Evaluating SpTRSV Design Performance for FPGAs Final Report

Dilshan Kumarathunga
dst5@sfu.ca

1. Introduction

Linear systems on sparse matrices are widely used in many scientific and engineering applications, such as circuit simulation, power system analysis, network graph analysis, computational fluid dynamics, and reservoir simulation. Among them, solving a set of linear equations associated with a sparse triangular matrix comes up as a very crucial computation. Hence, the Sparse Triangular Solver (SpTRSV) is one of the core components of direct and iterative linear system solvers [1].

Compared to other sparse matrix computations, SpTRSV is inherently sequential and has varying parallelism based on the matrix. Therefore, it is challenging to accelerate this kernel, and it has become a major bottleneck in certain computations. Analyzing and improving its efficiency can greatly help improve scientific and engineering computations.

Being reconfigurable and low-power devices, Field Programmable Gate Arrays (FPGAs) are becoming one of the most sought-after accelerator options in compute servers today. It can be observed in the literature that FPGAs are used to accelerate linear algebra kernels like LU Decomposition (LUD), Matrix-Vector multiplication (MV), Sparse Matrix-Vector multiplication (SpMV) and Sparse Matrix-Matrix multiplication (SpMM). Therefore, the main objective of this study is to explore the available parallelism in Sparse triangular solver for FPGAs and, build an analytical model to assess the runtime that can be achieved by implementing a SpTRSV architecture on FPGAs.

This project will be carried out in SFU's HiAccel Lab.

2. Background and Motivation

2.1 Triangular Solver

The basic idea of a triangular solver is found under the terms of “forward substitution” and “backward substitution” in the literature. The process of solving a set of linear algebraic equations in the form of Lx=y, where L is a lower triangular matrix is called forward substitution.

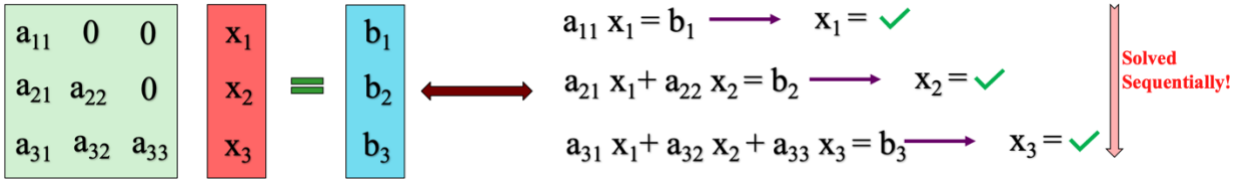


Fig. 2.1: Forward Substitution

Similarly, if the triangle is an upper triangle matrix, it is called "backward substitution". All the algorithms for the triangle system can be applied to both upper triangular matrix types, hence hereafter, we only discuss about forward substitution as our triangular solver.

A general equation for the solution of x_i in the solution vector can be written as follows.

$$x_i = \frac{(b_i - \sum_{j=0}^{i-1} a_{ij} x_j)}{a_{ii}}$$

This equation can be broken down into two major parts.

1) Processing non diagonal values: In this part, we calculate the multiplication and accumulation part of the computation related to non-diagonal elements of the row. i.e.,

$$s = \sum_{j=0}^{i-1} a_{ij} x_j$$

2) Processing the diagonal value: In this part, we take the accumulated sum from the previous step and subtract it from the respective dense vector value b_i. Then we divide the results by the diagonal value and generate a new x_i. i.e,

$$x_i = \frac{(b_i - s)}{a_{ii}}$$

2.2 Sparsity and Parallelism

2.2.1 General Introduction to the Sparsity

When the lower triangular matrix of the linear equation system is sparse, the solving process is denoted as "Sparse Triangular Solver (SpTRSV)". As seen in section 2.1, even though the triangular solving is done sequentially for a dense matrix, when the matrix is sparse, there is an opportunity for parallelism.

2.2.2 Row processing parallelism

For example, if we consider the sparse triangular matrix shown in Fig. 2.2, parallelism can be observed as follows.

- 1st set of row numbers that can be solved in parallel: 0, 1, 2, 4
- 2nd set of row numbers that can be solved in parallel after solving the 1st set: 3, 7
- 3rd set of row numbers that can be solved after 1st and 2nd sets: 5
- 4th set of row numbers that can be solved after 1st, 2nd, and 3rd sets: 6

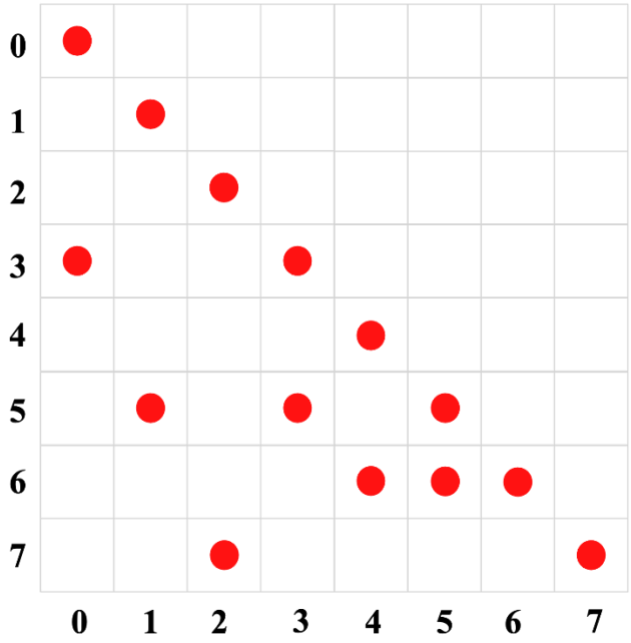


Fig. 2.2: A Sparse Triangular Matrix

Therefore, as seen in the example when the matrix is sparse, we can have some parallelism in processing rows as some rows can be processed in parallel. However, except for the 1st set of rows, this parallelism comes with a constraint. That is all the previous rows that have dependencies must be solved before processing the next set of rows. These dependencies are usually illustrated using a Directed Acyclic Graph (DAG) in the literature and used in some triangular solving algorithms as described in later sections.

2.2.3 Parallelism within a row

Apart from the row parallelism, we can explore a different parallelism within a row for the triangular solvers as well. That is when all the dependencies of a particular row are solved, we can perform the multiplication and accumulation part of the computations related to non-diagonal values in parallel. For example, in the above sparse matrix, we can perform multiplication and accumulation for non-diagonal values in row 5 while solving for that row. However, this parallelism also varies based on the row it processes as it depends on the number of non-zero values in the row. Another challenge is that, when processing a large sparse matrix, the non-zero values can be located far apart from each other, and hence, cannot assume they lie in consecutive locations.

2.2.2 Directed Acyclic Graph

As mentioned in section 2.2.1, the Directed Acyclic Graph can be used to visualize the dependencies of the rows of the triangle. This representation is very important in triangular solver studies as some algorithms are completely based on this and the performance of certain algorithms is based on the characteristics of this diagram.

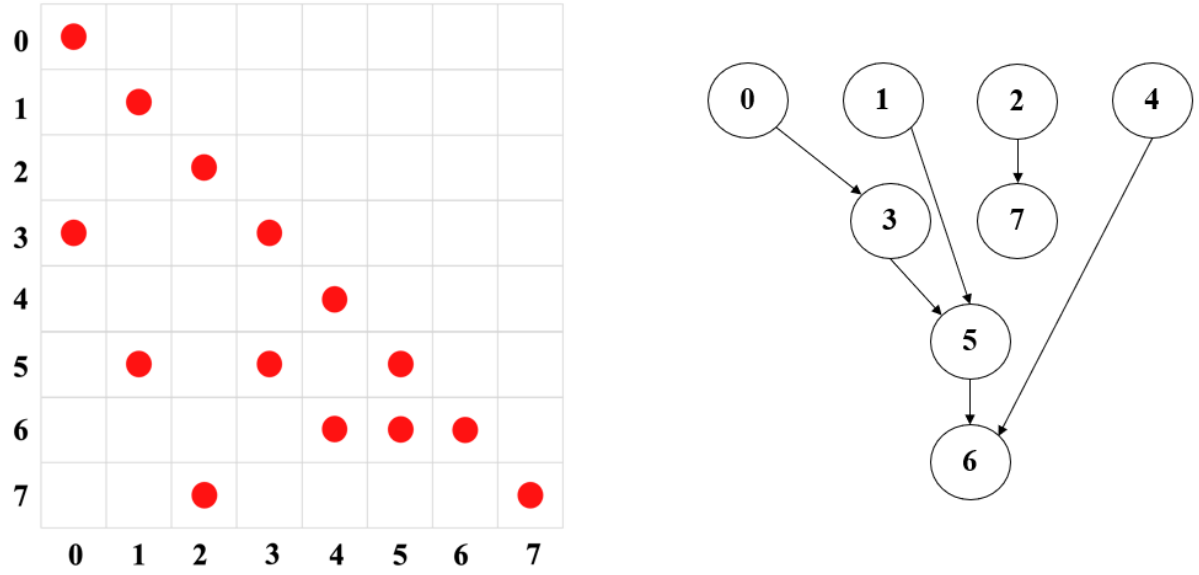


Fig. 2.3: A Sparse Triangular matrix and respective Directed Acyclic Graph (DAG)

Fig. 2.3. shows a sparse triangular matrix and its Directed Acyclic Graph (DAG). Each circle in the graph represents a row (or a diagonal value) and each arrow represents a dependency (or a non-diagonal value).

