bound [1]. Because the computation is arithmetically simple, usually involving only a floating point addition and multiplication [1], much work has been done to research and optimize the storage of these sparse matrices [2], [3]. And though some perform generally better than others, there is no one size fits all format yet discovered. Often times the best format might depend on the given machine architecture, and especially on the sparseness pattern [2]. Irregular sparsity has proven to be a particularly difficult problem to grapple with. Some have thought to combine different formats into so-called 'hybrid' formats to take advantage of cases in which not all of the matrix is irregular [4]. But these implementations have focused on just a few predetermined and specifically chosen matrix formats, and in addition they have been only tested on traditional hardware. Most research and experimentation thus far has been performed on CPU, with focus being placed mostly on SIMD and Thread parallelism as acceleration [1]. In the case of CPU parallelism, threads allow much greater use of available cache, taking advantage of the memory bound nature of the problem, but there is still performance to be squeezed out with reorganizing the memory representation [5]. Further, as GPUs have become increasingly popular as parallel accelerators, much research has been conducted in adapting these for use in SpMV. However, the inherently irregular access pattern and load imbalances has proven to be an issue for the SIMT architecture of GPUs [6], [7]. Formats such as ELLPACK have been developed to minimize this with padding to ameliorate the imbalance of parallel loads [1] [4], but nonetheless results in unnecessary work. FPGAs prove to be one other avenue for acceleration, in some cases sharing the higher memory bandwidth advantage with GPUs, and often having greater power efficiency [8], a growing concern with GPUs [9]. As well their has been some work in taking advantage of the data streaming capabilities offered by FPGAs, with promising results in addressing both increased performance in some cases and lower power consumption [10]. But there still seems to be room for new implementations focusing on a much wider selection of formats, with tunable granularity, developed for FPGA hardware that would be effective at handling many different formats and irregular access patterns, hopefully in a parallel fashion.

II. Theory

To take full advantage of the unique characteristics of highly irregular SpMV operations, we propose a new streaming multi-format structure. Where single-format matrices may succeed in general or uniform cases, there are others where the sparsity is not uniform. In these cases it may be beneficial to store parts of the larger matrix in the most efficient format. As well, because each value in a matrix for matrix-vector multiplication is read from only once, a streaming data structure may be particularly suited to the task. Furthermore, because of the heterogeneous logic involved in the different algorithms tailored to the different formats, current GPU hardware accelerators may not be sufficient, should any type of parallelism be pursued. Therefore, to address this unique challenge, we propose implementing this structure on FPGA. In summary, we propose a multi-format matrix block stream, in which block sizes may be of arbitrary granularity and of arbitrary format to best exploit the data layout on a matrix by matrix basis.

III. Mechanism

A. Implementation

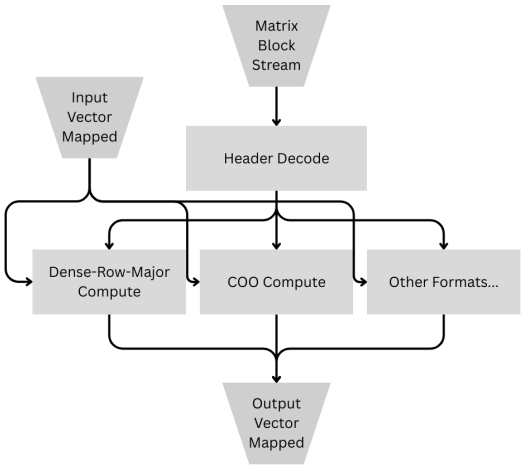

Fig. 1. Current Sequential Implementation

In this initial exploratory implementation, Figure 1, only two simple submatrix format options were chosen: Sparse coordinate (COO) and dense row-major / C-style. These were chosen for the simplicity of streaming. As well [11] found it to be quite resistant to very irregular matrices, meaning a smaller variance. In matrix-vector multiplication, each element of the matrix must only be visited/read from once, making a stream a satisfactory data structure for storing and retrieving matrix elements as they are needed. Vitis also provides the hls::stream library for just this purpose. By default, this wraps around a FIFO port when placed on the top interface, so out of order accesses are not allowed [12]. This constraint is not an issue for the two formats chosen. The test bench takes matrices already in COO format sorted by row, then column and divides them into blocks. Each block is tested for a simple threshold of sparsity:

$$\text{Format} = \begin{cases} \text{Dense Row-Major}, & \text{if } 8mn \leq 24nnz, \\ \text{COO}, & \text{otherwise.} \end{cases} \quad (1)$$

Here, m and n denote the number of rows and columns in the block, respectively, and nnz is the number of non-zero elements in the block. The constants 8 and 24 represent the number of bytes required to store one element in dense and COO formats, respectively. This heuristic favors dense format when its storage cost is less than or equal to that of COO. Future work may explore more sophisticated format-selection rules.

