Breaking Sequential Dependencies in FPGA-based Sparse LU Factorization

Siddhartha
Nanyang Technological University
Singapore
siddharth@pmail.ntu.edu.org

Nachiket Kapre
Nanyang Technological University
Singapore
nachiket@ieee.org

***Abstract*—**
Substitution, and reassociation of irregular sparse LU factorization can deliver up to 31% additional speedup over an existing state-of-the-art parallel FPGA implementation where further parallelization was deemed virtually impossible. The state-of-the-art implementation is already capable of delivering 3× acceleration over CPU-based sparse LU solvers. Sparse LU factorization is a well-known computational bottleneck in many existing scientific and engineering applications and is notoriously hard to parallelize due to inherent sequential dependencies in the computation graph. In this paper, we show how to break these alleged inherent dependencies using depth-limited substitution, and reassociation of the resulting computation. This is a work-parallelism tradeoff that is well-suited for implementation on FPGA-based token dataflow architectures. Such compute organizations are capable of fast parallel processing of large irregular graphs extracted from the sparse LU computation. We manage and control the growth in additional work due to substitution through careful selection of substitution depth. We exploit associativity in the generated graphs to restructure long compute chains into reduction trees.

I. Introduction

Sparse LU factorization is a commonly-used numerical kernel in many scientific and engineering problems. It is performance limited due to memory bottlenecks associated with irregular access of the sparse matrix data structures. On parallel hardware, it may be possible to distribute memory bandwidth across multiple concurrent interfaces, but the computation also consists of long sequential dependency chains that are notoriously hard to parallelize. Several software packages (*e.g.* [6], [3], [1], etc) and parallel hardware designs (*e.g.* [2], [5]) have been customized for computations on sparse matrices. These solvers exploit a few limited opportunities available for performance optimization such as (1) sparsity reduction, (2) static analysis for data layout, and (3) spatial processing.

In Figure 1, we show the potential for further parallelization of dataflow implementation of a sparse LU compute graph (`bomhof2`). We plot the number of nodes in the graph at a given level (work) against latency of that node from the input (depth). The original graph has a *long tail distribution* of parallelism. Here, most of the parallelism is at the head of the graph (depth=0) but we still have a long sequential tail (depth≈75) that defeats parallel scaling. When we apply a depth 8 substitution and reassociation transformation, we generate substantially more work (about 10-100×), but achieve an important saving in the depth of the graph (depth≈50) which reduces the critical latency in the computation by 1.5×. We see later in Section 3, how to control this explosion in amount of redundant work while still recovering useful additional parallelism on top of the ≈3× that is already possible [5].

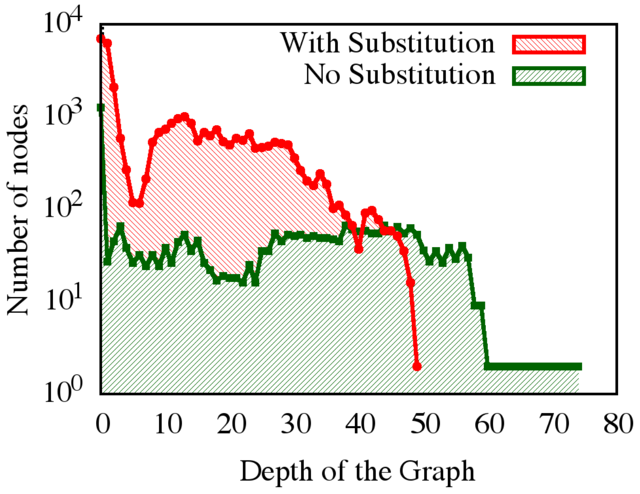

Fig. 1: Impact of substitution and reassociation osdsn node count and latency (`bomhof2`)

The contributions in this body of work can be summarized as follows:

- Combination of two techniques, substitution and reassociation, that exposes further parallelism in sparse LU factorization than previously thought possible (Section 3)
- Modified token-dataflow hardware design supported by compiler that statically transforms the dataflow graph using our novel approach (Section 3)
- Experimental framework to quantify the performance of various graph transformations and hardware configurations

