Fanout Decomposition Dataflow Optimizations for FPGA-based Sparse LU Factorization

Siddhartha
School of Computer Engineering
Nanyang Technological University
Email: siddharth@pmail.ntu.edu.sg

Nachiket Kapre
School of Computer Engineering
Nanyang Technological University
Email: nachiket@ieee.org

***Abstract*—**
Performance of FPGA-based token dataflow architectures is often limited by the long tail distribution of parallelism in the compute paths of the dataflow graphs. This is known to limit speedup of dataflow processing of Sparse LU factorization to only 3–10× over CPUs. One reason behind the limitations is the serialization penalty of processing high-fanout nodes in the dataflow graph on traditional dataflow processing architectures. In this paper, we show how to perform one-time static fanout decomposition and selective node replication transformations to input dataflow graphs. These transformations are one-time static compute costs that are typically amortized over millions of iterations. For dataflow graphs extracted for sparse LU factorization, we demonstrate up to 2.3× speedup (1.2× geomean average) with this technique across a range of benchmark problems.

I. Introduction

FPGA-based token dataflow architectures are an increasingly important design choice for accelerating many hard computational problems where parallelism is sparse, and irregular. In these circumstances, a raw unrolled dataflow graph exposes all possible parallelism in the computation in its purest form. The dataflow architectures allow asynchronous, decoupled evaluation of parallelism in irregular graphs without the need for complex synchronization protocols. The dataflow graph execution proceeds using a simple dataflow firing rule where a node is fired when all its inputs are received. Sparse LU factorization is one such representative engineering application that is notoriously hard to parallelize and is considered a challenging problem for conventional processors. It is a well-known compute bottleneck in fields such as circuit simulation [5], computational fluid dynamics [3], bioinformatics, among many others. When the sparse matrices are fixed, we can extract its unique dataflow graph for LU factorization. Hardware-assisted token dataflow acceleration of Sparse LU implementations (*e.g.* [6], [10]) can deliver non-trivial speedups of 3–10× over CPU-based solvers. However, this is currently achieved non-optimally, leaving substantial room for improvement.

Dataflow graphs often have a few nodes with large fanout counts. Typically fanout nodes are processed in sequence in the dataflow PE, resulting in serialization bottlenecks for large fanouts. In Figure 1, we observe serialization of up to ≈ 180 cycles to process a single node fanouts. As we scale system sizes, high fanout nodes quickly become the performance bottleneck. We need to reduce this overhead

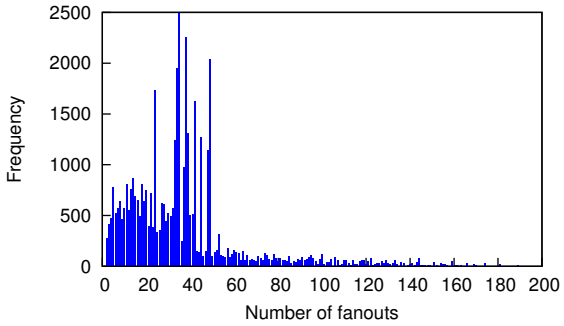


Fig. 1: Benchmark Fanout properties (`bomhof2`)

by distributing fanout serialization across multiple PEs. We achieve this by doing controlled fanout decomposition on high fanout operator nodes and node replication of constant nodes with high fanouts. These static transformations are one-time steps that can be amortized over millions of iterations in Sparse LU factorization problems.

The key contributions in this paper are:

- Design of a dataflow compiler that performs fanout decomposition/node replication in the dataflow graphs.
- Quantification of performance of the dataflow compiler on sparse matrix benchmarks selected from the circuit simulation domain.