After format-selection, a header is constructed for the data:

```
struct Mix_Matrix_Header
{
    Matrix_Format  format;  // Enum: NONE = 0,
     DENSE = 1, COO = 2
    uint64_t       vec_in_offset;  // Input
    vector offset
    uint64_t       vec_out_offset; // Output
    vector offset
    uint64_t       vec_count;  // Number of
    rows in the tile
    uint64_t       data_size;  // Stream
    offset to next tile
};
```

Listing 1. Block header structure for mixed-format streaming.

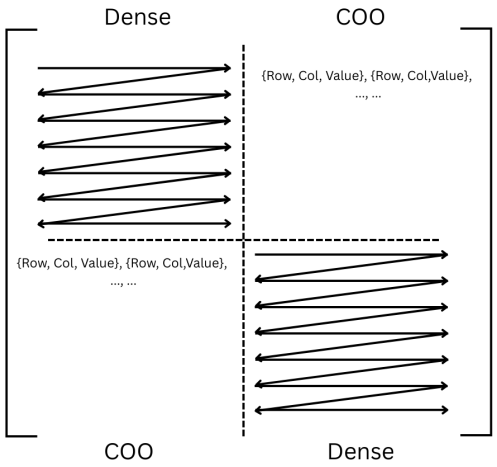

Fig. 2. Conceptual View of Matrix Stream

Here, each header contains a bit of information about each following block, most importantly the storage format, and the block offsets into the respective input and output vectors. Matrix data is then written into the stream in the specified format, and in the order the values will be consumed. Values in this implementation are all doubles or 64 bit floats. For the COO format, first the number of non zeroes in the block as a uint64 is streamed, then values are streamed as triples of $(row, column, value)$, exactly the order the COO compute function consumes them. And for Dense, first the number of rows and columns in the block are streamed first, again as uint64's, and then values in row major order. Conceptually, it might be thought of as Figure 2.

TABLE I
Stream Memory Layout: fixed-size header followed by format-specific data.

Bytes	Field	Description
0–39	Header	Format ID, offsets, vector count
40–(40+N)	Dense Payload	$m \times n$ values
...		
X–(X+39)	Next Header	Same format as first header
...	COO Payload	$nnz \times (row, col, val)$ triples
Y–(Y+39)	NONE Header	Signifies the end of the stream

In reality the stream memory layout is demonstrated by Table I. This stream is then consumed on FPGA, dispatching to the correct format compute function based on the header, Listing 2.

```
void mix_matrix_mul_vector_top(Data_Stream *
    mix_stream, f64 *vec_in, u64 vec_in_count,
     f64 *vec_out, u64 vec_out_count)
{
  ...
  do
  {
    head = read_matrix_header(mix_stream);

    switch (head.format)
    {
      case Matrix_Format::NONE:
      {
        hls::print("NONE, exiting...\n");
      }
      break;
      case Matrix_Format::DENSE_ROW_MAJOR:
      {
        dense_row_major_mul_vec(mix_stream,
    head, vec_in, vec_in_count, vec_out,
    vec_out_count);
      }
      break;
      case Matrix_Format::COO:
      {
        coo_mul_vector(mix_stream, head,
    vec_in, vec_in_count, vec_out,
    vec_out_count);
      }
      break;
    }
  }
  while (head.format != Matrix_Format::NONE);
}
```

