Heterogeneous Dataflow Architectures for FPGA-based Sparse LU Factorization

Siddhartha
School of Computer Engineering
Nanyang Technological University
50 Nanyang Avenue, S639798
Email: siddhart005@e.ntu.edu.sg

Nachiket Kapre
School of Computer Engineering
Nanyang Technological University
50 Nanyang Avenue, S639798
Email: nachiket@ieee.org

***Abstract*—**
FPGA-based token dataflow architectures with heterogeneous computation and communication subsystems can accelerate hard-to-parallelize, irregular computations in sparse LU factorization. We combine software pre-processing and architecture customization to fully expose and exploit the underlying heterogeneity in the factorization algorithm. We perform a one-time pre-processing of the sparse matrices in software to generate dataflow graphs that capture raw parallelism in the computation through substitution and reassociation transformations. We customize the dataflow architecture by picking the right mixture of addition and multiplication processing elements to match the observed balance in the dataflow graphs. Additionally, we modify the network-on-chip to route certain critical dependencies on a separate, faster communication channel while relegating less-critical traffic to the existing channels. Using our techniques, we show how to achieve speedups of up to 37% over existing state-of-the-art FPGA-based sparse LU factorization systems that can already run 3–4× faster than CPU-based sparse LU solvers using the same hardware constraints.

I. Introduction

Sparse LU factorization is a key computational bottleneck in many well-known scientific and engineering problems. Due to memory bottleneck associated with irregular access of sparse matrix structures, it is often classified as a hard-to-parallelize challenge problem. Several software packages support sparse LU factorization (*e.g.* [1]). A few customized hardware designs for sparse LU factorization (*e.g.* [2]) demonstrate non-trivial speedups over the software solvers. While these speedups are promising, imbalanced ALU designs and high communication costs on the dataflow network often limit speedups beyond what is theoretically possible. In this paper, we propose an improved heterogeneous design based on an existing token dataflow architecture used in [2] for sparse LU factorization. We modify the dataflow processing elements to better reflect the distribution of add and multiply operations in the graphs. Our design also focuses on alleviating the communication bottleneck by adding a faster communication channel for routing critical dependencies in the factorization graphs. To exploit this modified design, we develop a graph partitioning recipe to ensure balanced, locality-aware distribution of compute operations. Figure 1 shows the multiply-to-add ratios in these dataflow graphs for the *bomhof2* benchmark (only non-zero iterations are plotted). Figure 2 shows the distribution

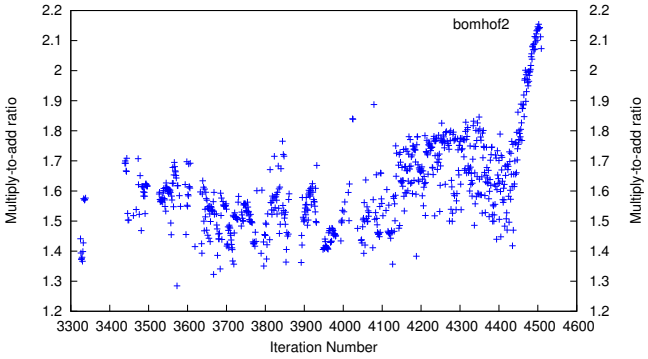

Fig. 1: Multiply-to-Add ratios in *bomhof2* benchmark

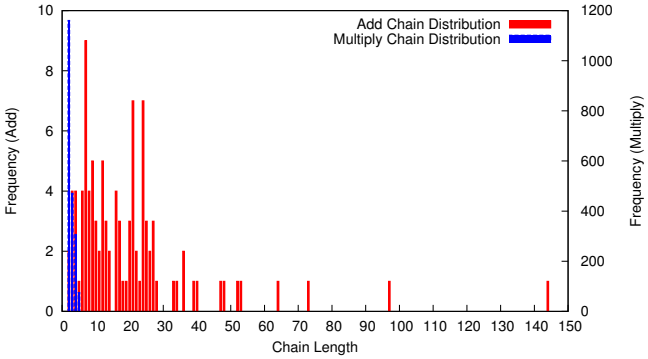

Fig. 2: Add/Multiply Chain Length Distribution in single depth-4 transformed iteration (*bomhof2*)