II. Background

A. Token Dataflow Architecture

Token Dataflow architectures were the subject of academic studies in the early 1990s *e.g.* [7], [2], [4]. However, due to the emergence of the killer microprocessors, these designs and ideas were largely relegated to academic projects. At an abstract level, the dataflow architecture is composed of PEs connected by switched network fabric. Computation on this architecture is organized as a sequence of "token" communication along graph dependency edges and subsequent "dataflow firing" at the graph nodes. Each PE has local memory blocks that are used to store portions of the dataflow graph for localized processing. Each PE is capable of performing logic and/or arithmetic operations on each node of the graph based on a dataflow firing rule. Under this rule, each node is allowed to independently and asynchronously compute when it has all inputs ready. Dependencies between nodes are

Algorithm 1: Gilbert-Peierls

```
   Data: sparse matrix A
   Result: factors L & U
1  L = I;
2  for i=1:N do
3  |   b = A(: , i);
4  |   x = L\b;
5  |   U(1:i , i) = x(1:i);
6  |   L(i+1:N , i) = x(i+1:N) / U(i , i);
7  end
```

routed through the packet-switched token communication network. For FPGA-based systems, dataflow processing offers an unique opportunity to deliver a reprogrammable and scalable computing substrate that can be tailored to different applications. In this paper, we consider a customized heterogeneous token dataflow architecture optimized for sparse LU factorization as the vehicle for our experiments and optimizations.

B. Sparse LU Factorization

In many numerical problems, we are required to solve a set of linear equations expressed as $A\vec{x} = \vec{b}$ in matrix-vector notation. Matrix A is often a highly sparse matrix that stays structurally unchanged when working with real-world applications. For *e.g.* , in circuit simulation, each circuit component is only connected to a few neighboring elements, thereby resulting in very localized non-zero patterns when represented as a matrix. The hardware design we consider in this paper is based on the KLU solver [1]. KLU does a one-time pre-ordering step that fixes non-zero locations in the matrix, allowing us to keep the dataflow and memory structure static throughout an iterative process. This step is especially suitable for parallel hardware-assisted solvers [6], as there is no need to recompute the dataflow graph and do dynamic memory allocation in each iterative step. At the heart of the KLU solver is the Gilbert-Peierels (GP) algorithm (Listing 1). The GP algorithm is responsible for generating the L & U factors for the input matrix A. In [6], we unrolled the for-loop in the GP algorithm to generate giant dataflow graphs that represent the GP compute flow. However, a front-solve (line 4) must be carried out in each step of the for-loop, which becomes a compute bottleneck as the lower-triangular (L) matrix is iteratively built in each for-loop iteration. In [8], we targeted this front-solve by doing a one-time recursive depth-limited substitution and reassociation to further expose any available parallelism. In this paper, we take our experiments further by testing our dataflow optimizations on the substituted and reassociated dataflow graphs and simulate performance on a heterogeneous token dataflow hardware architecture [9].

C. Token Dataflow Architecture for Sparse LU Factorization

For our intended scenario, the front-solve in sparse LU factorization operates in single-precision floating-point arithmetic. The numerical calculations are mostly multiplies and adds with a few divides. For our heterogeneous dataflow design shown in Figure 2, we customize the ALU functions

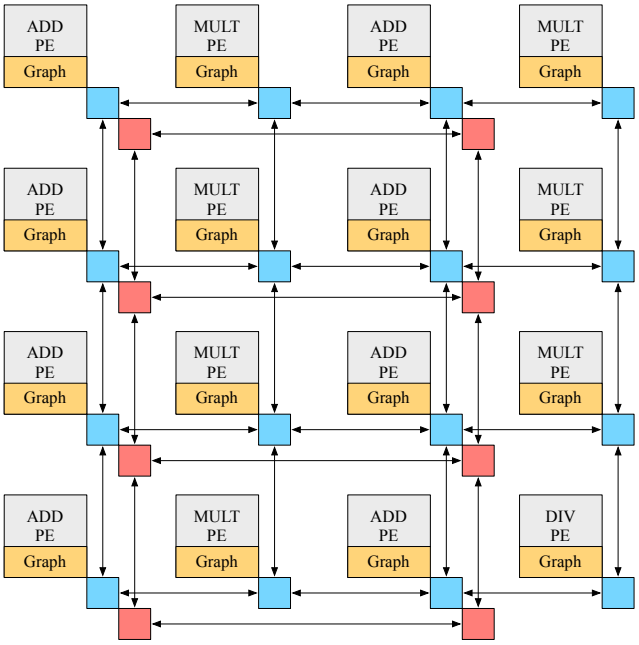


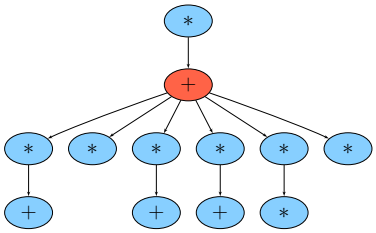
Fig. 2: Heterogeneous Token Dataflow architecture (add:mult = 1:1, two NoC channels)