3. Classical Algorithms

There are two major triangular solver algorithms found in the literature.

1. Level set algorithm [2]
2. Synchronous free algorithm [3]

3.1 Level Set Algorithm

The level set algorithm depends on the DAG of the sparse matrix. This algorithm defines a level as a set of rows that can be processed in parallel and splits the DAG into levels. The rows in a level do not have any interdependencies as shown in Fig. 3.1.

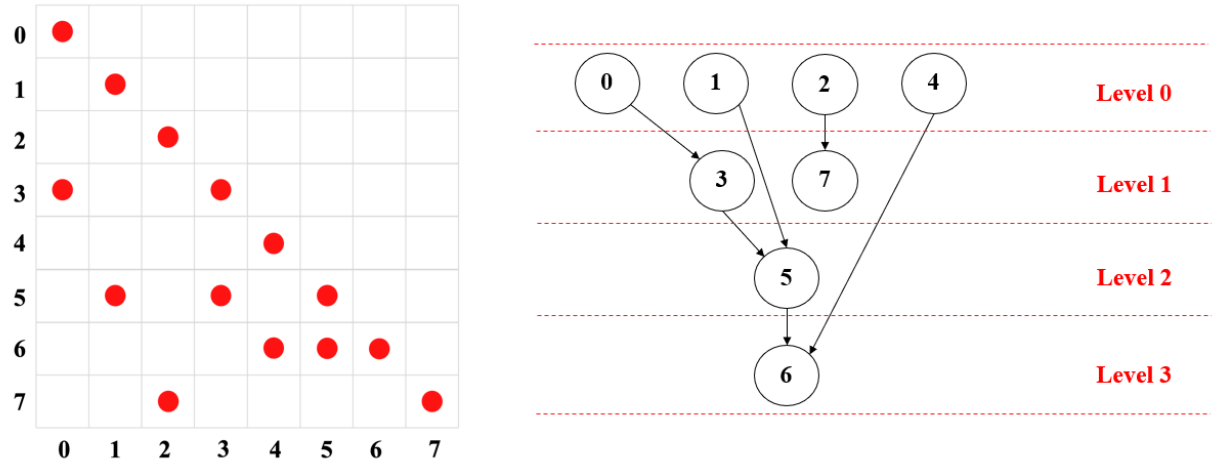


Fig. 3.1: A Sparse Triangular matrix and Directed Acyclic Graph (DAG) with Levels

The basic steps of the Level set algorithm can be summarized in the following steps.

1. Calculate DAG and Levels
2. For a level in the Level Set
    a) Solve the rows in the level in parallel
    b) Synchronize the barrier to make sure all the rows in the level are solved and updated
3. Return the results

Therefore, the level set algorithm utilizes the parallelism of rows in a level where there are no dependencies. However, if the sparse matrix has a higher number of levels, the total synchronization cost can outweigh the performance gain achieved by the parallelism and hence this algorithm performs poorly. For such cases, the synchronous free algorithm can perform better.

3.2 Synchronous Free Algorithm

This algorithm can perform better for the cases where the sparse matrix has a higher number of levels (i.e., more dependencies). It is majorly proposed for hardware platforms with multiple threads and a shared memory. Therefore, this algorithm is popular among the Graphics Processing Unit (GPU) community.

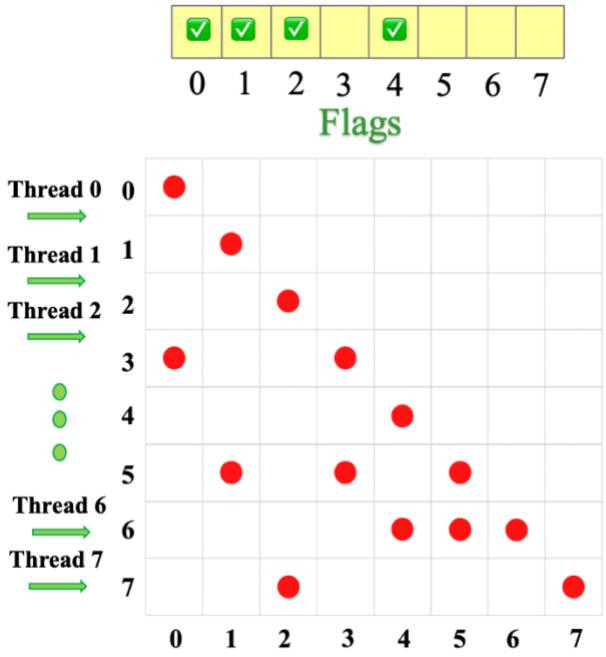


Fig. 3.2: Synchronous Free Algorithm

For this algorithm, a thread is deployed for each row and the thread will be working on solving the complete row and updating the solved x_i value in the shared memory. In this algorithm, there is a separate array of flags, which maintains the status of solved x_i values that are ready to use in solving rows. Therefore, each thread starts from the beginning of the row and waits till its dependent row is solved while checking the flag array. Once it sees that a dependency is solved, it continues to update the respective b_i value, move into the next dependency, and wait till that is solved. If there are no more dependencies, the thread completes the processing of the diagonal element, updates the global x array and the respective x_i flag, and moves into a new row. Fig. 3.2. illustrates this process and timestamp where threads are deployed for each row, and threads 0,1,2, and 4 have just finished processing their rows.

This algorithm is relatively efficient if the triangular matrix has a higher number of interdependent rows (i.e., a higher number of levels in the DAG and thinner levels). For FPGAs, this algorithm is challenging due to the following reasons:

- Managing a shared memory as both x vector and flag array has to be shared between multiple processing elements in an FPGA implementation.
- Providing multiple read access to the same memory location of a larger memory (i.e., broadcasting) is costly in FPGA.

4. Hardware Architecture for the Analytical Model

In this section, we will describe the SpTRSV architecture we are planning to build the analytical model.

4.1 Overview

Similar to the LevelST implementation, we assume that we will tile the triangular matrix into smaller triangular matrices and square/rectangular matrices in the final processing and try to assess the computation time for smaller triangular matrices.
We also use the level-set algorithm to reorder our data for the triangular matrix processing. We consider a node in the dependency graph to represent the entire row. Therefore, while processing a node, we send its non-diagonal data followed by the diagonal value. Similar to LevelST, when we send matrix data, we pack the row index, column index, and value together and send them as one data packet. We also send how many non-zero values in the row before sending actual data.