of the add/multiply chain lengths in one of the iterations in *bomhof2* benchmark. The heterogeneous approach is motivated by these add/multiply computational patterns observed in the dataflow graphs when they have been transformed using substitution and reassociation optimizations. The multiply-to-add ratios motivate us to decouple add/multiply processing into heterogeneous add/multiply PE block designs, while the significantly longer chain lengths of add operations motivate us to design a faster communication channel for the add PE blocks. Thus, deep customization of both compute and communication subsystems is crucial to delivering high performance for such challenging parallelization problems.

The key contributions in this paper are:

- Heterogeneous PE design and layout based on add/multi-

$$
\begin{aligned}
x_2 &= b_2 + L_{21} b_1 \\
x_3 &= b_3 + L_{32} b_1 + L_{31} x_2 \\
&= b_3 + L_{32} b_1 + L_{31} b_2 + L_{31} L_{21} b_1
\end{aligned}
$$

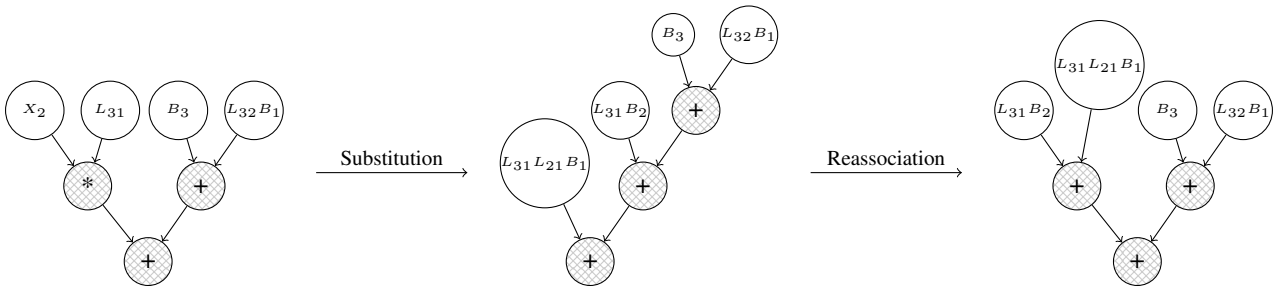

Fig. 3: x_3 row solution dataflow graphs: example of how x_2 expression can be substituted in to break sequential dependencies, and how reassociation of the x_3 expression (only add reassociation shown) can save compute latencies

ply profile for each benchmark after they have been transformed using existing techniques known as substitution and reassociation (Section 3)

- Reducing communication costs via enhanced graph partitioning to exploit locality and dual-channel communication network designs (Section 3)
- Performance results on sparse matrix benchmarks selected from real-world problem domains (Section 4)

II. Background

A. Sparse Matrix Factorization

Many numerical problems are composed of iterative loops where the goal is to solve a set of linear equations $A\vec{x} = \vec{b}$. This is often achieved by performing an LU factorization in every iterative loop. This factorization process is a computational bottleneck in many domains (*e.g.* circuit simulation). We start with the KLU solver [1], which is optimized for fast evaluation of circuit matrices, for our parallelization study. The KLU solver performs a one-time spatial reordering of rows and columns in the matrix at the start of the first iterative loop. This makes it possible for the non-zero structure in the intermediate matrices to remain static for subsequent iterations. This feature allows us to pre-allocate data structures at the start of an iterative phase. This can also help with arranging 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. At the heart of the KLU solver is the Gilbert-Peierels (GP) algorithm, which takes in a sparse matrix A and outputs its L & U factors. In the GP algorithm, a front-solve is called repeatedly in every iteration of the *for*-loop, which is the significant bottleneck in this algorithm. One way to speed up this front-solve is by breaking sequential dependencies using recursive depth-limited substitution, and subsequently rearranging the computation into an efficient compute order through a step known as reassociation.