handled by each PE to reflect this distribution. We also customize the communication network by adding a faster multi-hop channel between the add PEs to allow critical dependencies to be routed faster.

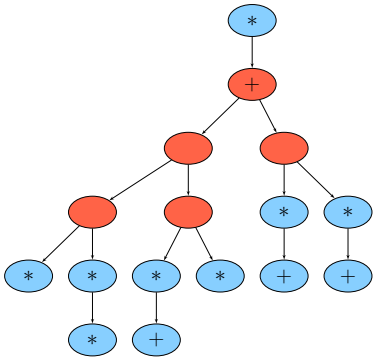
III. Fanout Optimizations

Performance of token dataflow architectures is often limited by the long tail distribution of parallelism in the dataflow graph. Within these constraints, performance is often exacerbated by the oblivious processing of high-fanout nodes. For high-fanout nodes, serialization of packet transmissions can limit performance as only one fanout can be serviced every cycle. To tackle this issue, we propose a fanout decomposition scheme as demonstrated in an example in Figure 3. We use a similar strategy for constant input nodes with large fanout, where instead of creating a fanout tree, we perform locality-aware node replication, which is a cheap memory tradeoff for improved performance.

To implement fanout decomposition, we first define two control parameters: threshold (f_t) and arity (f_a). If the fanout size of a node in our input dataflow graph is greater than f_t, we perform fanout decomposition on that node. The decomposition is carried out such that arity of decomposed fanout tree is not greater than f_a. Therefore, each node in the decomposed tree has no more than f_t fanouts, and each newly-created node in the decomposed tree has no more than f_a fanouts. Under these constraints, we decompose the fanouts in the most balanced way possible, such that the fanouts are distributed across the new copy nodes (new red nodes in Figure 3(b)) as evenly as possible. Selecting values for f_t and f_a could be potentially complex – for example, we could design a dynamic threshold/arity selection scheme based on graph properties, PE



(a) Undecomposed



(b) Arity-2 Decomposition

Fig. 3: Fanout Decomposition Example

configuration and/or placement information. In this study, we do a simplified design space search for potentially optimal f_t and f_a values (denoted as *(f_t,f_a)* tuple) for each benchmark. We limit our search space to powers of 2 threshold/arity values (up to *(16,4)*), and draw conclusions based on results observed. We observe that a *(16,4)* fanout decomposition design point has the best results when considered across all benchmarks.

IV. Methodology

In this section, we detail our compilation flow and comment on our hardware design characteristics.

A. Dataflow Compilation Flow

For quantifying the performance limits of our dataflow hardware and compiler, we extract dataflow graphs for sparse LU matrices. Our matrix pre-processors convert input matrices from circuit simulation domain represented in the Matrix Market (*.mtx*) format into corresponding dataflow graphs. Our dataflow compiler applies fanout transformations as discussed earlier in Section III. We quantify any speedups observed with reference to the baseline performance in [6], where no optimizations were used. Our flow is currently supporting sparse LU factorization graphs, but it is general and applicable to other domains beyond circuit simulation where applications can be characterized by large, irregular dataflow graphs.

B. Dataflow Hardware Design

We target the Xilinx Virtex-6 SX475T FPGA device similar to the one used in [6]. This limits the largest heterogeneous dataflow processing system we can accommodate on this device to 12x12 (144 PEs) [9]. The switching latencies are calibrated to meet the target 250MHz design frequency. Our design can optimally support dataflow graphs that can fit the entire data structure into the on-chip BRAM memory blocks. We use the same data transfer and communication latency models established in [5].

TABLE I: Benchmark Graph Properties

Benchm.	Rows	Sp.	Graph Properties					
			Nodes	Edges	Const.	Adds	Mults	Crit. Path
s27	189	3.3%	5.4k	5.7k	2.6k	0.9k	1.9k	1k
s208	1,296	0.5%	116k	137k	47k	22k	46k	12k
s344	1,992	0.3%	126k	145k	54k	22k	50k	15k
s349	2,017	0.3%	129k	147k	55k	23k	51k	13k
s298	1,801	0.4%	220k	267k	87k	48k	85k	16k
s382	2,219	0.3%	287k	351k	111k	57k	119k	20k
bomhof3	12,127	0.03%	760k	959k	280k	203k	277k	48k
bomhof1	2,624	0.5%	1.9m	2.6m	628k	575k	711k	24k
s953	4,872	0.2%	4.0m	5.3m	1.1m	841k	1.6m	76k
bomhof2	4,510	0.1%	6.1m	8.4m	1.6m	1.4m	2.4m	49k
simucad	4,875	0.3%	6.6m	8.8m	2.2m	1.9m	2.5m	75k

Sp. = Sparsity

TABLE II: Benchmark Cycles and Speedups

Benchmark	Cycles and Speedup				
	BASE	FANOUT			
	Cycles	f_t	f_a	Cycles	Speedup
s27	17k	8	2	25k	0.7×
s208	229k	4	1	290k	0.8×
s344	265k	4	1	349k	0.8×
s349	254k	8	2	318k	0.8×
s298	388k	4	1	437k	0.9×
s382	470k	16	4	548k	0.9×
bomhof3	1.2m	16	4	1.2m	1.0×
bomhof1	2.1m	16	4	1.1m	1.9×
s953	2.8m	16	4	2.2m	1.3×
bomhof2	2.7m	16	4	1.5m	1.8×
simucad	6.7m	16	4	2.9m	2.3×
GEOMEAN					**1.2**×

V. Results

In this section, we present our results and offer a brief discussion and future outlook on the observed trends. We test a total of 11 benchmarks, extracted from the circuit simulation domain. The benchmarks range in size/sparsity from hundreds to tens of thousands of non-zeros. The graph properties of dataflow graphs extracted from the input matrices are tabulated in Table I. We report graph properties in Table I for *(f_t,f_a) = (16,4)* design point.

A. Benchmark Sizes

We note speedups between 0.7–2.3× when our fanout decomposition optimization is enabled. For small benchmarks,

e.g. `s27`, `s208` and `s298`, we observe a slowdown of 0.7–0.9× as the graphs are too small to benefit from fanout restructuring. These benchmarks are composed of few nodes, and they have a relatively fewer number of nodes with high fanout counts. Decomposing high-fanout nodes in small benchmarks adds communication workload (extra packets representing decomposed nodes) that is a significant proportion of overall workload. Hence, any small benefits from decomposing high fanout nodes is lost to communication penalties. For large dataflow graphs, speedups are proportionally larger as noted in Figure 4. These larger dataflow graphs contain many more nodes with relatively higher fanout counts, and often, the critical path lies along these high fanout nodes. Fanout decomposition in these dataflow graphs delivers significant cycle savings that offset any extra communication overhead from extra nodes created during fanout decomposition. From Figure 4, we observe that we can only start achieving speedups for benchmarks larger than ≈1M nodes. Note the (f_t, f_a) selection also varies with size of benchmark (Table II), which could be used as a heuristic for building a performance model for cheap one-time (f_t, f_a) design point selection.

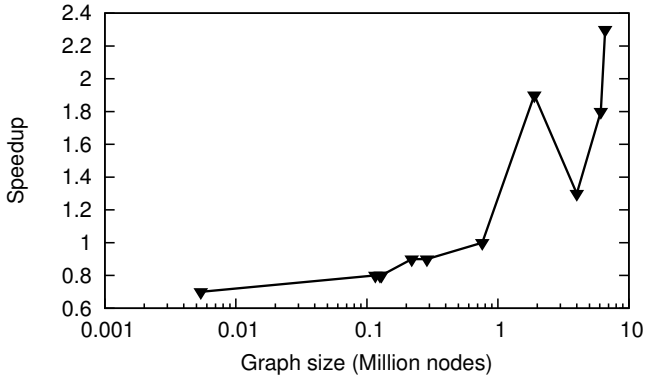


Fig. 4: Correlation between size of dataflow graph and speedup with dataflow optimization enabled

B. Benchmark Scaling Properties

In Figure 5, we show the effect of scaling PEs on performance of a select 5 benchmarks. We note a linear improvement in performance at small PE counts with a saturation effect at PE counts above 64 for most of the cases. Certain benchmarks like `bomhof3` and `s953` do not scale particularly well due to limited parallelism in the input itself (long critical paths). These benchmarks serve as motivation for developing dataflow optimizations that are able to push performance despite lack of available parallelism.

VI. Conclusions

In this paper, we show how to improve performance of FPGA-based token dataflow architectures through a static fanout restructuring dataflow optimization strategy. We show how to achieve an additional speedup of up to 2.3× (mean

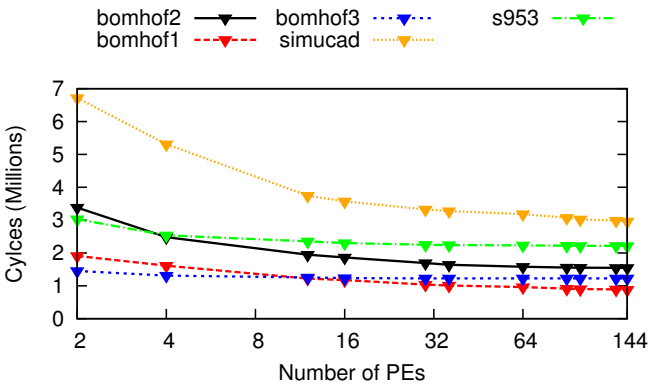


Fig. 5: Cycles vs PEs scaling trends for several benchmarks when all optimizations are enabled

1.2×) on top of existing performance of parallel dataflow hardware. Our static optimizations focus on decomposing/replicating high fanout operator/constant nodes to reduce serialization delays. The optimizations take advantage of statically-computed metrics, which are negligible one-time costs in highly iterative applications (*e.g.* SPICE). We demonstrate our approach on dataflow graphs that are representative of Sparse LU Factorization kernels, but our methods could be easily adapted for other domains that have fine-grained irregular data compute requirements.

VII. Future Work

We intend to extend our compiler to handle dataflow graphs from other domains beyond circuit simulation. We also seek to develop more static and runtime dataflow optimizations for improved dataflow architecture efficiency.

References

[1] 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.

[2] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, Dec. 1974.

[3] J. H. Ferziger and M. Perić. *Computational methods for fluid dynamics*, volume 3. Springer Berlin, 1996.

[4] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, Jan. 1985.

[5] N. Kapre. *SPICE2–A Spatial Parallel Architecture for Accelerating the SPICE Circuit Simulator*. PhD thesis, California Institute of Technology, Pasadena, 2010.

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

[7] G. Papadopoulos. Monsoon: a dataflow computing architecture suitable for intelligent control. *Intelligent Control, 1990. Proceedings., 5th IEEE International Symposium on*, 1990.

[8] Siddhartha and N. Kapre. Breaking Sequential Dependencies in FPGA-based Sparse LU Factorization. In *The International Conference on Field Programmable Logic and Applications 2014*, pages 1–4, Sept. 2014.

[9] Siddhartha and N. Kapre. Heterogeneous Dataflow Architectures for FPGA-based Sparse LU Factorization. In *FPL '14: Proceedings of the 2014 22nd IEEE Symposium on Field Programmable Custom Computing Machines*, pages 1–4, Mar. 2014.

[10] X. Wang and S. G. Ziavras. Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines. *Concurrency and Computation: Practice and Experience*, 2004.