4.2 A Single Processing Element (PE)

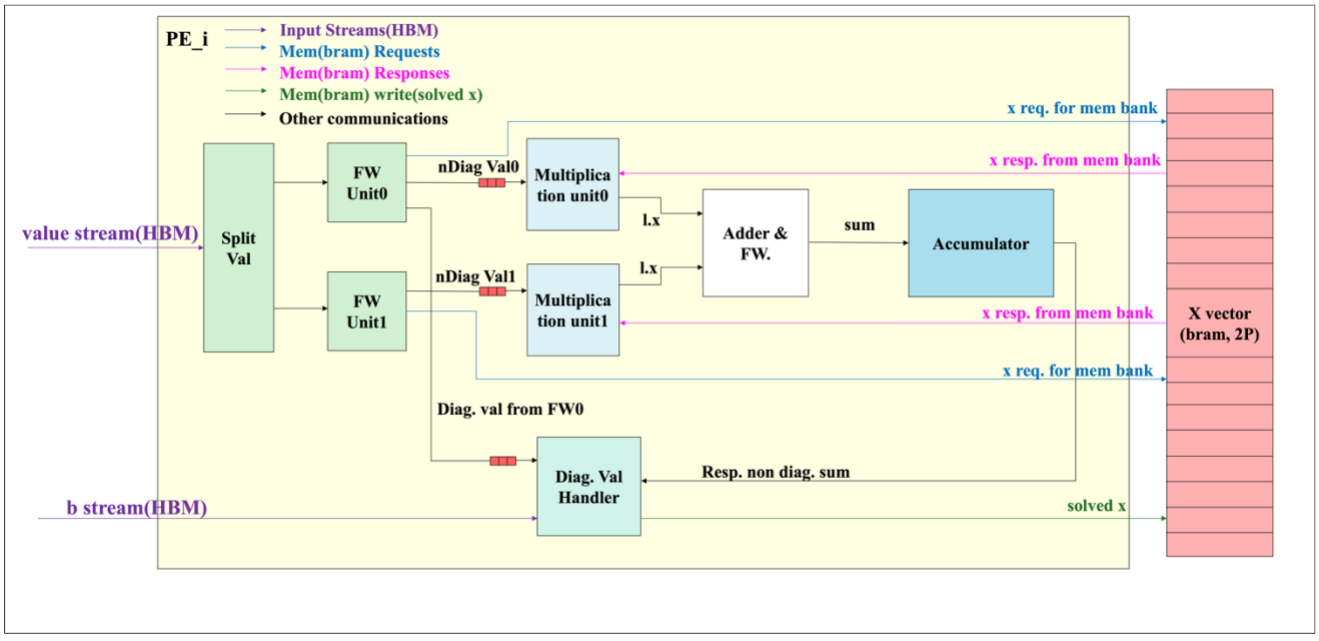


Fig. 4.1: A Processing Element (PE)

Fig. 4.1 illustrates an example architecture of a processing element in our analytical model. This example drawing has two multiplication units in it and this number is configurable.

The functionality of each subcomponent is summarized below.

- Split Val: This component receives the matrix data from the off-chip memory. Since High Bandwidth Memory (HBM) has a wide channel, we use coalescing to utilize off-chip bandwidth and pack two data packets as a PE contains two multiplication units that can work in parallel. "Split Val" unit splits coalesced data into two data packets and forwards them into FW_unit0 and FW_unit1.
- FW_unit0/FW_unit1: The first value forwarding units receive is the number of non-zero values in the next incoming row. It first samples that data, stores it in a local register, and starts sampling actual data.

  FW_unit0 decides whether the arrived data is a diagonal value or a non-diagonal value. If it is a diagonal value, then the unit forwards the value to the "Diag. Val Handler" to generate a new x_i value. We make sure that diagonal values are only received in FW_unit0 during data packing so the routing inside the PE is simple. This does not affect the compute performance anyway as a PE only contains one Diag. Val Handler.

  If it is a non-diagonal value, that means, multiplication and accumulation have to be done. FW_unit0/FW_unit1 sends a request to the X vector for the respective x_i value and inserts the matrix value to a FIFO connected to the respective multiplication unit.

  In the case of the last non-diagonal value of a row, the forwarding unit appends a signal to the data stream which will be later used in the accumulator to send the accumulated sum to the diagonal value handler.
- Multiplication unit0/Multiplication unit1: these units perform the multiplication for non-diagonal values between the matrix value and previously solved x_i values. Each multiplication unit has two inputs: 1) matrix value coming from the forwarding units and 2) x_i value coming from the X vector. Both inputs are connected to the multiplication unit via FIFOs. Therefore, the multiplication unit waits till both inputs are available to process data to make sure the correct matrix value is multiplied with the correct x_i.
- Adder & FW: This unit receives the multiplication outputs from both multiplication units. This unit adds them together and forwards the sum into the accumulator.
- Accumulator: This unit handles the floating-point accumulation of processed non-diagonal data. One of the challenges in this unit is that floating-point accumulation takes multiple clock cycles to complete. Hence, if we try to accumulate to the same

variable, it will not be able to perform accumulation with the initiation interval (II) of 1. Therefore, this unit has an array to keep partial sums, and accumulation is done to these partial sums based on the dependency distance. After all the values related to the processing row are received (the accumulator gets to know of this based on the signal generated by the forwarding units), the accumulator computes the total sum and passes the value to the diagonal value handler. Since floating point sum also requires a couple of clock cycles, this operation is performed by an adder tree inside the accumulator to maintain II=1.

- Diag. Val Handler: The diagonal value handler does the final computation of solving for a new x_i value. It takes inputs as b_i value, accumulated sum, and the diagonal value. If all the values are available in their respective FIFOs, this unit samples the data and, performs the subtraction and division to calculate a new x_i value. Then, it stores the newly solved value in the X vector to be used later.

With these compute blocks, a single PE is equipped with all the computation units that are required to perform SpTRSV computation. Also, since we have multiple multiplication units inside the PE, we utilize the parallelism within the row in our PE.

4.3 A Processing Elements Group (PEG)

Two multiplication units inside the PE help to utilize the parallelism within a row (section 2.2.3). Therefore, this number is configurable, and we can create a PEG by combining multiple PEs together.

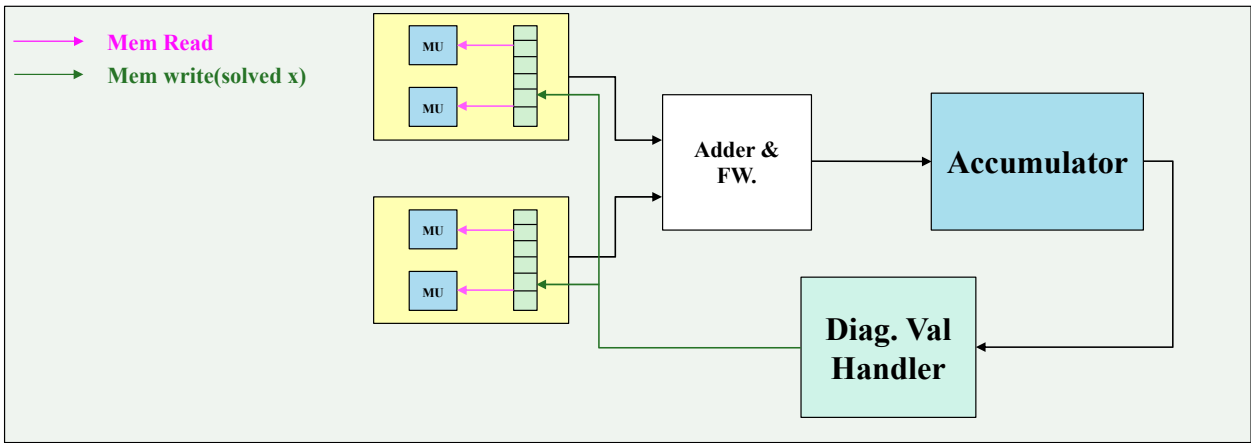


Fig. 4.2: A Processing Elements Group (PEG)

Fig. 4.2. illustrates the essential components of a PEG created by combining two PEs. Each multiplication unit needs to have access to an X vector which is stored in a two-port bram. Therefore, we duplicate the X vector for each two multiplication units and once a new x_i value is calculated, we update that value in all the copies. As shown in Fig. 4.2, adder & FW unit, accumulator and Diag. Val Handler is taken out from the PEs and shared between PEs inside a PEG.

4.4 X-vector update between PEGs

We allocate different rows in a level to different PEGs to process in parallel (i.e., row parallelism). However, it is necessary to have every PEGs to be updated with all the newly solved x_i values before moving to the next level. Since all to all broadcasting is very costly in FPGAs, similar to LevelST, we use the ring communication method to broadcast data to all the PEGs as shown in Fig. 4.3.

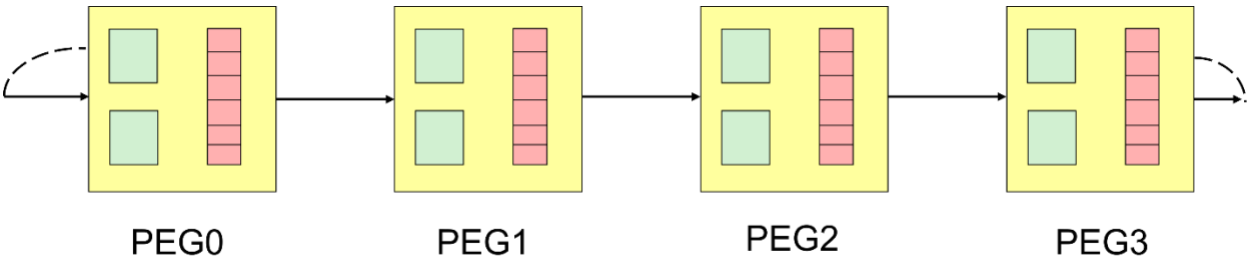


Fig. 4.3: Ring communication among PEGs

There are two ways we can start the ring communication.