B. Recursive Depth-Limited Substitution & Reassociation

In this subsection, we describe two existing optimization techniques that are used to reduce sequential dependencies and introduce opportunities for parallelism in highly sequential dataflow graphs of sparse LU kernels.

In Figure 3, we see how we can apply a substitution transformation to a single row solution (from x_2 into x_3) of a front-solve. By recursively substituting expressions in the downstream row expressions, we can transform the entire front-solve such that there are no data dependencies between each row. This is advantageous since, with sufficient parallelism, we can now evaluate each row in parallel. However, the downside to this approach is the increase in computational work, as we are recomputing the same expressions in downstream row solutions. This results in an exponential explosion in the number of nodes and edges, and we typically observe $> 30\times$ growth in the graph size for most sparse benchmarks. To control the work-parallelism balance, a recursive *depth-limited* substitution can be used instead. In this approach, only a fixed number of rows, d_{sub}, are transformed at a time. This reduces the sequential dependencies to between every d_{sub} rows. This is a work-parallelism tradeoff that enables us to match the amount of work with the amount of parallelism we are able to offer.

Despite the reduction in the sequential dependencies and increased potential for parallelism, we need to combine substitution with reassociation in order to achieve compute path latency savings to offset the growth in required work. On applying the substitution transformation, we obtain row solution expressions which are long sums of multiply chains (see expression of x_3 in Figure 3). Due to mathematical associativity, we can rearrange the compute order of these expressions to a more efficient reduction tree structure (see dataflow graph in Figure 3). Due to rounding and truncation errors from finite precision hardware, we must ensure that the final solutions are within an acceptable range of accuracy from the original solutions. We compute the residue of the final solution vector and compare it with the original residue. We find that there is negligible difference, if any, between the two residues, and hence, reassociation transformations are not corrupting the final solutions.

These optimizations on the dataflow graphs are the starting point in our experimental design of heterogeneous networks.

C. Token Dataflow Architecture

A token dataflow architecture was used to do a sparse matrix solve in [2], [3]. In this design, the authors perform a full

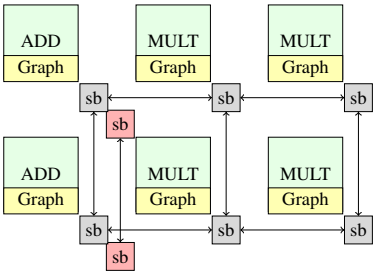

Fig. 4: Heterogeneous Packet Switching Network on an FPGA (2:1 multiply-to-add PE ratio)

unroll of the *for*-loop in the GP Algorithm. The unrolling produced large dataflow graphs that are very irregular and unbalanced, with majority of the compute bottleneck occurring in every front-solve inside each iteration of the *for*-loop. This large dataflow graph was then partitioned across a 2D homogeneous network of processing elements (PEs) and each PE was designed to handle any addition, multiplication and division operations. The PE obeyed dataflow firing rules instead of a procedural design with program counters. Under these dataflow firing rules, each node arriving at the input of the PE is handled independently and asynchronously. An add/multiply/division operation is fired when all the inputs for that instance have been received. The result is then communicated over a 2D network using switchboxes via packets that contain the relevant information. The graph memory in each PE stores information about all these nodes and their fanout node(s) locations. This graph memory is implemented using local BRAMs in each PE. This homogeneous token dataflow architecture [4] is used as a basis for our starting point to construct more complex, heterogeneous and benchmark-customized token dataflow network designs.

III. Heterogeneous Network Design

In this section, we explain the key ideas that allow us to implement a heterogeneous processor and network architecture for dataflow computations. This design better matches our hardware design with the observed compute graphs transformed using substitution and reassociation.