Listing 2. HLS Stream Processing on FPGA

```
void coo_mul_vector(Data_Stream *mix_stream,
    Mix_Matrix_Header head, f64 *vec_in, u64
    vec_in_count, f64 *vec_out, u64
    vec_out_count)
{
  u64 non_zero_count = mix_stream->read();
  hls::print("COO:\n");
  hls::print("  non_zero_count = %d\n",
    non_zero_count);

  COO_LOOP:
  for (usize i = 0; i < non_zero_count; i++) {
    u64 coo_row   = mix_stream->read();
    u64 coo_col   = mix_stream->read();
    f64 coo_value = read_f64_from_stream(
    mix_stream);

    u64 vec_in_index  = coo_col + head.
    vec_in_offset;
    u64 vec_out_index = coo_row + head.
    vec_out_offset;

    f64 current   = vec_out[vec_out_index];
    f64 vector_in = vec_in[vec_in_index];

    vec_out[vec_out_index] = current + (
    coo_value * vector_in);
  }
}
```

Listing 3. HLS Stream COO Block Compute

One can observe that this could be expanded to any number of potential formats, as long as they were conducive to streaming. The compute function/module for COO is given in Listing 3. In hindsight, choosing COO proved to be a little hazardous in the read-modify-write-cycle. Ensuring correctness required disabling pipelining for the COO_LOOP, greatly limiting potential throughput. This revealed a major limitation of some formats that have very irregular access like COO for the streaming, pipe-lined architecture of FPGAs. We see this as an important lesson guiding future work in exploring other formats. For now these are relatively naive implementations, but one could imagine using the given format's fastest implementation for each compute module, combining the most effective implementation with the most effective format for each blocks data layout. However, this first implementation is very sequential, having only one stream and writing potentially to the same rows in the memory mapped output vector. Both the input and output vectors are AXI Master memory mapped, since we are accessing these values multiple times. Nonetheless we take advantage of Matrix-Vector multiplication's need to access each matrix value only once with the use of a streaming data structure. The tooling used for this implementation was AMD's Vitis HLS suite [12], and the board we used for our simulations was the Pynq-7000, specifically the xc7z020clg400-1.

IV. Analysis

A. Preliminary Results

We created a simple script to generate sparse matrices in edge list format. For this simple exploratory implementation we focused on 2 cases a small 32x32 matrix for initial testing, and a much larger matrix of 2048x2048, for a more rigorous test case. In all cases our simple heuristic for deciding format usage achieved a small data footprint, even including stream header overhead, see Table II. However, the simplicity of the rule leaves much to be desired, and further work should be done, both in finding a more efficient rule perhaps even machine learning based, as Chen et al. have [4], as well as expanding the available formats. However, we are leaving much of the available hardware unused, as seen by Table III. We would like to achieve a much fuller saturation of the DSP units, as well as at least some usage of the fast BRAM. But, there is potential in the streaming structure [10], one could imagine having a much larger sparse matrix that one does not want to materialize all at once. One could instead only materialize by block, and at that moment also decide the format for streaming. This is especially useful as those blocks can then be dematerialized as they are consumed on FPGA, as stated before matrix-vector multiplication only requires one read for any one value of the matrix.

B. Future Work

Though we have the current working implementation, we believe there is much more potential in this idea. So far, we

TABLE II
MEMORY USAGE FOR MATRIX SIZE

Matrix Dimension	Memory Footprint (KB)
32x32	0.856
2048x2048	9827.224

TABLE III
FPGA SYNTHESIS RESOURCE USAGE

BRAM Usage	DSP Usage	LUT Usage	FF Usage
2%	6%	4%	10%

have not fully utilized the strengths of FPGA as a platform, namely parallelism and BRAM usage. Therefore we propose an improvement to the current design, seen in Figure 3. This design would include alongside the other decode and compute modules, a final reduction module. In this implementation, each block and compute module would receive a slice of BRAM to act as a personal subset of the output vector. In this way, each block could run completely in parallel once it received its data from the stream. A final reduce step would sum any overlapping subset output vectors and write back into the memory mapped true output vector. By allowing compute to be pipe-lined more effectively with reading from the stream, we would hopefully achieve much better memory and resource saturation. As mentioned before, slightly more regularly accessed sparse formats would aid in pipelining inner compute loops. In future, we would also like to explore a multiple stream implementation, allowing multiple blocks to be read in parallel.