1. Start the ring communication after completion of the entire level.
2. Start the ring communication soon after the data is computed.

Since some of the computation and memory writes can overlap by using the 2nd method, in this analytical model we consider the second method.

4.5 Two architectures based on the X-Vector partitioning

During our analysis, we tested two architectures based on the partitioning decisions that we can take for the X-vector.

1. No partitioning
2. A partition dedicated for each PEG

4.5.1 Architecture1: No partitioning

In this option, we have a single array in each PEG (duplicated for PEs) without any partitioning. The advantage of this is multiplication units have access to all the values of the X-vector irrespective of what index it is trying to access. The disadvantage is that, since the X-vector has only two ports, it must deprioritize reading data while data writing. Data writing happens due to internal solving of the PEG and due to ring communication.

4.5.2 Architecture2: A partition for each PEG

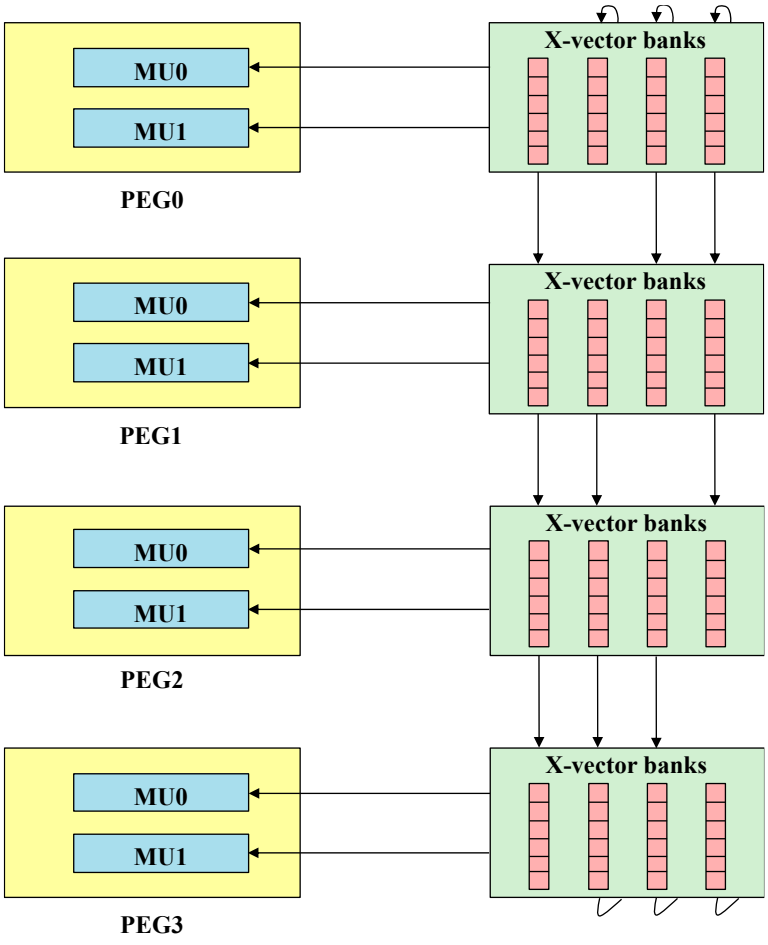


Fig. 4.4: Ring communication with separate banks for each PEG

An alternative option is to have a data partition for each PEG as shown in Fig. 4.4. In this figure, it has 4 PEGs, and therefore, each PEG has 4 partitions each representing a PEG. The advantage of this way is there will be no data conflicts between local data

writing and ring communication data writing. However, the disadvantage is that we have to dedicate access to a set of banks to a particular multiplication unit (MU). For example,

- MU0 has access to the partitions 0 and 2
- MU1 has access to the partitions 1 and 3

Having this dedicated access can cause an imbalance in the input data stream as we have to make sure respective indexes are scheduled to respective multiplication units.

4.6 Analytical model

Considering the above architectural model we have implemented modules using high level synthesis (HLS) to extract latency values with an initiation interval of 1 to be used in the analytical model. The extracted data is shown in Table 4.1.

Unit	Latency
FW unit	2
Multiplication unit	6
Adder & FW. Unit	9
Accumulator	27
Diag. Val Handler	21

Table. 4.1: Latency values of each sub module

Using this data, an analytical model was developed in C++ to compute the expected cycle count and other statistics of processing a SpTRSV tile.

5.Results

5.1 Definitions

5.1.1 Synchronization time

As we are using the level set algorithm, we need to make sure every newly solved x_i value is shared among all the PEGs before moving to the next level. This is called the synchronization barrier and the time taken for this is considered as synchronization time.

Since we start the ring communication just after doing a computation, the barrier synchronization time is the time taken to process the last value of the level and the time taken to propagate that resultant value to all the PEGs. Therefore,

Synchronization time per level = Time for the last row to be solved (Pipeline depth) + Time to update memory banks in all the PEGs (#PE groups -1)

As shown in Fig. 4.5, the synchronization happens after every level.

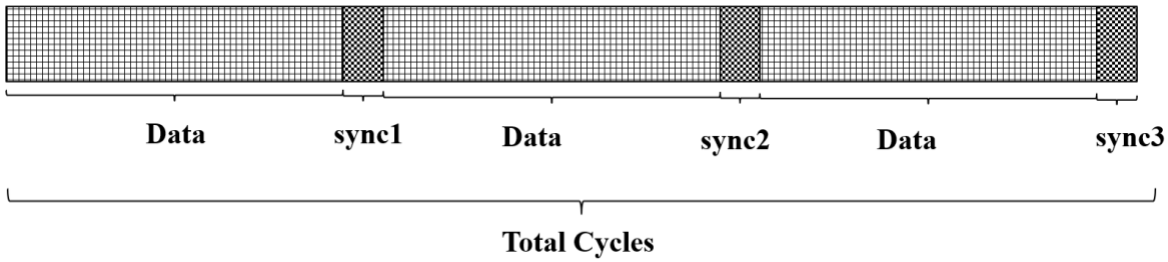


Fig. 4.5 Synchronization after each level

Therefore, total synchronization cost can be calculated as,

Total sync. cost = sync. time1 + sync. time2 + sync. time3 + …

Sync cost percentage = Total sync. cost/Total cycles

5.1.2 Dummy data

Invalid extra data that has to be added to balance data streams can be denoted as dummy data. Dummy data can be added for two reasons.

5.1.2.1 Imbalance row

Since we have multiple multiplication units inside a PEG, we need to parallelly send data to each multiplication unit by coalescing input data. This coalescing factor (or

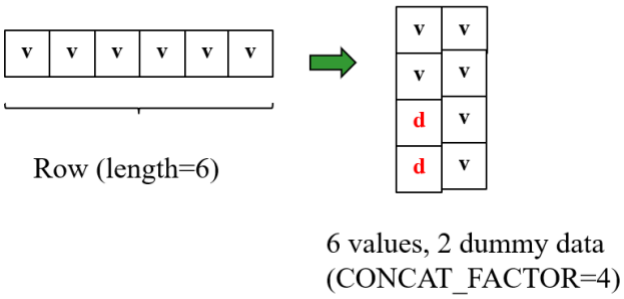


Fig. 4.6 Dummy data due to imbalance row

CONCAT_FACTOR) is equal to the number of multiplication units in a PEG. During this coalescing dummy data has to be added to balance imbalance rows. An example with a row size of 6 is shown in Fig. 4.6. Two dummy data have been added to balance the row in this figure.

5.1.2.2 Imbalanced PEGs

We assign rows in a level to PEGs cyclically. As the number of non-zero values in a row can vary, the length of the total data stream that goes to each PEG can also vary. If data multiple data streams are concatenated together to utilize off-chip HBM bandwidth, this also can lead to inserting dummy data to balance the data stream as shown by the red color in Fig. 4.7.