A. PE Design & Placement Challenges

In previous packet-switching NoC designs, each PE is able to handle all types of compute operations observed in the graphs, as they are sharing the same design. Such homogeneous designs are suitable for problems that have balanced compute type requirements (*e.g.* number of additions = number of multiplies). This was the observed case in the original front-solve, as every row had at most one extra multiply and an extra addition/subtraction from the preceding row. However, with the new transformations, we see the number of multiplications increase much more quickly than number of additions. Figure 1 shows the new multiply-to-add ratios in the depth-4 transformed graphs for *bomhof2*. With this information, we can now design heterogeneous packet NoC designs, where each PE can be customized to compute either an add or a multiply and where the frequency distribution of each PE type is related to the multiply-to-add distribution seen in *bomhof2*. Figure 4 shows an example of such a heterogeneous PE design, where the multiply-to-add PE ratio is 2:1.

With this new design, however, graph partitioning and node placement has to be designed to match the requirements of the new heterogeneous designs (*e.g.* only multiply nodes in multiply-PEs). To address this problem, we employ a two-part graph partitioning approach. We first obtain a high-quality homogeneous placement using MLPart-5.2.14. This gives us a high-level placement that minimizes the critical path latency as best as possible for a given PE configuration on a 2D plane. We then do a local, cluster-level, re-placement where all multiply nodes in add PEs are moved to multiply PEs in the same cluster. We define a cluster as a group of PEs that represent the simplest repeating pattern in the heterogeneous design (*e.g.* in Figure 4, PEs *(0,0)*,*(0,1)*, and *(0,2)* are in one cluster). We do the same for add nodes that were initially placed into multiply-PEs. We do this relocation of nodes based on size of each graph memory in each PE, such that all PEs have close to an equal number of nodes to process. Since there are many more multiply nodes than add nodes, we cluster the constant nodes in add PEs as well.

B. Dual-channel heterogeneous network design

In addition to graph transformations, we also customize the communication network of our NoC. Figure 2 shows the add/multiply chain length distribution for a single iteration in the *bomhof2* benchmark, when substituted at depth 4. We observe that long compute latencies are due to the add chains while all multiply chains are relatively shorter, with a maximum of length 5. Hence, we design a new, faster communication channel between add PEs that allow shorter packet communication times when evaluating long add chains. This customization reduces our critical path latency in hardware, since our critical path lies along the long add chains. Figure 4 shows the new communication channel between add PEs, connected by switch boxes colored in red. We route results of the multiplication node on the less critical (slower, more hops) network while the results of the additions are routed on the critical network (faster, fewer hops). This dual-channel design is analogous to the *segment length* concept in FPGA routing.

IV. Methodology

In this section, we describe our experimental setup and methodology to test our new heterogeneous design to the homogeneous design in [2]. We evaluate the performance of both network designs on the same dataflow graphs that have been transformed using substitution and reassociation.

A. Experimental Setup

We develop a software cycle-accurate simulator that enables us to do fast prototyping and testing of our new hardware designs. We utilize the standard Matrix Market format (*.mtx*) as input matrices and develop preprocessor modules to perform substitution and reassociation on the original dataflow graphs.

TABLE I: Benchmark Performance Cycles

Benchmark	Sub.	Configuration					
		FST	PE	Ratio	Cycles	SA	SB
bomhof1	WS	-	4x4	-	1.4m	-	-
	D4	25%	8x8	-	1.09m	1.22×	-
	D4	11%	12x12	1:1	957k	1.32×	1.12×
	D4	9%	12x12	1:2	977k	1.30×	1.10×
	D4	8.6%	12x12	1:3	979k	1.30×	1.10×
bomhof2	WS	-	4x4	-	2.0m	-	-
	D4	84%	8x8	-	1.59m	1.21×	-
	D4	79%	12x12	1:1	1.57m	1.22×	1.01×
	D4	77%	12x12	1:2	1.60m	1.20×	1.00×
	D4	77%	12x12	1:3	1.61m	1.20×	0.99×
simucad	WS	-	4x4	-	5.2m	-	-
	D4	84%	8x8	-	3.65m	1.30×	-
	D4	75%	12x12	1:1	3.27m	1.37×	1.10×
	D4	74%	12x12	1:2	3.34m	1.36×	1.09×
	D4	73%	12x12	1:3	3.42m	1.34×	1.06×