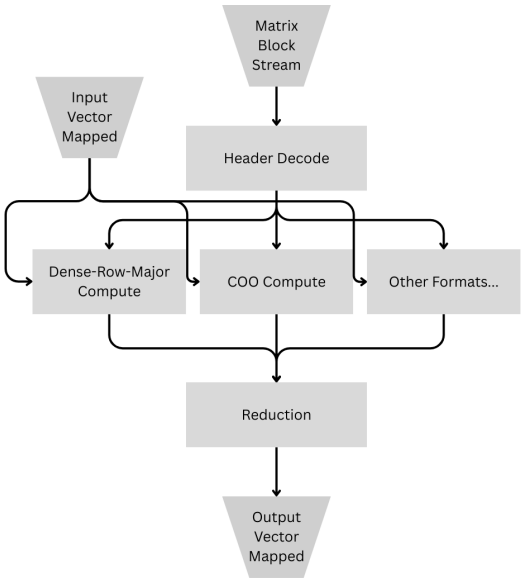

Fig. 3. Future Parallel Implementation

V. SUMMARY

In conclusion, we have motivated the need for new solutions to very irregular sparse matrices, as well as called attention to previous and older solutions. We proposed a novel streaming structure that might combine the strengths of previous formats, tailoring them to the needed block granularity on an individual matrix basis. In addition, we implemented such structure on FPGA to enable future work in parallelizing the heterogeneous computation and unpacking logic of many different formats. Our baseline implementation demonstrated a working computation in which blocks of disparate types were consumed, as well as demonstrated that memory footprint was kept small even with streaming header overhead. Further, we proposed future work involving usage of BRAM and reduction functional modules for parallelizing computation in hopes of further saturating memory bandwidth.

REFERENCES CITED

[1] G. Xiao, C. Yin, T. Zhou, X. Li, Y. Chen, and K. Li, "A Survey of Accelerating Parallel Sparse Linear Algebra," ACM Computing Surveys, vol. 56, no. 1, pp. 1–38, Jun. 2023, doi: https://doi.org/10.1145/3604606.

[2] D. Langr and Pavel Tvrdík, "Evaluation Criteria for Sparse Matrix Storage Formats," vol. 27, no. 2, pp. 428–440, Feb. 2016,

[3] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units," SIAM Journal on Scientific Computing, vol. 36, no. 5, pp. C401–C423, Jan. 2014

[4] S. Chen, J. Fang, C. Xu, and Z. Wang, "Adaptive Hybrid Storage Format for Sparse Matrix–Vector Multiplication on Multi-Core SIMD CPUs" *Applied Sciences*, vol. 12, no. 19, p. 9812, 2022.

[5] Z. Jiang, J. Jiang, J. Du, D. Huang, and Y. Lu, "Optimizing massively parallel sparse matrix computing on ARM many-core processor," Parallel Computing, vol. 117, pp. 103035–103035, Sep. 2023, doi: https://doi.org/10.1016/j.parco.2023.103035.

[6] M. Li, Y. Ao, and C. Yang, "Adaptive SpMV/SpMSpV on GPUs for Input Vectors of Varied Sparsity," IEEE Transactions on Parallel and Distributed Systems, pp. 1–1, Jan. 2020, doi: https://doi.org/10.1109/tpds.2020.3040150.

[7] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs," Journal of Parallel and Distributed Computing, vol. 76, pp. 3–15, Nov. 2014, doi: https://doi.org/10.1016/j.jpdc.2014.11.001.

[8] S. Li, Y. Wang, W. Wen, Y. Wang, Y. Chen, and H. Li, "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel," Nov. 2016, doi: https://doi.org/10.1145/2966986.2966987.

[9] L. Braun, S. Nikas, C. Song, V. Heuveline, and Holger Fröning, "A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels," ACM Transactions on Architecture and Code Optimization, vol. 18, no. 1, pp. 1–25, Dec. 2020, doi: https://doi.org/10.1145/3431731.

[10] M. Hosseinabady and J. Nunez-Yanez, "A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis," vol. 39, no. 6, pp. 1272–1285, Jun. 2020, doi: https://doi.org/10.1109/tcad.2019.2912923.

[11] Goran Flegar and Hartwig Anzt, "Overcoming Load Imbalance for Irregular Sparse Matrices," Zenodo (CERN European Organization for Nuclear Research), Nov. 2017,

[12] AMD Xilinx, "Vitis High-Level Synthesis User Guide (UG1399)," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls