Inspector Executor SpMV on FPGA

Ziheng Wang, MIT CSAIL, Shuotao Xu, MIT CSAIL

December 2019

1 Introduction

In this project, I design an accelerator to perform sparse matrix dense vector multiplication (SpMV). We assume that the sparse matrix is known at compile time, and we can transform its layout in host memory before loading it onto the accelerator. Such transformations are typically done to improve data locality or enhance parallelism [1, 2]. This can be thought of in terms of the inspector-executor framework [3]. In the inspector (software) stage, we inspect the sparse matrix structure and perform the transformations on the host, typically a general purpose processor. In the executor stage, the computation is executed on the accelerator hardware using the modified layout. The overhead paid by the inspector is amortized over multiple calls to the executor, which is common in applications such as neural net inference and iterative solvers.

In this report, we examine one specific transformation strategy of the sparse matrix. We present a full stack implementation of the inspector in software, and synthesize our hardware executor design on an FPGA. We compare the resulting hardware accelerator performance with optimized CPU and GPU implementations.

State of the art sparse linear algebra accelerators typically adopt a streaming approach. For example, in sparse matrix vector multiplication, all the sparse matrix values and indices are streamed onto multipliers on the accelerator. For each value/index pair, the corresponding dense input vector element is fetched to perform the multiplication. This is an input data-dependent load, whose latency cannot be hidden. This is recognized as the key challenge in designing sparse matrix vector accelerators [4, 5]. Current remedies typically involve replicating the dense input vector storage [5], which quickly runs into problems if the dense input vector is too large. In addition, this kind of design does not consider any kind of input reuse: if two value/index pairs actually request the same dense input vector value, that value is fetched in separate memory transactions.

In this work, we examine if our compile time transformations can improve the memory access patterns of the sparse matrix. In addition, we study if we can process the sparse matrix in tiles to potentially use hardened on-chip vector processing resources, such as the AI engines in the Xilinx Versal ACAP.

This project is a first step towards a larger goal to design a compiler stack that efficiently explores the design space of linear algebra accelerators through

transformations on the sparse matrix. In this work, the sequence of transformations is fixed, and the hardware design is fixed to target this particular sequence of transformations. However, future work will allow a user to specify an arbitrary sequence of supported transformations. The compiler stack should be able to transform this matrix in software, and generate the hardware required to execute the transformed computation.

2 Sparse Matrix Vector Multiplication

2.1 Basic Terminology

Matrix vector multiplication can be modeled as a map reduce. Let's imagine that the matrix values are stationary in a grid of registers. When a new dense vector is presented, its values are broadcast along each column of the matrix. A multiplication is performed at each matrix value in the map stage. If the matrix is sparse, then some of the multiplications may be skipped. In the reduce stage, all the multiplication products in the same row are reduced to the output. This is illustrated in Figure 1.

2.2 Inspector Transformations

The input to the system is a sparse matrix in a generic format such as COO or CSR. We will preprocess the sparse matrix with a series of transformations to obtain a format amenable for an optimized hardware implementation. This preprocessing stage will be implemented on a general purpose processor for two reasons: 1) adaptivity: different transformations could be quickly explored in software 2) performance insensitive: we can amortize the transformation cost over all subsequent operations involving this matrix.

There are several transformations commonly used on a sparse matrix to improve data locality, enhance parallelism or increase load balance. **In order to be able to carry out the original computation, the transformations need to be invertible.** [6] Some allowed transformations include:

- **Pack**: Packing all the nonzeros along one dimension. This transformation greatly improves data locality on vector or tile based processing units.
- **Split**: Splitting up the sparse matrix along a dimension. This transformation enables one to parallelize the computation over different processing units.
- **Reorder**: Reordering the sparse matrix along a dimension, given a permutation. The benefit of this transformation depends on the permutation function.

Note that all of these transformations are invertible, provided that certain bookkeeping information is saved. For example, if you pack a sparse matrix and save the original locations of the nonzeros, you can reconstruct the original

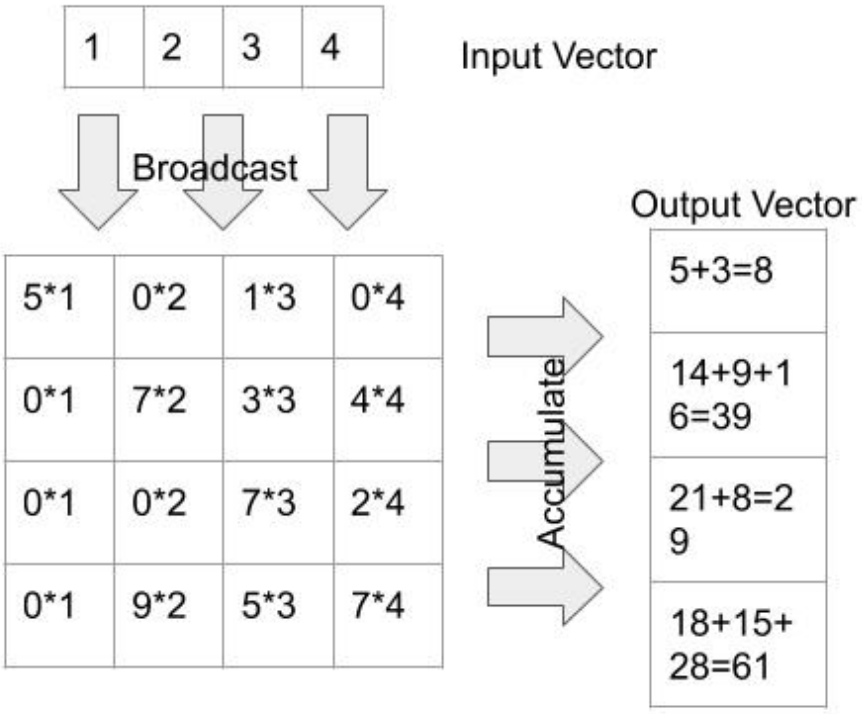

Figure 1: Matrix vector multiplication.

sparse matrix from the packed values and the saved indices. If you reorder the columns of a sparse matrix and save the permutation, you can reconstruct the original sparse matrix. These are auxiliary data structures that are produced from the transformations. They need to be loaded on to the accelerator as well.

In this work, we consider the following sequence of transformations, which I call a **schedule**, on a sparse matrix $A(i, j)$:

```
Tunable parameters C = 4, B = 4, H = 2, W = 2;
A.split(i,C); //Split up A in the i dimension to get
    C sparse matrices. A is now a vector of sparse
    matrices
A[i].split(j,B) for i in range(C); // Split each
    element in A into B sparse matrices, so A is now a
     2D (C by B) array of sparse matrices
A[i,j].reorder(i,nnz) for all i,j; // For each sparse
     matrix element in A, reorder the rows in that
    sparse matrix by the number of nonzeros
A[i,j].pack(j) for all i, j; // pack all the nonzeros
     in each sparse matrix element of A along the j
    dimension.
A[i,j].tile(size=(H,W)) for all i,j; // tile each
    sparse matrix into blocks of size 2 by 2
A[i,j].pad() for all i,j; // pad each tile
A[i,j].fix_order(i,j) for all i,j; // for each sparse
     matrix, iterate through the tiles in i major
    order (row major)
```

Let's elaborate on what just happened in these transformations. First, we will divide up the matrix. We will first cut horizontally, so that each horizontal stripe has the same number of nonzeros (except last one). We will then divide each horizontal stripe into vertical blocks such that each block has the same number of nonzeros (except last one). Each **block** is a sparse matrix by itself. It's evident that all the blocks resulting from this division will have roughly the same number of nonzeros. The overall SpMV can be decomposed into sub-problems for each block, as illustrated in Figure 2. Each block is responsible for the matrix values in that block, and perform a matrix vector multiplication with its portion of the input vector, writing to its portion of the output vector. This transformation is done to parallelize the computation in a load balanced way.

For example, if the original matrix A is 100 by 100 with input vector x and output vector y, and the block in question is A[34:50,23:45], then this block performs the matrix vector multiplication with input vector x[23:45] and matrix A[34:50,23:45]. It will produce a partial output which needs to be accumulated to y[34:50]. There are other blocks which will also accumulate to y[34:50]. Note that we cut first horizontally, ensuring that the partial outputs are "lined up". The price we pay is that the input reads are not lined up across vertically adjacent blocks.

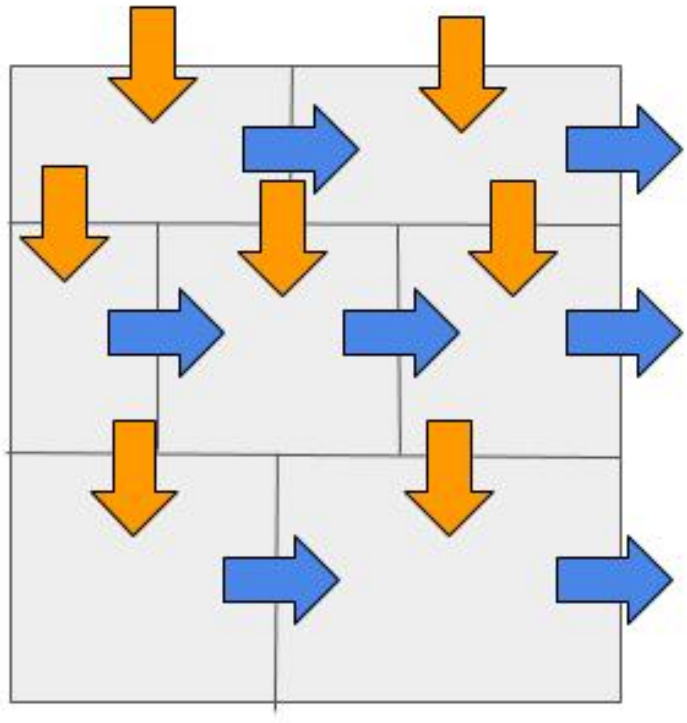

Figure 2: Matrix vector multiplication decomposed to subproblems in each block. The orange arrows denote broadcast from the corresponding portions of the input vector and the blue arows denote reduction into the corresponding portions of the output vector. In the case where multiple blue arrows reduce to the same portion, there exists possibility for output races.

We have divided the original sparse matrix into a 2D array of smaller sparse matrix blocks. Now let's transform each block. The sequence of transformations on each block is presented in Figure 3. First, we will sort the rows based on the number of nonzeros. We will record this permutation, per the invertibility requirement. Then we will "pack" the nonzeros in each row to the left. This will produce two data structures, one corresponding to the nonzero values in the original matrix and another necessary to keep track of the column index of the nonzero value. Note that we do not need to keep track of the row index since pack does not change the row index of a nonzero value. The column index is necessary for fetching the appropriate dense vector value for multiplication. For example, for the value 7 in the top left corner, we need to keep track of its column index 2 since we need to multiply 7 with the second element in the dense vector.

Finally, we will tile the packed nonzeros. The tile size is a tunable parameter. We will pad the empty positions in the tiles. We will transform these data structures to a serialized stream to populate the DRAM of the accelerator. We specified in our schedule that we will iterate through the tiles in row-major order.

These transformations on a sparse matrix block brings several benefits. First, we are iterating over tiles of nonzero values, which allow us to use vector processing units. Since we are iterating over tiles, the sorting and packing greatly increases the density of each tile, reducing the total number of iterations. Now if different positions inside the same tile share the same dense vector BRAM access index, such as the two 1s in the blue tile in Figure 3, then it can only be loaded once from memory and replicated locally, partially addressing the dense vector BRAM access challenge.

These transformations are associated with their own parameters, such as split factor and tile size, which all can be tuned. Our choices are listed above: we split into a 4 by 4 grid, with a tile size of 2 by 2.

These transformations have been fully implemented in Python and tested with a few sample test cases. The performance is acceptable for relatively small input matrices, but might run into problems if the matrix gets too large. This mostly has to do with the way I am storing the sparse matrix as a dense 2D array with 0 values: we'll quickly run out of memory for really big matrices. The current software implementation has only been tested with this particular schedule, though for future work, it can be extended to other optimization schedules.

2.3 Executor: Functional Specification

The hardware design almost immediately follows from the sparse matrix transformation. The overall design consists of a grid of processing elements (PE), one for each block. The processing element is a module a grid of multipliers with the same size as the tile size 8. This PE will iterate through all the tiles and perform the matrix vector multiplication for this subproblem. We have fixed the order of iteration in the schedule. The PE starts at the top left, slides to the right, and then slides down when it can't slide right anymore. (So the order

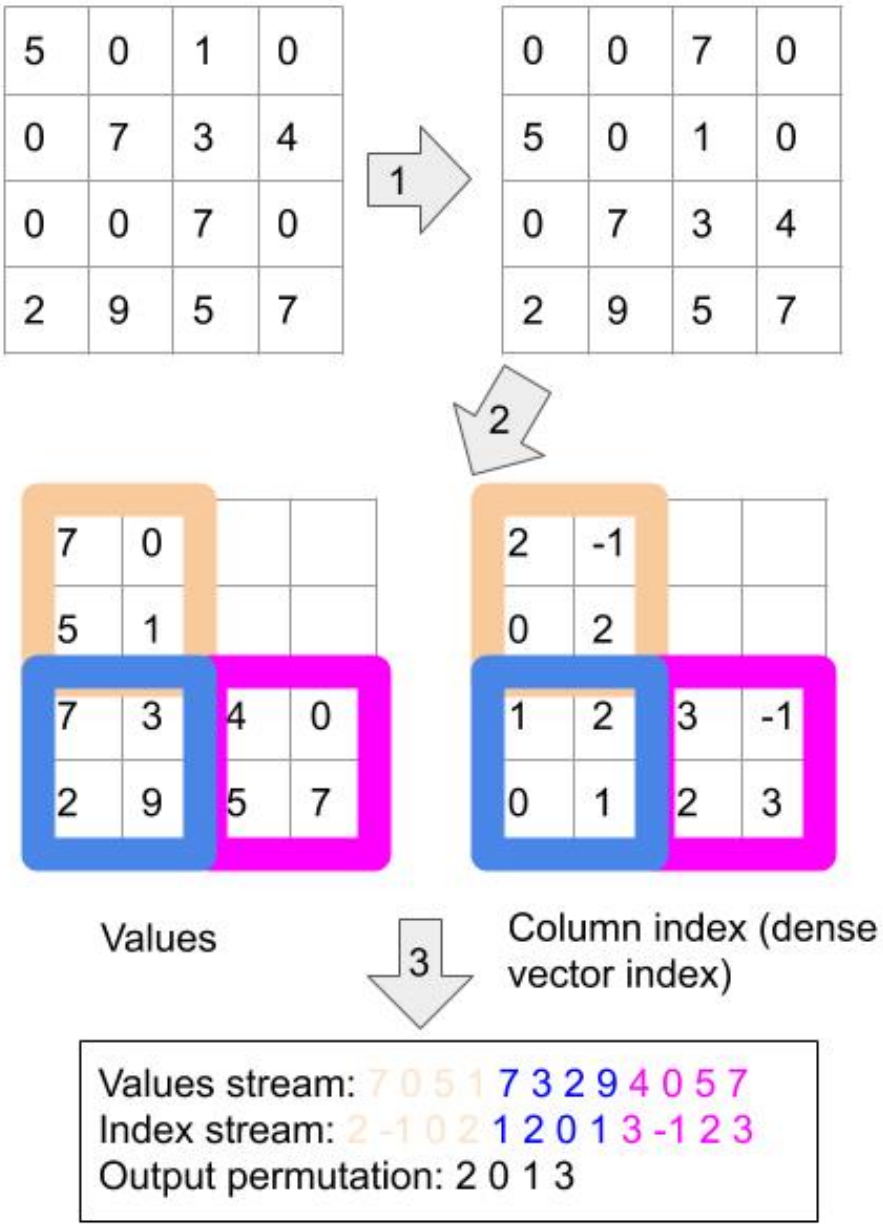

Figure 3: Transformations of the sparse matrix in a block.

in the example in Fig 3 would be pink, blue, magenta.) Different PEs in the same horizontal stripe (see Figure 2) share the same output buffer, which they accumulate to atomically.

At each position in the iteration, the PE loads in the appropriate sparse matrix values and the corresponding dense vector values (determined by the sparse matrix indices data structure we stored from the pack transformation), multiplies them element-wise, and accumulates the result. Note that as aforementioned, if there are repeated indices in the tile, then we need to make fewer requests to the dense vector memory, exploiting input reuse. In practice, we find that many real sparse matrices offer high potential for reuse in this sense. If the reuse is poor, we can permute the rows of the original sparse matrix so that rows with nonzeros in the same column index can be next to each other, using some other inspector transformation.

Different PEs operating on the same horizontal band in the sparse matrix write to the same portion of the output vector. In addition, their rows might be permuted in different ways internally, complicating the problem even further. When a PE of size 2 by 2 slides all the way to the left, it will produce intermediate outputs for 2 indices of the output vector portion, which must be accumulated to the output vector. We need to look in our saved permutation order to see what output vector indices these 2 indices correspond to. This is a sparse scatter operation. Different PEs might perform this sparse scatter at the same time, and some of the scatter targets might conflict. Since this is not a bottleneck of the system, the current implementation simply locks the entire buffer when a particular PE is doing a scatter operation.

3 Exectuor: Microarchitecture

Now that I have described the architecture's functional specifications, let's discuss the microarchitecture. The top level module diagram is shown in Figure 4. The top level module contains 4 Column of Controllers (CoC) modules as well as the input vector BRAM. Each CoC module is responsible for one entire horizontal band in Figure 2. Each Column of Controller module contains 4 PE Controller modules as well as its own matrix values/index BRAM, permutation index BRAM and output buffer BRAM. The matrix values/index BRAM contains all the sparse matrix values and indices that PEs in this column are going to need. The permutation index BRAM stores information required to direct the outputs of each PE to the correct address in the output buffer, since we might locally permute the rows in the sparse matrix block the PE is processing. Each PE Controller module contains a PE module, a memory controller module and a lengths BRAM. The lengths BRAM stores some bookkeeping information on the shape of each packed block, so the PE knows when to slide right and when to slide down. The memory controller interfaces with the shared input dense vector BRAM to process memory requests for this PE.

The CoC module is designed as a FSM with several rules, as shown in Figure 5. It always tries to prefetch from the permutation BRAM and the matrix

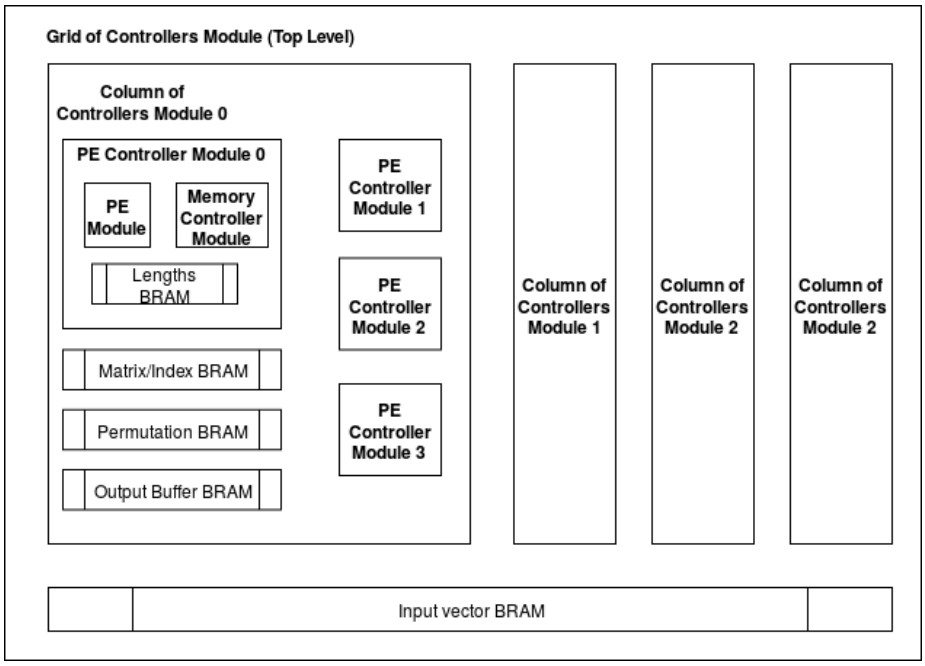

Figure 4: Architecture

values/index BRAM to enqueue into the appropriate FIFOs and registers to feed each PE controller, which is responsible for controlling the PE and producing the correct results. The FIFO sizes are parameterizable. If the FIFOs are larger we use more resources, but the prefetching system's latency can be better hidden. The PE Controller produces results for a part of the output at at time. Whenever a new result is produced by a PE Controller, the CoC atomically scatters it into the output buffer, using the prefetched permutation indices.

Within a PE Controller module (Figure 6), we try to dequeue from the sparse matrix values/indices FIFO. The sparse matrix indices are used to feed the Memory Controller module, which interfaces with the shared dense vector BRAM to produce the dense vector values. Once these values are available they are fed to the PE alongside with the sparse matrix values to produce the local computation result. We then move the PE to the right. After it has depleted all the positions in its current band of rows, it moves down and starts again from the leftmost position. It enqueues the intermediate accumulation output for this row band into the output FIFO, which is then scattered into the output buffer with the appropriate permutation indices in the CoC module. We need the lengths BRAM to store how many positions are there in each row band. In practice, we only need to load from this BRAM once for every row band, and its access pattern is completely predictable. Its access time is completely hidden.

The Memory Controller module (Figure 7) is by far the most complicated

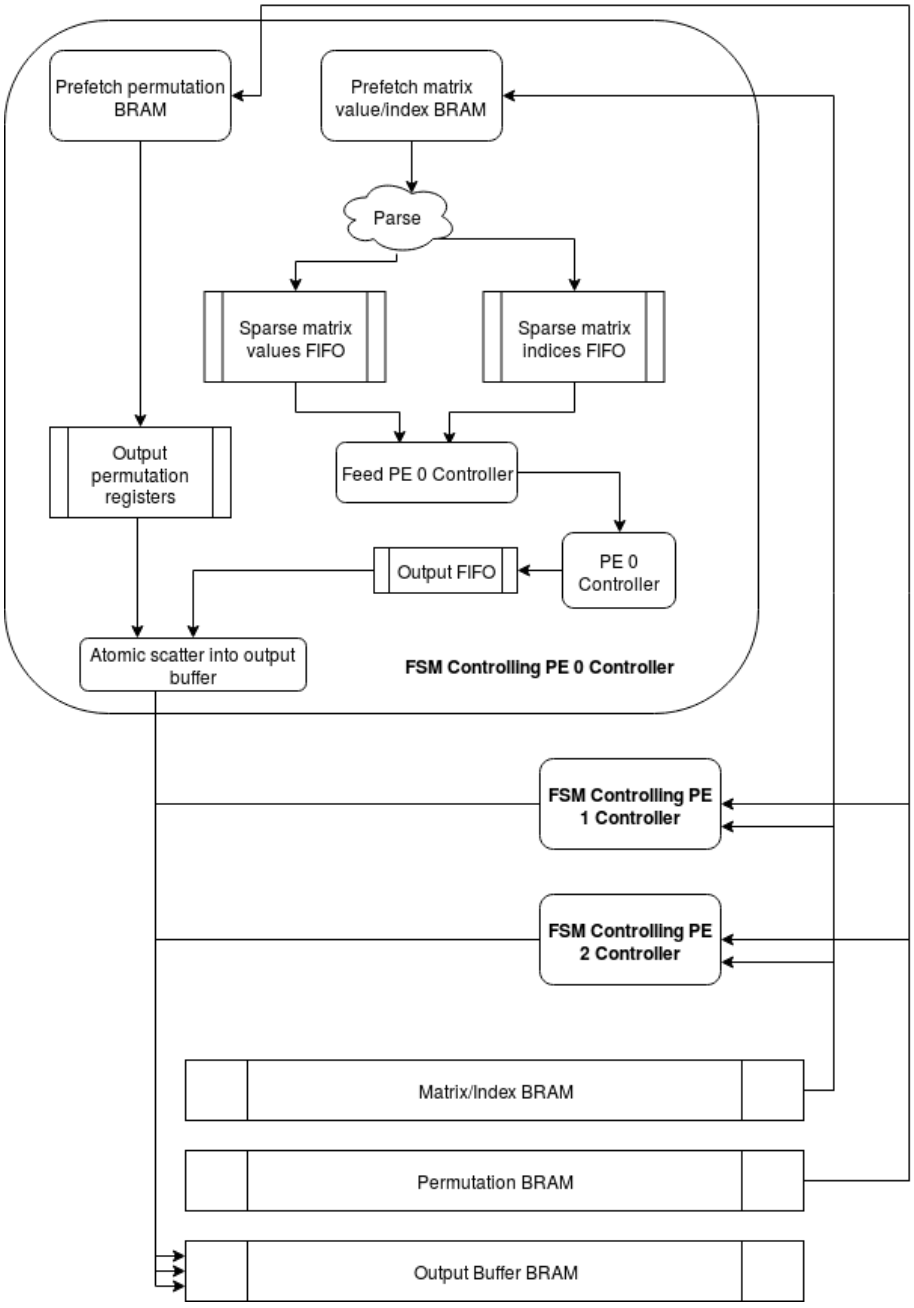

Figure 5: CoC module FSM. Arrows indicate dependence relationships. (Guards in BSV)

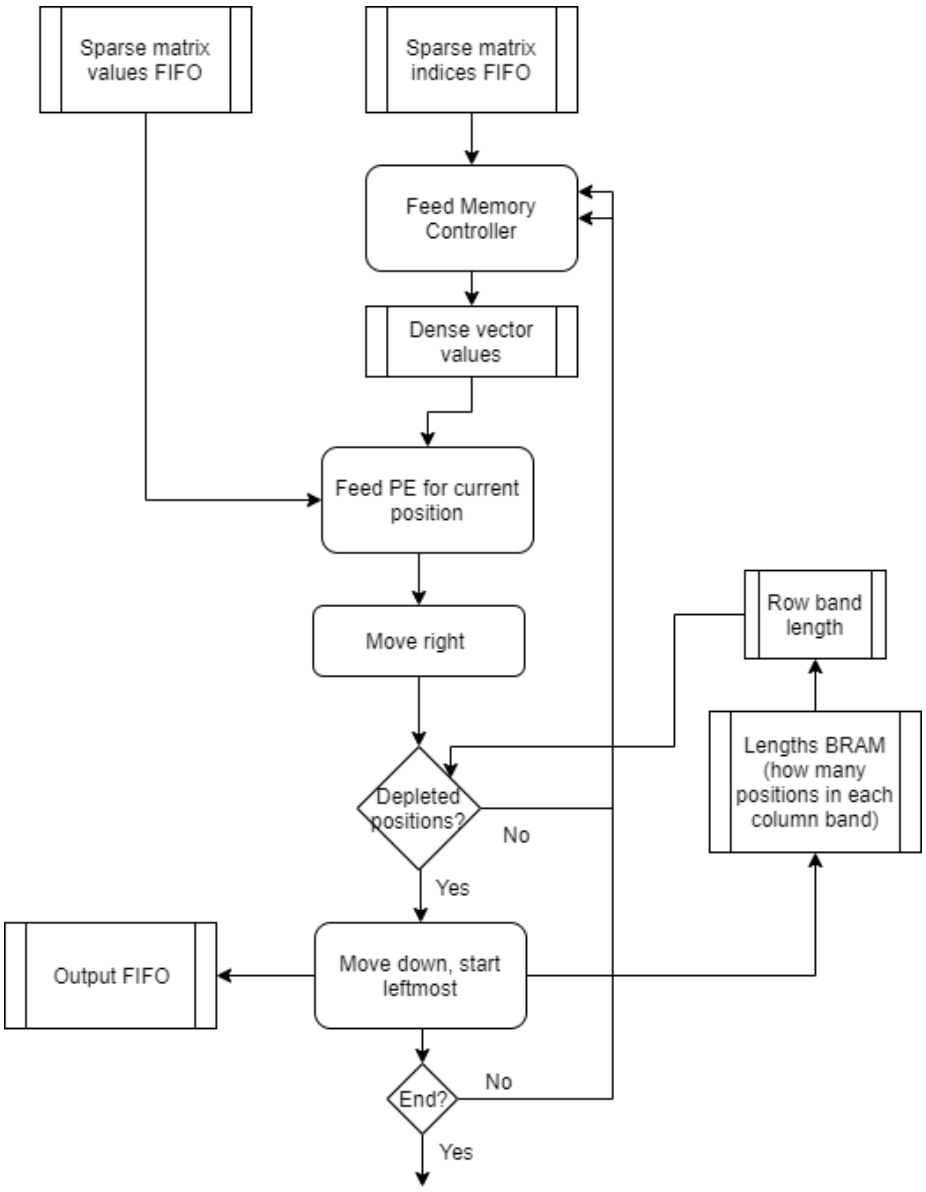

Figure 6: PE Controller Module FSM.Arrows indicate dependence relationships. (Guards in BSV)

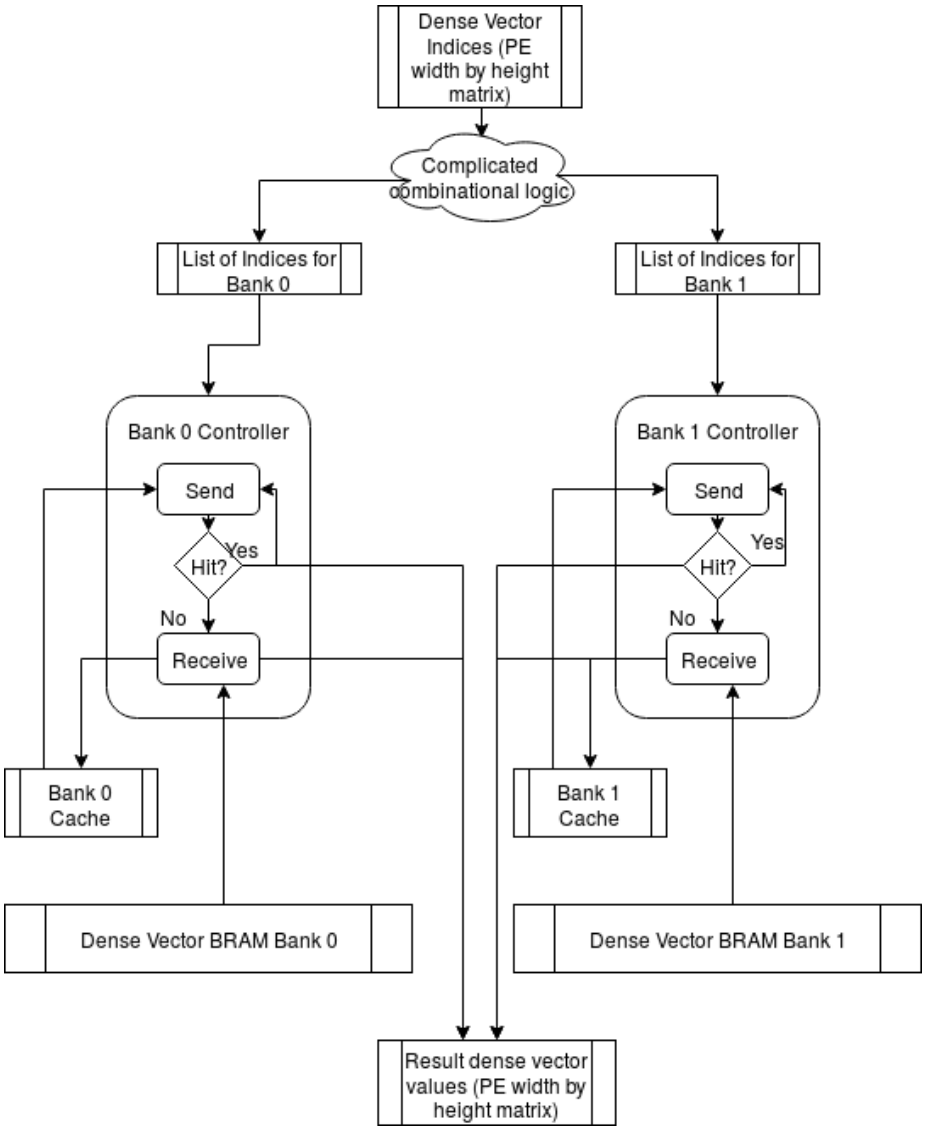

Figure 7: Architecture of the memory controller unit. It consists of one FSM to control requests to each bank of the shared BRAM. Each FSM has its own caching system. The combinational logic that distributes work to the FSMs is very complicated and is likely the bottleneck of the system.

of all the modules. Each PE has its associated Memory Controller module. It is responsible for converting the sparse matrix indices in a PE position to the appropriate input dense vector values. All the PEs in the system share access to a single dense vector BRAM structure. In our design, this BRAM is chosen to be 4-way banked single port. Each Memory Controller module has four Bank FSMs, each controlling accesses to their corresponding bank. When a sparse matrix indices request comes in, it is translated through complicated combinational logic to work assignments for each Bank FSM. Each Bank FSM then iterates through its assignment.

To handle potential reuse inside of a tile, we implement a 4-way fully associative cache inside each bank FSM. The FA cache is implemented as a circular buffer, which effectively uses a LRU eviction policy. If the request is in cache, the bank FSM uses the cached result and avoids the trip to the BRAM. This appears to only save one cycle, but also greatly reduces contention on the shared BRAM resource, saving many more cycles. If it's a cache miss, then the Bank FSM sends the request to BRAM. It will write the response to the cache as well as the result. When all the Bank FSMs are done, the Memory Controller module produces the final result dense vector values for this PE position.

The PE itself is actually very simple. It's just a grid of multipliers. In BSV, it's implemented right now with two mutually exclusive rules: fill-compute and flush. The fill rule-compute fills the multipliers with the input vector values and the input matrix values, then does the multiplication at each location in the grid. The flush rule adds multiplier outputs vertically through either cascade adder or tree adder and then produces the output. This effectively performs a 2 by 2 matrix vector multiplication in two cycles. We can also implement this as a systolic array with higher throughput, a potential future optimization.

4 Synthesis and Simulation Results

The design was compiled and synthesized successfully on the VC707 evaluation board for a 4 by 4 grid of 2 by 2 PEs, with a 4-way banked dense vector BRAM. The FA cache size was chosen to be 4. The cycle count on FPGA matches the cycle count in Bluesim, suggesting that to evaluate the performance of different designs, we can just use Bluesim (assuming same clock rate).

We notice a disproportionately high usage of LUTs to DSPs. Indeed, we only use 10 percent of the DSPs on the chip with 33 percent of the LUTs and FFs. This limits the scaling of the design to effectively use all of the hardened computational resources on chip. This is most likely because of the over complicated memory controller logic, which is a target of future optimizations. This high logic usage also slows down the max clock rate of the design to 50 Mhz. (The critical path is not in my design logic but the system clock.) The BRAM usage depends on the input matrix, which is less than 5 percent for a small 1K by 1K matrix with around 10K nonzeros and is projected to scale linearly with the number of nonzeros in the input matrix.

There are more severe challenges in testing larger designs, even in simulation.

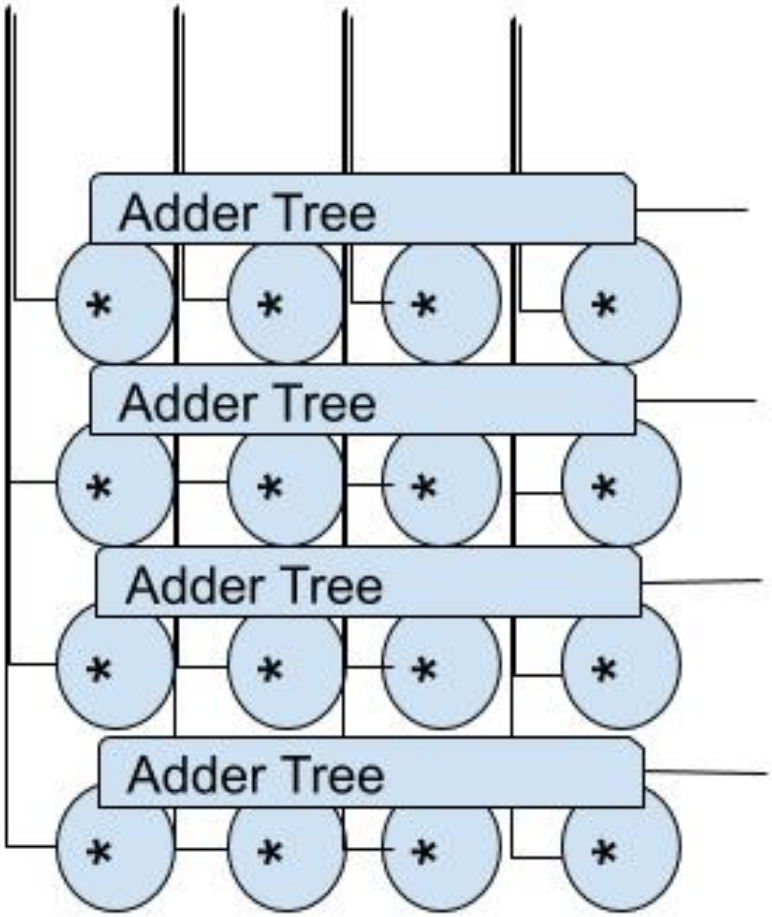

Figure 8: PE Microarchitecture

	Application	Dimension	Nonzeros	Density
Rajat04	Circuit Sim	1041	8725	8.1e-3
Garon2	CFD	13535	373235	2.0e-3
ffR_11	Optimal Control	5438	40054	1.4e-3
Circuit_03	Circuit Sim	12127	48137	3.3e-4

	cuSPARSE V100-SXM2	MKL 8 Cores Intel X7550	FPGA Cycles	FPGA Time (50Mhz)
Rajat04	43.6 us	8 us	5091	101.8us
Garon2	7us	110 us	118282	2365.6us
ffR_11	60.6 us	19 us	16608	332.2us
Circuit_03	149 us	31 us	20570	411.4us

Figure 9: Performance comparisons between FPGA and GPU for four sparse matrices with different sizes and densities.

This is because the memory controller combinational logic uses a lot of for loops and other constructs which are poorly handled by the Bluespec compiler. As a result, this severely limited the extent of design exploration I was able to do.

5 Performance Characteristics

We evaluate the performance of our BRAM-based architecture on four sparse matrices. The statistics of these sparse matrices and the performance of our FPGA vs baseline GPU and CPU implementations are shown in Figure 9 and Figure 10.

We run GPU benchmarking results on a V100 with 15.7 TFLOPs and 900Gb/s memory bandwidth. This GPU typically costs around 10,000 dollars with TDP 300 W. The peak clock speed is nearly 1.5 Ghz. The CPU implementation runs on 8 cores with AVX-512, for a total of theoretical peak 64 multiplications per cycle at 2.00 Ghz, which is 128 GFLOPs.

In comparison, the hardware design in this project has only 64 multipliers. The design was synthesized with a clock speed of 50 Mhz, which translates to only 3.2 GFLOPs. (about 5000 times slower than the V100 and 40 times slower than the CPU). The performance numbers are roughly in line with those GFLOPs, as shown in Figure 10. We are about an order of magnitude slower than the CPU implementation for all matrices, and our runtime scales with the number of nonzeros. For the GPU, the runtime is not dictated by the number of nonzeros but by matrix specific access patterns. Garon2 has large dense blocks which greatly reduce runtime due to favorable L1 caching characteristics.

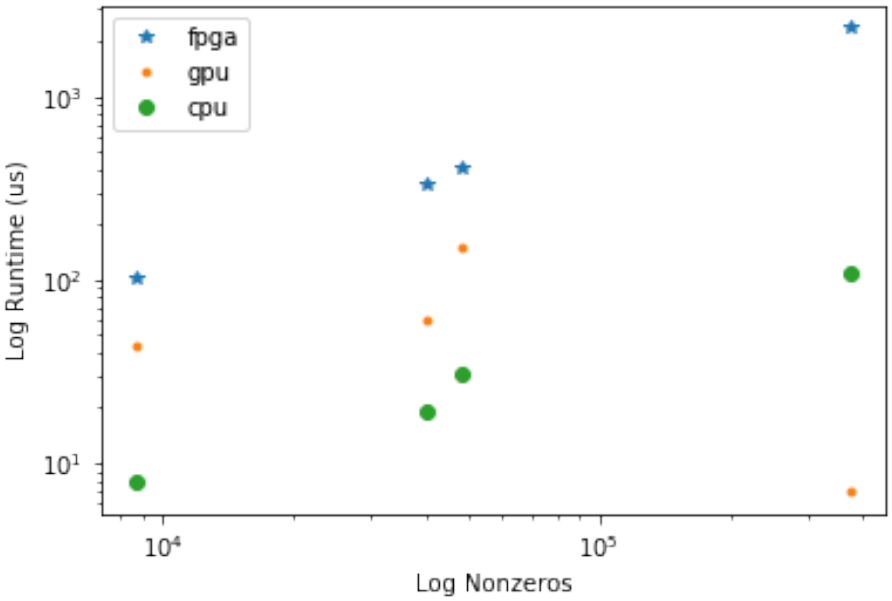

Figure 10: Performance scaling with number of nonzeros for FPGA and GPU.

From the start, we know that our design will be bottlenecked by dense vector BRAM bandwidth. We present some results in Figure 11 to confirm this hypothesis. In the figure, the x axis is 100 cycles. First, we try a baseline implementation with a shared single port dense vector BRAM without any banking or local caching. The theoretical peak number of BRAM responses is 100. We see that the dense vector BRAM utilization fluctuates between 40 to 80 percent, and dominates all memory traffic. When we add 4-way banking, the theoretical peak number of BRAM responses is 200, as in my implementation this introduces a conflict between the send request rule and the get response rule, so they can no longer be concurrent. (100 times 4 divide 2 = 200) We see that the BRAM utilization improves dramatically, and halves the total number of cycles. When we add in local caching, the total number of BRAM requests decrease, further decreasing the number of cycles by 20 percent. Figure 11d compares the BRAM traffic under those three schemes. The banking improves utilization but does not decrease the number of total requests. The caching keeps utilization high but decreases requests.

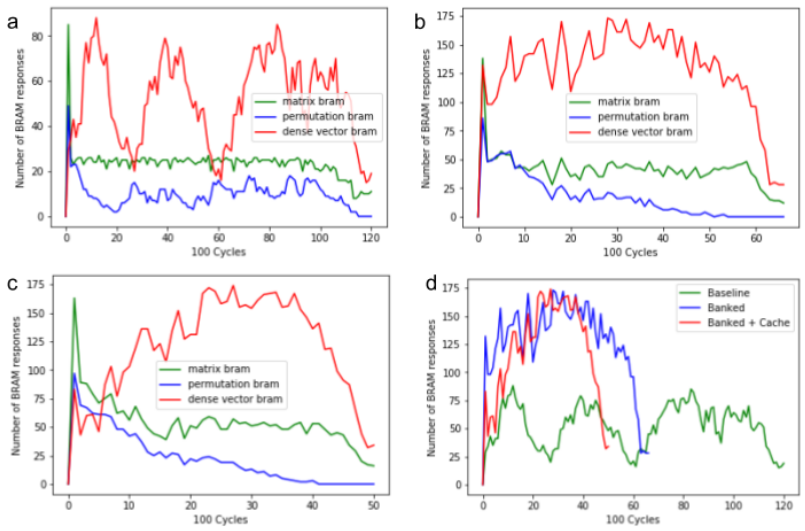

Figure 11: BRAM traffic for matrix values/index BRAM, permutation BRAM and dense vector BRAM for a) Baseline design with single port shared dense vector BRAM for all PEs b) 4-way banked dense vector BRAM c) 4-way banked dense vector BRAM with 4-way FA cache for each bank for each PE. d) A comparison of dense vector BRAM traffic achieved for these three designs.

6 Future Work

6.1 Dense vector BRAM replication

We observe that further increasing the number of PEs does not lead to greatly improved performance. Indeed, if we don't change our dense vector memory configuration, and just scale the number of PEs by 4, we only increase performance by about 20 percent. This is because we have already saturated the bandwidth to the dense vector BRAM, as indicated in Figure 12. Even though we have four times as many PEs, the number of BRAM responses do not significantly increase. To alleviate this problem, we can add more banks to the dense vector BRAM, but this required significant logic area overhead due to the complicated bank control logic. We can also adopt the prevalent approach in literature and replicate the dense vector BRAM. This is acceptable when the matrix is small and there is sufficient on chip BRAM resources. We plan to add the replication factor as a tunable parameter of the design and explore this option in the future.

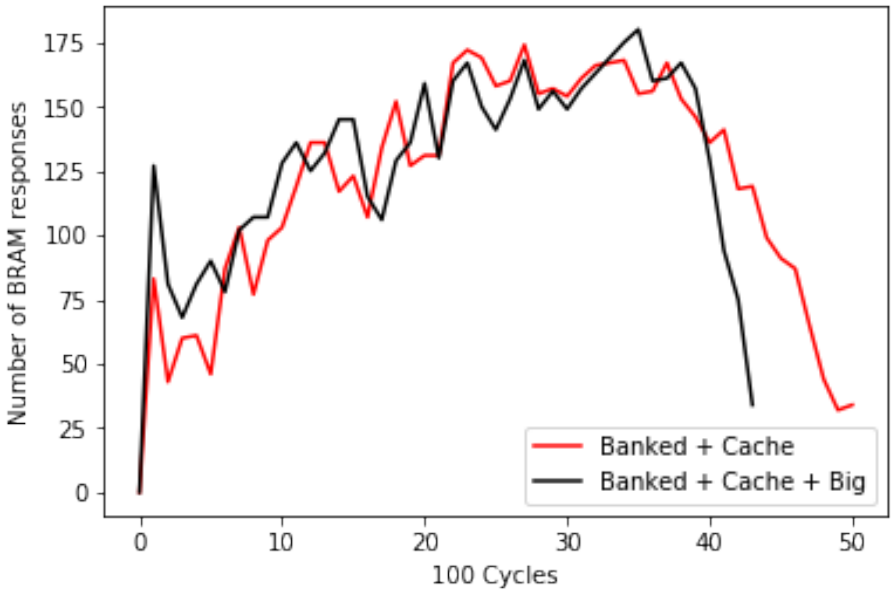

Figure 12: Poor scaling with increased number of PEs. Dense vector BRAM response throughput fails to scale with more PEs.

6.2 Restructuring sparse matrix/index BRAM

Currently, a line of the sparse matrix/index BRAM contains all the sparse matrix values and indices necessary for a single PE computation (4 matrix values and 4 indices). We can increase the line width further to include multiple PE positions to lower the amount of requests made to the sparse matrix/index BRAMs. In addition, currently, we allocate one single port BRAM for each column of PEs. We could allocate a separate BRAM for each PE with no duplication of memory, since we know each PE will access different parts of this shared BRAM. In short, since sparse matrix/index access pattern is static, it can be greatly optimized. These further optimizations were not explored since they are not the bottleneck of the design.

6.3 Optimization of dense vector bank control

The logic for each PE to correspond with the dense vector BRAM is quite complicated, as explained above and outlined in Figure 7. This logic could be simplified in different ways to accelerate Bluespec compilation and reduce eventual logic area, which could also result in a higher operating frequency. I experimented with a new implementation together with Shuotao. It was not able to significantly accelerate compile time, though it could be synthesized to operate at a higher frequency (66 Mhz vs. 50 Mhz now). What we really need

to do is to add synthesize boundaries around the memory controller modules, which is not currently possible since we are passing the dense vector BRAM interface to every instance of the memory controller modules.

6.4 Making dense vector BRAM push based

Since the matrix sparsity pattern is known in advance, we know which PEs are going to need which dense vector values when. We can implement a push based dense vector BRAM. We thus no longer need to make requests. Kudos to Shuotao for this idea. I will work on this.

6.5 DRAM

I codesigned a DRAM interface for the accelerator together with Josh Noel. For large matrices we might not be able to store the sparse matrix values/indices in the BRAM. The design is shown in Figure 13. In DRAM, each PE will have assigned to it a continuous chunk of addresses. The lines will just contain the tile data in compressed format. The DRAM line boundary is meaningless, it's just all chunks of data. Locally, for each PE, we will keep a circular buffer of a parameterizable size. The circular buffer keeps track of the lines we have fetched in the DRAM. There will be a rule that looks at that circular buffer, takes off data corresponding to a tile and frees up space. This interfaces directly with the PE controller module. Each PE will have a rule that makes request to the DRAM to fill up this circular buffer. This means that BSV will impose an internal ordering on those requests. Whenever a PE gets backpressured from its circular buffer, the PEs after it can go. It's not really fair but should work in practice. In Figure 13, blue arrows represent data movement and red arrows represent backpressure. We are able to set it up through Connectal and have our DRAM stack give results. They are just not correct yet.

7 Acknowledgement

Firstly, I thank Shuotao for all his help in this project. Secondly, I thank Josh Noel for all his helpful discussions and collaboration on the DRAM interface. I also thank all course staff and Arvind for a great course. I definitely learned a lot.

References

[1] Sicheng Li, Yandan Wang, Wujie Wen, Yu Wang, Yiran Chen, and Hai Li. A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2016.

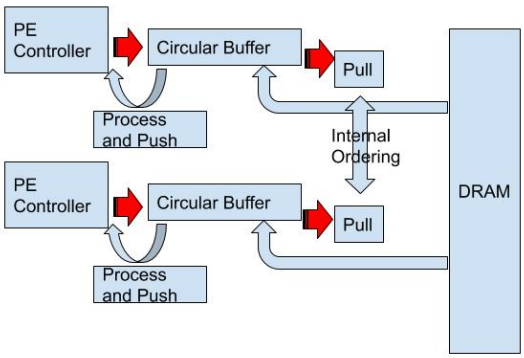

Figure 13: Proposed DRAM Interface.

[2] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[3] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[4] Yaman Umuroglu and Magnus Jahre. Random access schemes for efficient fpga spmv acceleration. *Microprocessors and Microsystems*, 47:321–332, 2016.

[5] Paul Grigoras, Pavel Burovskiy, and Wayne Luk. Cask-open-source custom architectures for sparse kernels. 2016.

[6] Hongbo Rong. Expressing sparse matrix computations for productive performance on spatial architectures. *arXiv preprint arXiv:1810.07517*, 2018.