II. Background

A. Sparse Matrix Factorization

Numerical problems in computing often require solving a system of linear equations $A\vec{x} = \vec{b}$ as part of an iterative loop. In this paper, we focus on parallelization of LU factorization in such scenarios (*e.g.* circuit simulation) that permit upfront,

	Sequential	**Complete Substitution + Reassociation**	**Depth 2 Substitution + Reassociation**
x_1	b_1	b_1	b_1
x_2	$b_2 - L_{21}\ x_1$	$b_2 - L_{21}\ b_1$	$b_2 - L_{21}\ b_1$
x_3	$b_3 - L_{32}\ x_2 - L_{31}\ x_1$	$b_3 - L_{32}\ b_2 + L_{32}\ L_{21}\ b_1 - L_{31}\ b_1$	$b_3 - L_{32}\ x_2 - L_{31}\ x_1$
x_4	$b_4 - L_{43}\ x_3 - L_{42}\ x_2 - L_{41}\ x_1$	$b_4 - L_{43}\ b_3 + L_{43}\ L_{32}\ b_2 - L_{43}\ L_{32}\ L_{21}\ b_1 + L_{43}\ L_{31}\ b_1 - L_{42}\ b_2 + L_{42}\ L_{21}\ b_1 - L_{41}\ b_1$	$b_4 - L_{43}\ b_3 + L_{43}\ L_{32}\ x_2 + L_{43}\ L_{31}\ x_1 - L_{42}\ x_2 - L_{41}\ x_1$
Operations	6 multiplies, 6 adds	17 multiplies, 11 adds	10 multiplies, 8 adds
Critical Latency[1]	8	5	6

[1]assuming unit delay model

Fig. 2: Substitution & Reassociation transformations on the front-solve

one-off static analysis of the computation to extract and exploit parallelism. We extend the KLU solver, [4] optimized for fast evaluation of circuit simulation matrices, for our parallelization study. The KLU solver performs a spatial reordering of rows and columns in the matrix which makes it possible for the non-zero structure in the intermediate matrices to remain static or fixed for subsequent iterations. This feature allows us to pre-allocate the data structures at the start of an iterative phase and allow us to arrange the sparse matrix in memory for faster access. More importantly, it makes it possible to expose dataflow parallelism in the resulting unrolled compute graph for a token dataflow FPGA implementation. The Gilbert-Peierls (GP) algorithm [4] is the key building block of the KLU solver. Runtime is dominated by the repeated call to a *front-solve* in every iteration of the *for* loop over matrix columns. In this paper, we show specifically how this front-solve can be sped up in every iteration to achieve better performance.

B. Token Dataflow Architecture for Sparse Matrix Solve

Sparse LU factorization on FPGAs has previously been considered in [5], by fully unrolling the loops in the GP algorithm. From this unroll, the authors construct large, irregular, unstructured graphs corresponding to each matrix benchmark. Non-zero entries in A are inputs to this graph, the non-zero values of the L and U factors are the outputs, and the rest of the nodes are multiplication, addition and division (very few) operations. This dataflow graph is then partitioned across a 2D network of processing elements organized as a parallel token dataflow architecture. Each PE evaluates the nodes using the dataflow firing rule instead of using a program counter to sequence operations. Under the dataflow firing rule, a node will execute asynchronously and autonomously if it receives all its inputs. Each PE contains logic to manage the fully pipelined arithmetic operators (multiply, add and divide) and dataflow triggers. It also contains a local memory implemented using BRAMs to store portions of the graph structure. The PEs are connected using an NoC and are capable of processing a packet per cycle at the PE-NoC interface. We use this token dataflow architecture as a starting point for our hardware parallelization experiments.

III. Parallelization Strategy

In this section, we explain the two key ideas in our parallel design. As noted earlier, even with complete unroll of all the for-loops in GP Algorithm, there still remain sequential dependencies between the evaluation of different elements of x inside the front-solve. This is the cause of the long tail distribution observed in Figure 1. In Figure 2, we show row solutions for a simple *dense* 4x4 front-solve computation. We will use this example consistently to explain key concepts in this section.

A. Recursive Substitution

From Figure 2, we note the equations required to resolve elements in x have a sequential order. If we recursively substitute the solutions for earlier x in the downstream expressions, we can rewrite the expressions such that they can be resolved independently without sequential dependencies. Unfortunately, a complete recursive substitution for the entire computation gives us an exponentially larger compute graph for dense LU factorization, with mostly redundancy computations (see growth of expressions in Figure 2). Even for *sparse* matrices, our transformations yielded a 30–40$\times$ increase in graph size . Hence, our challenge was to create opportunities for asymptotic reductions while controlling growth in redundant work. We address this challenge by doing a **depth-limited** substitution instead. For the 4x4 example in Figure 2, when substituting up to a depth of 2, we reduce the number of nodes in the resulting dataflow graph while still exposing opportunities for parallelism. This transformation is closely linked to the next step, reassociation, which enables reduction in the critical path latency.

B. Reassociation

Substitution by itself decouples the computation and removes unnecessary dependencies, but it does not reduce the critical path. If we restructure the long multiply and add chains into $\lceil log(N) \rceil$ trees, we can obtain an asymptotic reduction in critical input$\rightarrow$output latency. We quantify these improvements for the dense 4x4 example in Figure 2. Note that depth-limited substitution affects the reassociation such that we achieve smaller savings in the critical path latency. However, this is still advantageous since we save communication latency costs that are important when considering parallel scaling on NoC hardware. We should note that since we use floating-point arithmetic, a casual reassociation will not yield bit-exact results. We evaluate the resulting error residue on the two graphs through a C++ simulation and find very little if any change in resulting error properties.

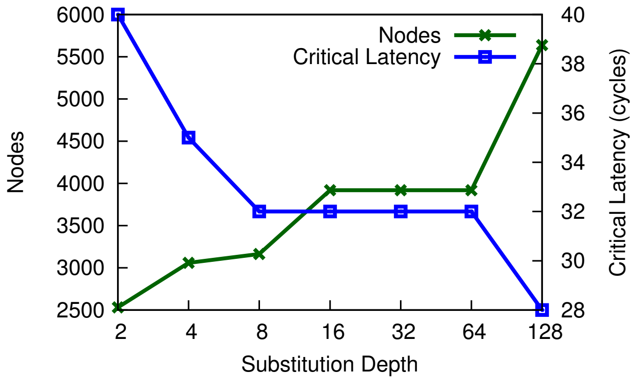

Fig. 3: Node and critical path latency trends with increasing depth-limited substitution (`bomhof2`)

In Figure 3, we show the operation count and critical path latency trends for a single front-solve iteration in `bomhof2` sparse matrix benchmark when substituted to varying depths and subsequently reassociated. By observing the trends in Figure 3, we can postulate that a depth of 4 or 8 is suitable for this particular iteration.

IV. Methodology

In this section, we describe our experiments to determine the performance of our new design in comparison to the state-of-the-art implementation of the packet-switching NoC [5].

A. Experimental Setup

We do fast prototyping and testing of different FPGA configurations using a home-brewed cycle-accuracte simulator. We support matrices in the Matrix Market format (*.mtx*), which can then be analyzed by the KLU pre-processor to be converted into dataflow graphs. We partition the dataflow graphs to fit into each PE's local memory using MLPart-5.2.14 and calibrate the switching latency based on the size of each PE to meet the target 250MHz frequency. We also restrict the division operation only for one PE, since there are significantly fewer division operations required than additions and multiplications. This allows us to fit an 8x8 PE configuration with our new PE design, as opposed to 4x4 previously in [5].

B. Design Space Search

For each of the benchmarks to be tested, we have a very large design space to search. We need to select a suitable substitution depth, d, architecture configuration PE combination for parallel scaling. To keep the search space feasible, we limit our substitution depth search by tracking the increase in work. Likewise, we limit the possible combinations of d to powers of 2, which generate dataflow PEs that use similar resources as the original token dataflow design. We also only consider square PE mesh topology configurations (*e.g.* 3x3, 4x4, etc). All these experiments must be repeated for **every** Front-Solve step in the GP algorithm for each benchmark. To accelerate the search of our design space, we run these experiments in parallel across a cluster of eight Intel Xeon E5-2609 machines. This large design space search challenge opens up new research avenues to develop tools that could be trained to find these optimal operating points for each benchmark.