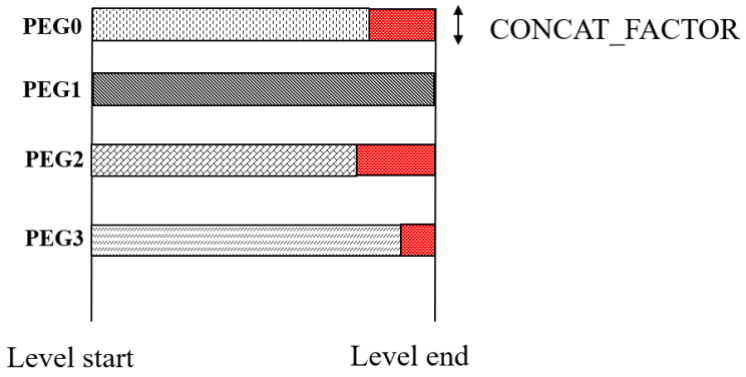


Fig. 4.7 Dummy data due to imbalance PEG

Therefore, total dummy data can be calculated as follows.

Total dummy data = (dummy data due to imbalanced rows) + (dummy data due to imbalanced PEG)

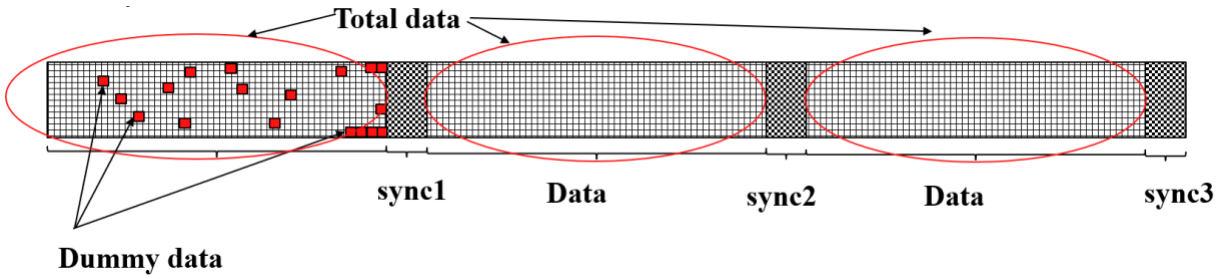


Fig. 4.8 Dummy data in total stream

As shown in Fig. 4.8., total dummy data percentage can be calculated as,

Total dummy data percentage = (Total dummy data) / (Total data)

5.1.3 Bubbles due to memory ports conflicts

As mentioned in section 4.5, managing the X-vector with two port brams can cause conflicts and create bubbles (idle computation cycles) inside the pipeline. We identify two causes of bubbles.

1. Write-write conflicts: this can happen when the units are trying to update the local X-vector from the values calculated by the group and ring communication simultaneously. In this kind of situation, we must prioritize one. In our analytical model, we delay the computation and allow ring communication to complete.
2. Read-write conflicts: For every newly solved x_i value, we need to get access to one port to write that value to memory. Therefore, we need to stop the reading on one port and allow write to happen making a bubble in computation happening on that read port.

Total bubbles can be calculated as the sum of bubbles created by the above two scenarios.

5.2 Test configurations

We collected results on the following hardware configuration which matches with available resources on Xilinx Alveo U280 FPGA.

- Number of multiplication units per PEG = 4
- Number of PEGs = 48
- Number of HBM banks = 24
- Triangular matrix tile size = 32768 x 32768

Using these configurations we have collected results for the 1st diagonal tile of multiple sparse matrices collected from suite sparse collection [8] and compared against HDAGG performance.

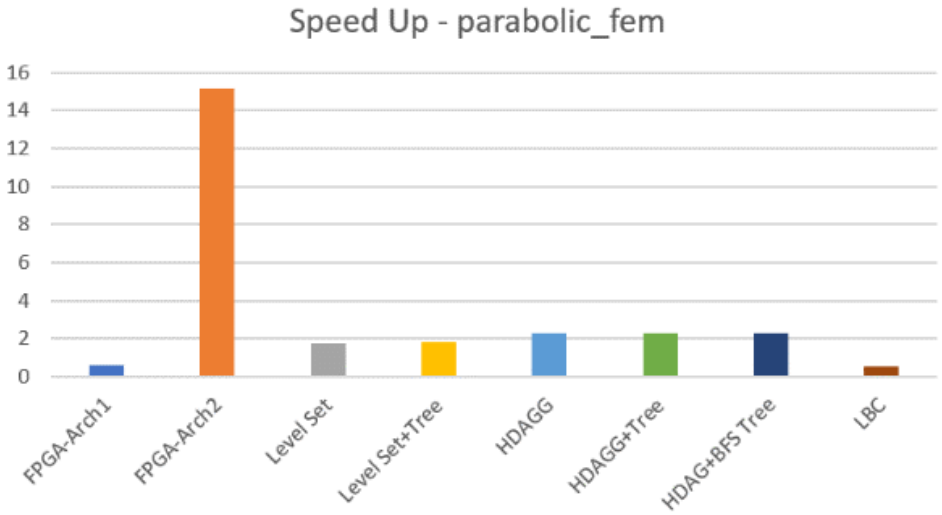
5.3 Test results for sparse matrices

The following sub sections report results observed in some unique matrix patterns in increasing number of levels order.

5.3.1 parabolic_fem

nnz	#Levels	Largest L. idx	Largest L. size	Smallest L. idx	Smallest L. size
32768	1	1	32768	1	32768

Feature	Arch 1	Arch 2
Dummy data percentage	75.01% (Row-74.96%, PE-0.05%)	75.01% (Row-74.96%, PE-0.05%)
Bubbles	15686(95.42%)	-
Sync. Cycles	70(0.43%)	70(9.30 %)
Total cycles	16349	753
Total time(with 225MHz)	0.000073 s	0.000003 s

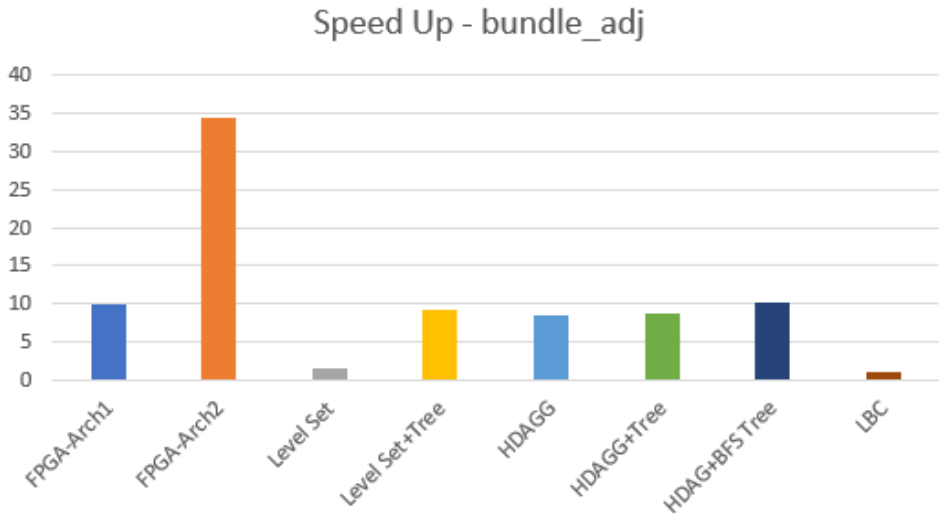


This matrix tile has perfect parallelism, with only values on the diagonal. Therefore, our FPGA-architecture two with partitioning can achieve a very high speed up compared to CPU due to parallelism and pipelining. However, architecture 1 suffers from write conflicts and creates many bubbles (compute stalls) causing it to have a slowdown.