Ratio = Multiply-to-Add ratio, Sub. = Substitution, D4 = Depth 4
WS = Without Substitution [2], SA = Speedups over WS baseline
SB = Speedups over homogeneous networks with substitution optimization

We calibrate our PE design and switching latencies to meet the 250MHz operating frequency target on an FPGA. Due to the substitution transformation, there are significantly fewer division operations than add/multiply operations and hence, we restrict our division operator to just one PE. For this study, we target the Xilinx Virtex-6 SX475T FPGA device. We can fit an 8x8 homogeneous network/PE design configuration, and a 12x12 heterogeneous design configuration (since each PE is now smaller) on this FPGA. The design in [2], in contrast, is only able to fit up to 4x4 configurations on the same FPGA due to each PE having logic to compute all add, multiply and division operations. The target FPGA has approximately 37MB available BRAM memory, which based on a conservative estimate, can fit almost 5 million non-zero floating-point node values. In the benchmarks tested, we need to store at most 3.3 million nodes, and hence, we can fit the entire graph onto the FPGA on-chip memory.

B. Optimal Design Point

We observe that benchmarks tested in this study have a multiply-to-add ratio between 2:1 and 1:1. Hence, we do a design space search over a small set of feasible multiply-to-add ratio configurations – 1:1, 2:1 and 3:1 – to show how performance is affected when different optimal/non-optimal points are selected instead. We kept the depth of substitution fixed at 4. The experiments were repeated at varying PE configurations, valid for each ratio. All the front-solve iterations in the GP algorithm were tested, and we use the MLPart-5.2.14 partitioner in all our two-part placement approach.

V. Results & Discussion

In this section, we present and discuss our observed results. Table I shows the performance results when we evaluate these benchmarks with different heterogeneous network design ratio configurations. We observe a best speedup of 12% and a mean speedup of 8% across all 3 benchmarks that we tested in this study (column SB in Table I). These speedups translate to about 20–40% improvements (column SA in Table I) over work in [2], where neither substitution/reassociation optimizations were used, nor any heterogeneous design concepts were explored.

A. Front-solve transformations (FST)

Small dataflow graphs have insignificant communication costs and by creating extra work from substitution, we incur extra communication costs that may offset any advantage gained from parallelization. Hence, we only implement these optimizations on graphs that are sufficiently large and demonstrate speedup over the original graphs. The percentage of all iterations that are transformed at each configuration is displayed under the FST column in Table I.

B. Multiply-to-Add Ratio

From Figure 1, we note that the preferred ratio configuration for *bomhof2* is closely tied to the multiply-to-add ratio. Most of the iterations lie between 1:1 and 2:1 multiply-to-add ratio spectrum, and we observe best speedups when employing these same 1:1 or 2:1 PE distribution configurations on our heterogeneous network designs. This behavior is also similar for the other two benchmarks, *bomhof1* & *simucad*, tested in this study. Since we evaluate some of the iterations without substitution and reassociation transformations, a more tightly clustered, i.e. 1:1 instead of 2:1, would be naturally preferred as the dataflow graphs are small. This causes the slight performance advantage of using the 1:1 ratio configuration than the 2:1 configuration, observed in this paper.

VI. Conclusions

We show how to accelerate sparse LU factorization on a heterogeneous dataflow architecture by as much as 37% over state-of-the-art designs that are already 3–4× faster than CPU-based solvers. This is achieved through a modification of the dataflow architecture to support heterogeneous processing elements as well as a dual-channel network design that reflect dataflow properties in the LU factorization graphs.

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] N. Kapre and A. DeHon. Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs. In *Field-Programmable Technology (FPT), 2010 International Conference on*, 2010.

[3] N. Kapre and A. DeHon. SPICE2: Spatial Processors Interconnected for Concurrent Execution for Accelerating the SPICE Circuit Simulator Using an FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(1):9–22, 2012.

[4] Siddhartha and N. Kapre. Breaking Sequential Dependencies in FPGA-based Sparse LU Factorization. In *FCCM '14: Proceedings of the 2014 22nd IEEE Symposium on Field Programmable Custom Computing Machines*, pages 1–4, Mar. 2014.