V. Evaluation

In this section, we investigate key factors affect the final performance of our new design. The experimental results are summarised in Table I and Table II. We observe the best speedup for a single benchmark to be 31% and the mean speedup across all benchmarks to be 20%.

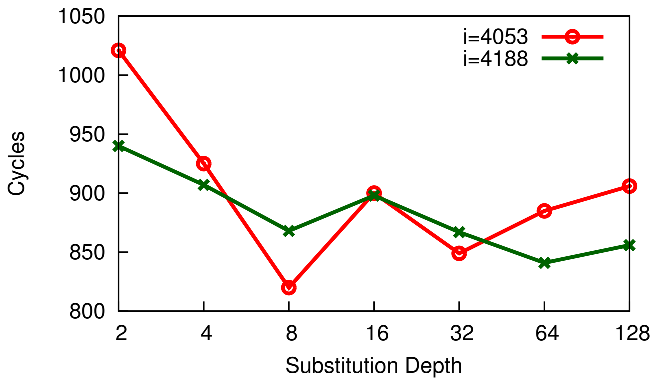

Fig. 4: Impact of Depth (`bomhof2`)

Depth. The choice of substitution depth is a significant consideration, as increasing the substitution depth causes an increase in the size of the dataflow graph. It must be chosen such that the latency savings are not swamped by the cost of processing this increased workload on the NoC. For most benchmarks, a depth of 4 is a suitable value that consistently delivers reasonable results. Figure 4 shows how depth of substitution affects the performance for two front-solve iterations (4053 & 4188) in `bomhof2` benchmark. Note how each of the front-solves has its own minima at different depths. This is part of the challenge of finding a suitable depth for the entire benchmark and also a potential research topic on hybrid depth selection. For now, we select a uniform substitution depth for our each front-solve in all benchmarks. Furthermore, not every front-Solve in each iteration of the GP algorithm is compute-intensive. In most cases, the front-solve dataflow graphs are small and have insignificant communication costs (*e.g.* less than 8% of front-solve graphs in `sandia` have > 100 edges). Transforming the smaller dataflow graphs does not deliver savings in cycle count and in many cases, results in a drop in performance due to the increased communication costs (*e.g.* `10stages`). Hence, we only carry out our graph transformations on front-solves that are sufficiently large and receptive to these techniques. The percentage of all front-solves transformed with depth-limited substitution and reassociation is reported under the "FST" column in Table II.

PE Configuration. For parallel scalability analysis, we vary the number of PEs in our token dataflow design and identify the ideal PE configuration that yields best performance. Too few PEs mean performance is limited by computing power while a larger number of PEs will suffer from long communication delays. Figure 5 shows this effect measured on a single Front-Solve iteration in `bomhof2`. Substitution and

TABLE I: Benchmark Graph Properties

Benchmark	**Rows**	**Sp.**	**Sub.**	**Graph Properties**						
				Nodes	**Edges**	**Constants**	**Adds**	**Multiplies**	**Operations**	**Critical Path**
10stages	3,920	0.2%	WS	341k	389k	146k	72k	123k	195k	63k
			D4	340k (1.0×)	391k (1.0×)	145k (1.0×)	72k (1.0×)	124k (1.0×)	196k (1.0×)	63k (1.0×)
bomhof1	2,624	0.5%	WS	1.6m	2.0m	554k	500k	518k	1.0m	24k
			D4	2.1m (1.3×)	2.9m (1.5×)	673k (1.2×)	618k (1.2×)	821k (1.6×)	1.4m (1.4×)	22k (0.9×)
bomhof2	4,510	0.1%	WS	2.7m	3.5m	929k	834k	908k	1.7m	53k
			D4	4.1m (1.5×)	5.6m (1.6×)	1.3m (1.4×)	1.1m (1.3×)	1.7m (1.9×)	2.8m (1.6×)	53k (1.0×)
bomhof3	12,127	0.03%	WS	700k	840k	280k	188k	232k	420k	50k
			D4	760k (1.1×)	959k (1.1×)	280k (1.0×)	203k (1.1×)	277k (1.2×)	446k (1.1×)	48k (1.0×)
simucad	4,875	0.3%	WS	5.5m	7.1m	2.0m	1.6m	1.9m	3.6m	79k
			D4	11.0m (2.0×)	15.6m (2.2×)	3.2m (1.6×)	2.8m (1.8×)	5.0m (2.6×)	7.8m (2.2×)	81k (1.0×)
hamm	17,758	0.4%	WS	12.0m	15.1m	4.3m	3.4m	4.2m	7.6m	283k
			D4	14.8m (1.2×)	19.4m (1.3×)	5.1m (1.2×)	3.9m (1.2×)	5.8m (1.4×)	9.6m (1.3×)	309k (1.1×)
sandia	25,187	0.03%	WS	989k	961k	508k	166k	315k	481k	111k
			D4	981k (1.0×)	999k (1.0×)	482k (0.9×)	169k (1.0×)	330k (1.0×)	500k (1.0×)	110k (1.0×)