5.3.2 bundle_adj

nnz	#Levels	Largest L.idx	Largest L. size	Smallest L. idx	Smallest L. size
847146	9	8	9804	6	552

Feature	Arch 1	Arch 2
Dummy data percentage	6.48% (Row-6.32%, PE-0.16%)	32.45% (Row-22.58%, PE-9.88%)
Bubbles	21010(78.54%)	-
Sync. Cycles	1022(3.82%)	1022(13.53%)
Total cycles	26750	7554
Total time(with 225MHz)	0.000119 s	0.000034 s

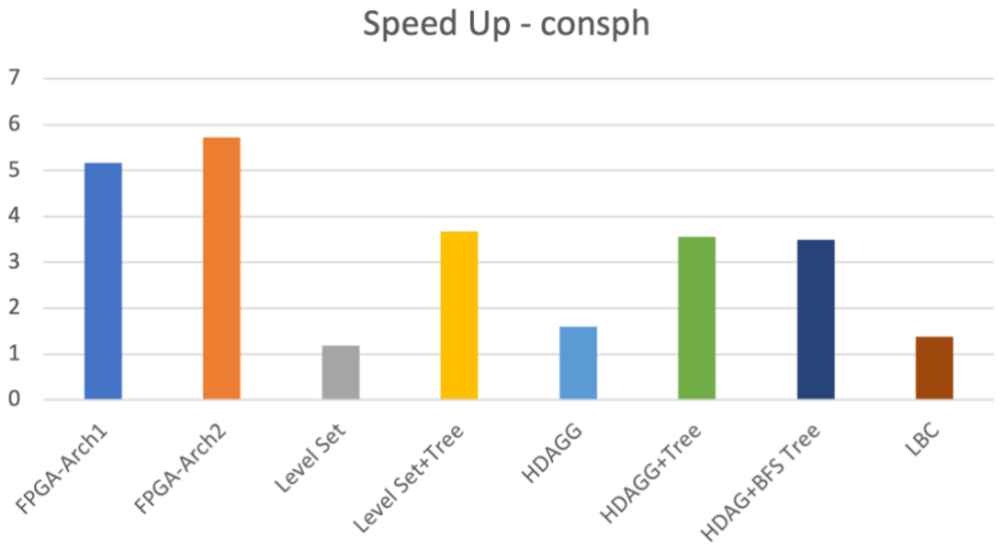


This matrix size also has a lower number of levels and larger levels. Therefore, the FPGA architecture 2 can achieve higher performance even with a higher number of dummy data due to not having writing conflicts. In the meantime, architecture 1 gets similar performance as well optimized CPU solution. This is majorly due to even though it utilizes the available parallelism, write conflicts create many bubbles.

5.3.3 consph

nnz	#Levels	Largest L.idx	Largest L. size	Smallest L. idx	Smallest L. size
1140030	330	1	1612	330	1

Feature	Arch 1	Arch 2
Dummy data percentage	23.62% (Row-3.70%, PE-19.92%)	49.22% (Row-15.71%, PE-33.51%)
Bubbles	9819(17.45%)	-
Sync. Cycles	39221(69.70%)	39221(77.03%)
Total cycles	56268	50914
Total time(with 225MHz)	0.000250 s	0.000226 s

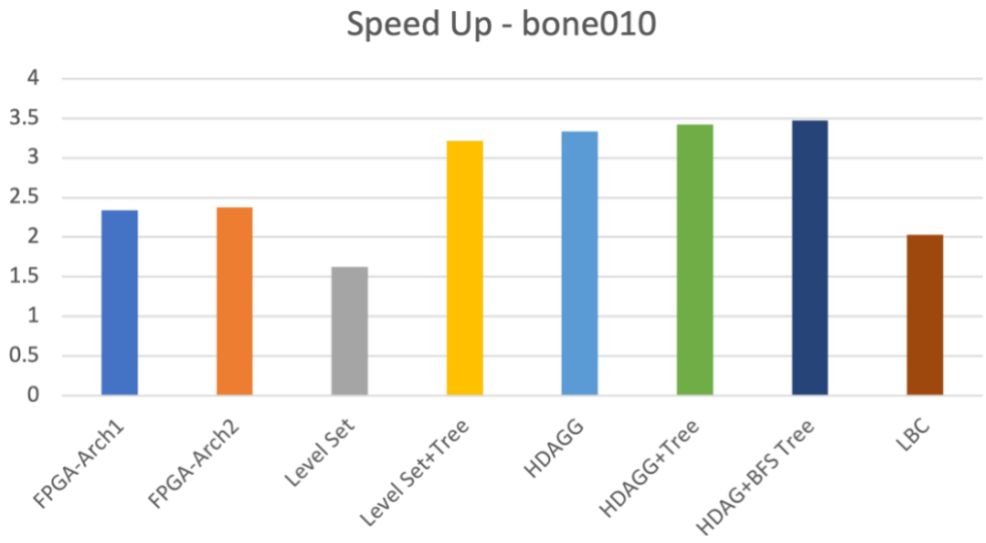


The 1st tile of the consph matrix can perform better in both of our architectures compared to CPU execution. Even though it has hundreds of levels and causes higher synchronization overhead, since the level sizes are big, both FPGA solutions benefit from the row level parallelism, parallelism within a row, and pipelining. Therefore, both achieve a good speed up over the CPU solution.

5.3.4 bone010

nnz	#Levels	Largest L.idx	Largest L. size	Smallest L. idx	Smallest L. size
671231	532	98	124	532	2

Feature	Arch 1	Arch 2
Dummy data percentage	31.55% (Row-6.01%, PE-25.53%)	54.31% (Row-13.47%, PE-40.85%)
Bubbles	4088(5.68%)	-
Sync. Cycles	63259(87.83%)	63259(89.21%)
Total cycles	72028	70911
Total time(with 225MHz)	0.000320 s	0.000315 s

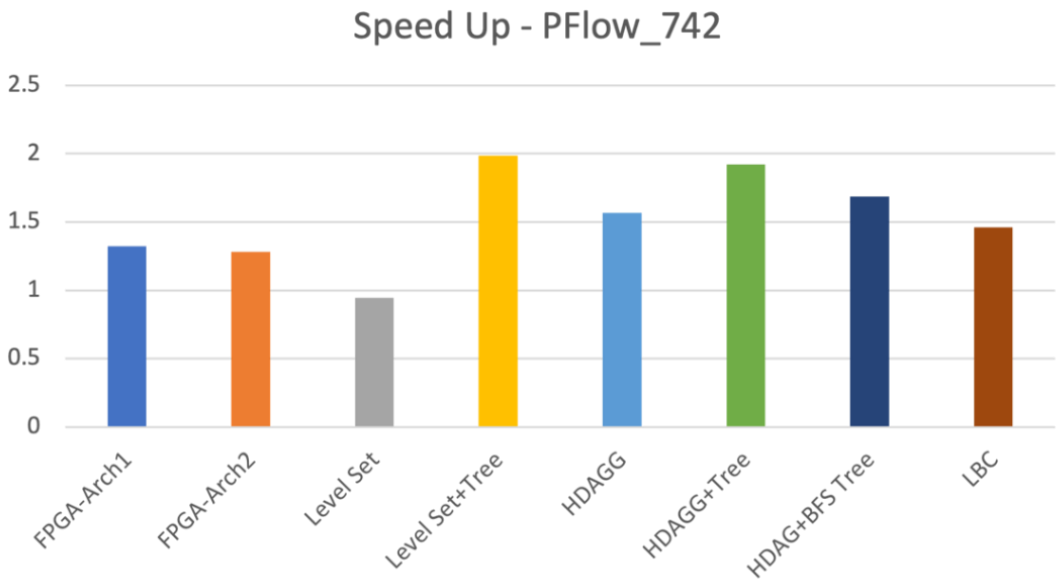


When it comes to this tile, the number of levels has risen to 532, and level sizes have gone down. Therefore, we can observe that the synchronization cost has gone higher in both FPGA architectures and started to perform slower than the optimized CPU versions. One thing to note here is that even though the CPU solution proposed by HDAGG is also faced with similar or lower synchronization overhead (due to their optimizations number of levels can be smaller) it can achieve better timing as the CPU clock frequency is nearly 10x faster than the FPGA clock frequency.

5.3.5 PFlow_742

nnz	#Levels	Largest L.idx	Largest L. size	Smallest L. idx	Smallest L. size
872367	1328	887	49	1328	1

Feature	Arch 1	Arch 2
Dummy data percentage	61.12% (Row-2.07%, PE-59.05%)	73.40% (Row-9.18%, PE-64.22%)
Bubbles	1(0.00%)	-
Sync. Cycles	157983(93.11%)	157983(90.24%)
Total cycles	169670	175063
Total time(with 225MHz)	0.000754 s	0.000778 s

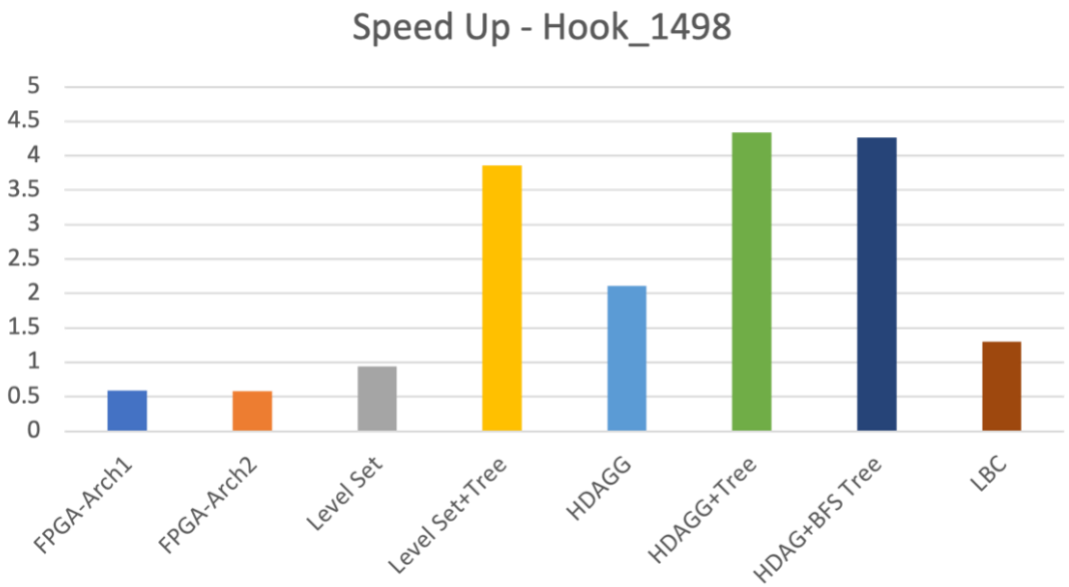


Similar to the consph matrix, this tile in Pflow_742 also has a higher number of shorter levels. Therefore, even though both FPGA architectures achieve a smaller speed up compared to the single thread CPU, more optimized CPU versions can achieve a better speed up compared to the FPGA architectures.

5.3.6 Hook_1498

nnz	#Levels	Largest L.idx	Largest L. size	Smallest L. idx	Smallest L. size
584223	1861	1	3606	1861	1

Feature	Arch 1	Arch 2
Dummy data percentage	79.31% (Row-1.94%, PE-77.37%)	85.61% (Row-6.84%, PE-78.77%)
Bubbles	1702(0.72%)	-
Sync. Cycles	221410(93.10%)	221410(91.28%)
Total cycles	237819	242551
Total time(with 225MHz)	0.001057 s	0.001078 s

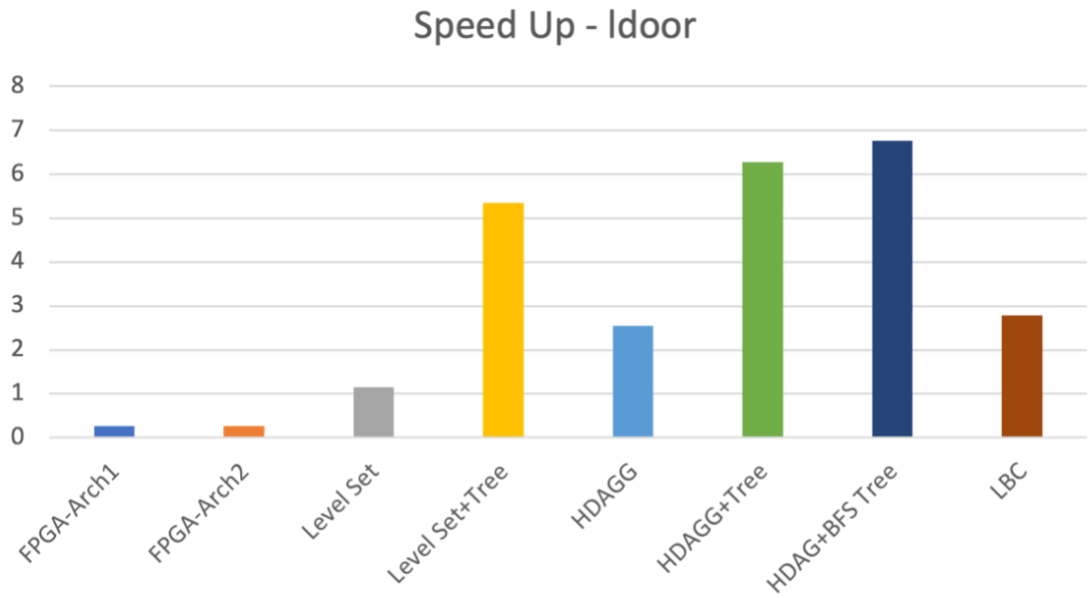


With this matrix tile, the FPGA architecture starts to see a slowdown compared to the single-thread CPU. This is majorly because, as the number of levels has risen to 1861, synchronization cost outweighs the parallelism on the FPGA. However, more optimized CPU solutions continue to perform better as they can reduce synchronization costs by merging multiple levels into one.

5.3.7 ldoor

nnz	#Levels	Largest L.idx	Largest L. size	Smallest L. idx	Smallest L. size
669542	4675	1	1247	4668	1

Feature	Arch 1	Arch 2
Dummy data percentage	91.28% (Row-0.67%, PE-90.61%)	94.37% (Row-3.13%, PE-91.23%)
Bubbles	2356(0.39%)	-
Sync. Cycles	556276(92.93%)	556276(89.99%)
Total cycles	598608	618179
Total time(with 225MHz)	0.002660 s	0.002747 s

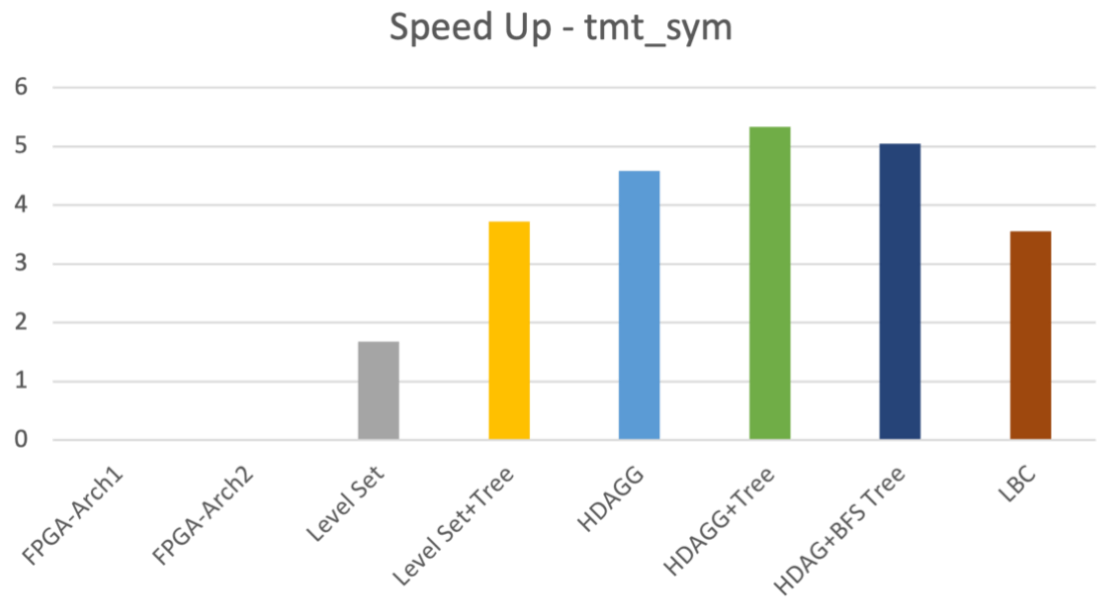


With this tile in matrix door, with an even higher number of levels, FPGA architectures start to perform very poorly and have a considerable slowdown compared to the single thread CPU. Lower clock frequency along with the synchronization cost are the major causes of this.

5.3.8 tmt_sym

nnz	#Levels	Largest L.idx	Largest L. size	Smallest L. idx	Smallest L. size
129930	32587	1	80	32587	1

Feature	Arch 1	Arch 2
Dummy data percentage	98.33% (Row-0.42%, PE-97.91%)	98.64% (Row-0.73%, PE-97.91%)
Bubbles	0(0.00%)	-
Sync. Cycles	3877804(98.96%)	3877804(98.73%)
Total cycles	3918373	3927536
Total time(with 225MHz)	0.017415 s	0.017456 s



This tile in tmt_sym has a near serial access pattern as the non-diagonal values in each row tend to be present closer to the diagonal. Therefore, it can be observed that the synchronization cost of both FPGA solutions has risen up close to 99%. However, CPU implementations still perform better with merging possible levels, exploiting cache locality, and operating on higher frequency.