Sp. = Sparsity, Sub. = Substitution, WS = Without Substitution, D4 = Depth 4

TABLE II: Benchmark Performance Cycles

Benchmark	**Sub.**	**Configuration**			
		FST	**PE**	**Cycles**	**Speedup**
10stages	WS	-	1x1	971k	-
	D4	5%	2x2	966k	1.01×
bomhof1	WS	-	4x4	1.4m	-
	D4	25%	8x8	1.1m	1.21×
bomhof2	WS	-	4x4	2.0m	-
	D4	84%	8x8	1.6m	1.20×
bomhof3	WS	-	2x2	1.2m	-
	D4	20%	8x8	948k	1.21x
simucad	WS	-	4x4	5.2m	-
	D4	84%	8x8	3.6m	1.31×
hamm	WS	-	3x3	12.1m	-
	D4	95%	8x8	8.8m	1.27×
sandia	WS	-	1x1	2.0m	-
	D4	10%	8x8	1.7m	1.18×

WS = Without Substitution, Sub. = Substitution, D4 = Depth 4
FST = percentage of front-solves transformed

reassociation allows us to continue scaling up to 36 PEs from the 16 PEs that was previously possible while achieving a 26% improvement in performance.

VI. Conclusions

We show how to parallelize sparse LU factorization using FPGAs beyond what was previously thought possible. The underlying idea of "recursive substitution and reassociation", at first glance, appears infeasible due to the increased cost of managing redundant work. To handle this challenge, we combined two techniques, depth-limited substitution and re-association, to break the inherent sequential dependencies in

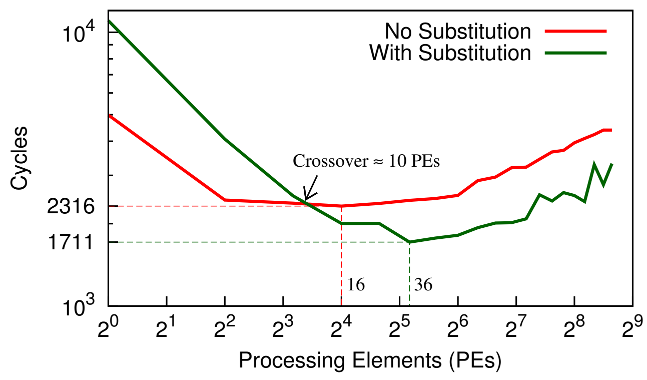

Fig. 5: Impact of PE scaling (`bomhof2`)

the computation. Across a range of benchmarks, we show a speedup of 1.01–1.31× when compared to the state-of-the-art FPGA design that already delivers 3× speedup over conventional CPU hardware.

References

[1] P. Amestoy, I. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Eng.*, 2000.

[2] X. Chen, Y. Wang, and H. Yang. NICSLU: An adaptive sparse matrix solver for parallel circuit simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(2):261–274, 2013.

[3] T. A. Davis. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions On Mathematical Software*, 30(2):196–199, June 2004.

[4] T. A. Davis and E. Palamadai Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3):36:1–36:17, Sept. 2010.

[5] N. Kapre and A. DeHon. Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs. In *Field-Programmable Technology*, 2010.

[6] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions On Mathematical Software*, 31(3):302–325, September 2005.