5.4 *Average non-zeros per level vs speed up*

Similar to above, we calculated the speed up of multiple sparse matrices and plotted the speed up compared to the CPU version with average non-zeros per level as illustrated in Fig. 5.1 and Fig. 5.2.

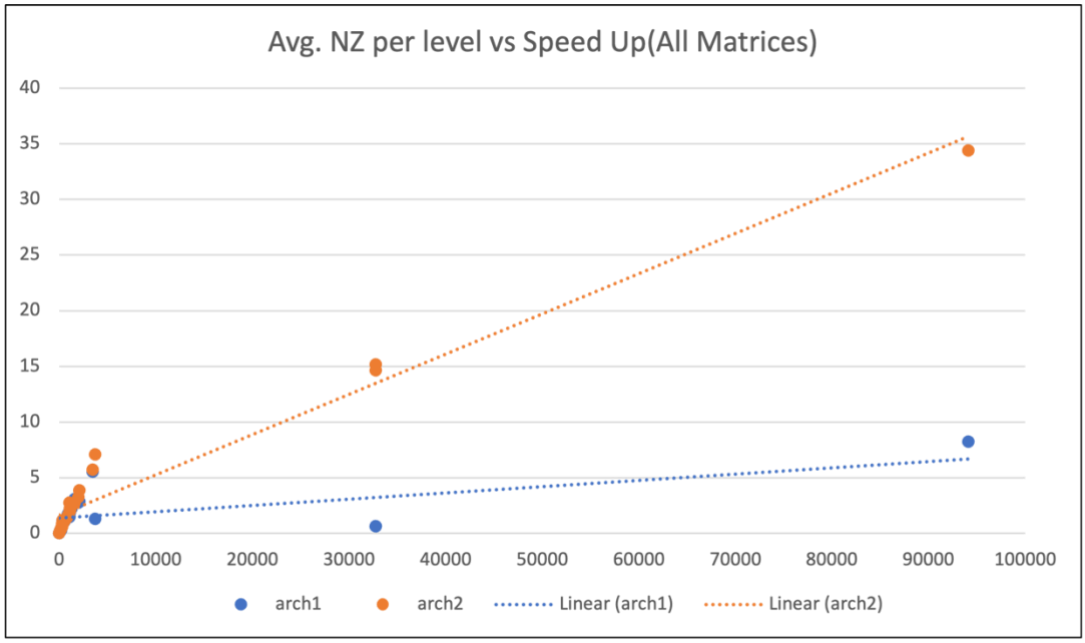


Fig. 5.1 Average non-zeros per level vs speed up

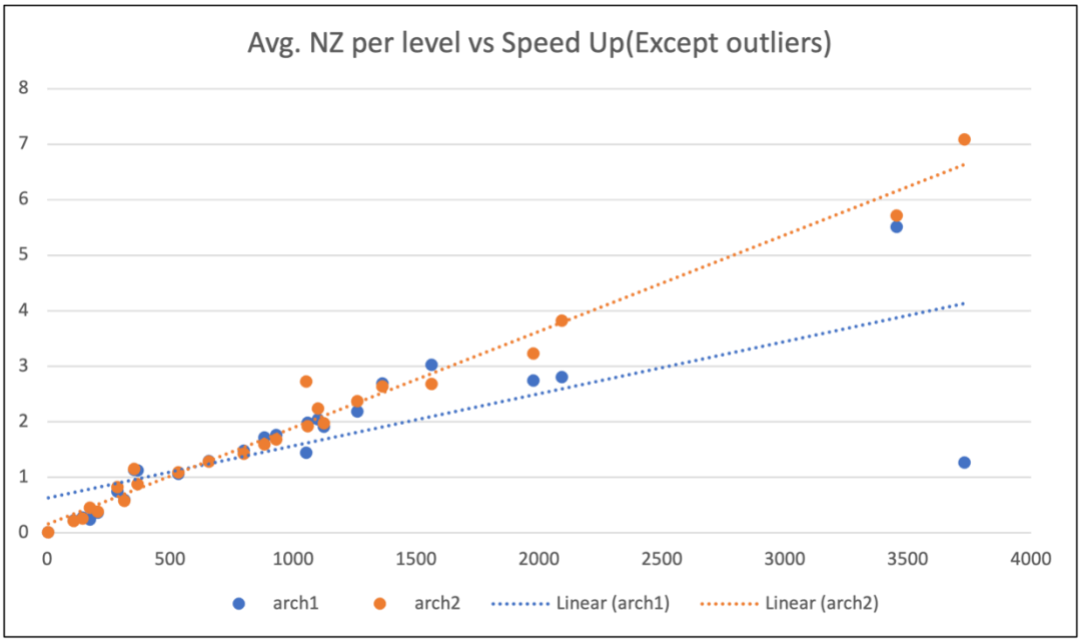


Fig. 5.2 Average non-zeros per level vs speed up (except outliers)

5.5 Discussion

According to these results, it can be observed that level set based SpTRSV performance varies a lot based on the characteristics of the input matrix. It can be observed that when number of levels increases to a couple of hundreds FPGA implementation starts to perform poorer than the CPU mainly due to synchronization cost outweighing the parallelism. Based on Fig. 5.1 and Fig. 5.2, it is clear that the higher the average number of non-zeros per level, the better the speed up in our proposed architecture.

In addition, we can see that even though bank access is bound to a particular multiplication unit (and hence has a higher dummy data percentage), architecture 2 with X-vector partitions performs better than the single memory bank architecture.

However, overall, we can observe that the synchronization cost, bubble percentage, and dummy data percentage are higher in our proposed architecture. Certain optimizations like using stream-aware tiled level-set algorithm used in LevelST can help to reduce bubble percentage.

6. Conclusion

In this work, we studied SpTRSV and related accelerators proposed for different hardware platforms. Then, we developed an analytical model for a SpTRSV architecture for FPGAs. This analytical model helped us predict the cycle counter and speed up along with different characteristics of the input matrix. We observe that this architecture performs better with input matrices with a lesser number of levels and bigger levels. i.e., if the average non-zeros per level is higher, we can achieve a better speed-up. In addition, this analytical model helps us identify bottlenecks like synchronization cost, number of bubbles, and number of dummy data while processing each input. These details help us to further investigate optimization strategies and fine-tune the architecture.

7. References

[1] N. Ahmad, B. Yilmaz and D. Unat, "A Split Execution Model for SpTRSV," in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 11, pp. 2809-2822, 1 Nov. 2021, doi: 10.1109/TPDS.2021.3074501.

[2] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. Int. J. High Speed Comput. 1, 1 (May 1989), 73–95. https://doi.org/10.1142/S0129053389000056

[3] Alan George, Michael Heath, Joseph Liu, and Esmond Ng. 1986. Solution of sparse positive definite systems on a shared-memory multiprocessor. Int. J. Parallel Program. 15, 4 (Oct. 1986), 309–325. https://doi.org/10.1007/BF01407878

[4] B. Zarebavani, K. Cheshmi, B. Liu, M. M. Strout and M. M. Dehnavi, "HDagg: Hybrid Aggregation of Loop-carried Dependence Iterations in Sparse Matrix Computations," 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 2022, pp. 1217-1227, doi: 10.1109/IPDPS53621.2022.00121.

[5] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient Block Algorithms for Parallel Sparse Triangular Solve. In Proceedings of the 49th International Conference on Parallel Processing (ICPP '20). Association for Computing Machinery, New York, NY, USA, Article 63, 1–11. https://doi.org/10.1145/3404397.3404413

[6] Federico Favaro, Ernesto Dufrechou, Pablo Ezzatti, and Juan P. Oliver. 2020. Exploring fpga Optimizations to Compute Sparse Numerical Linear Algebra Kernels. In Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings. Springer-Verlag, Berlin, Heidelberg, 258–268. https://doi.org/10.1007/978-3-030-44534-8_20

[7] Zifan He, Linghao Song, Robert F. Lucas, and Jason Cong. 2024. LevelST: Stream-based Accelerator for Sparse Triangular Solver. In Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24). Association for Computing Machinery, New York, NY, USA, 67–77. https://doi.org/10.1145/3626202.3637568

[8] Kolodziej et al., (2019). The SuiteSparse Matrix Collection Website Interface. Journal of Open Source Software, 4(35), 1244, https://doi.org/10.21105/joss.01244