Scalable and Energy-Efficient Reconfigurable Accelerator for Column-wise Givens Rotation

Zoltán Endre Rákossy†, Farhad Merchant‡, Axel Acosta-Aponte†, S.K. Nandy‡ and Anupam Chattopadhyay†
†Institute for Communication Technologies and Embedded Systems (ICE), RWTH University Aachen, Germany
Email: {rakossy,axjacosta,anupam}@ice.rwth-aachen.de
‡CADLab, Indian Institute of Science, Bangalore 560012, India
Email: farhad@cadl.iisc.ernet.in; nandy@serc.iisc.ernet.in

***Abstract*—A new layered reconfigurable architecture is proposed which exploits modularity, scalability and flexibility to achieve high energy efficiency and memory bandwidth. Using two flavors of Column-wise Givens rotation, derived from traditional Fast Givens and Square root and Division Free Givens Rotation algorithms the architecture is thoroughly evaluated for scalability, speed, area and energy. Combining an efficient mapping strategy of the highly parallel algorithms capable of annihilation of multiple elements of a column of the input matrix and using the new features of the architecture, 9 architectural variants were explored achieving a clean trade-off of execution speed versus area, while keeping relatively constant energy.**

***Keywords*—High-level Modeling; ADL Modeling; Coarse-grained Reconfigurable Architecture (CGRA); Orthogonal Transforms; Givens Rotation; 3D Architecture**

I. Introduction

In the last decade, Coarse Grained Reconfigurable Architectures (CGRAs) have received tremendous popularity due to their energy efficiency or performance compared to multi-core general purpose processors, FPGAs and GPUs [1] [2] [3]. While for some CGRAs a mature tool-flow exists to support advanced mapping and scheduling schemes [1] [2], many designs face great challenges with finding optimal mapping via tedious manual processes, leaving throughput maximization and energy minimization untapped. Furthermore, *scalable* CGRA mapping and design remains unexplored, which could hold the key for finding optimal resource trade-offs in a huge design space.

Highly parallel and computationally complex applications such as Numerical Linear Algebra (NLA) kernels represent the perfect application domain to fully tap the advantages of CGRA parallel execution and flexibility. However, with high parallelism also requires high storage access pressure in various patterns, which is one of the limiting factors when seeking efficient execution on CGRAs. Many times, optimal execution is limited by storage bandwidth, or the hardware processing resources do not fit the optimal algorithmic execution window.

On one hand, this can be countered to some extent by redesigning the algorithms such that parallelism is enabled and execution dependency bottle-necks are reduced. On the other hand, a modular and flexible architecture can adapt to the algorithmic requirements gaining performance and efficiency.

In this paper we explore scalability from the architectural and application mapping point of view to reach the optimal energy trade-off versus used resources while maintaining high efficiency. Beside the architectural features, another key component for achieving this is choosing a suitable algorithm that allows sufficient parallelism without dependencies. Column-wise versions of the Square-root Free Givens rotation (CSFG) and Square-root and Division-free Givens Rotation (CSDFG) allow annihilation of multiple elements in a column of the input matrix and are derived from the standard versions of these – SFG [4] and SDFG [5] –, which only zero out one element at a time. Such generalizations were already conducted for the classical Givens rotation algorithm in [6].

The organization of the paper is as follows: After briefly mentioning previous implementations for Givens rotation, we discuss the column-wise algorithms for implementation in Section III. The target architecture is discussed in detail in Section IV and mapping optimizations follow in Section V. Comparisons and results analysis is presented in Section VI, closing with conclusions.

II. Previous Implementations

A $2D$ systolic array implementation of GR [7] on an FPGA platform targeting Virtex-5 XC5VLX220 outperforms one-dimensional systolic implementation of GR and commercially available QRD implementations like QinetiQ and Altera's QRD prior to that. The matrix size of up to 7×7 (954 clock cycles) is supported in single precision arithmetic while sizes up to 12×12 (1412 clock cycles) are supported in 20-bit format. One of the difficulties in $2D$ systolic array implementation is an underlying assumption that $n + \frac{n(n-1)}{2}$ resources are available that is $O(n^2)$ for the matrix of size $n \times n$, limiting scalability. In a practical scenario the resources are $O(k)$ for the matrix of size $n \times n$ where $k \leq n$ and complex controller is required to schedule the matrix elements on the $2D$ systolic array.

The Tournament-based Complex GR targeting MIMO receivers presented in [8] is similar to the scheme presented in [9] where multiple elements of a column of a matrix are annihilated simultaneously by operating on the pairs of rows simultaneously. The triangularization process is accelerated by selecting multiple pivots in one column. For the column of size n, it takes $\lfloor \log_2 n \rfloor$ iterations compared to $n-1$ iterations in the classical scheme. The major drawback of this scheme is not being able to reduce the computational complexity of the algorithm, even though it exploits the parallelism available by operating on the disjoint pairs of rows. Apart from that, this scheme is also unable to exploit available concurrency.

In Modified Squared GR [10], a conventional mathematical operator based approach is adopted over CORDIC-based approach due to area advantage.

The Gauss elimination-based Squared GR [11] is a variant of squared givens rotation [12] targeted at a TMS320C6670 DSP, for high-speed MIMO applications.

REDEFINE CGRA implementation of Q-R decomposition is presented in [13] where the performance is achieved by Custom Functional Unit (CFU) inside Compute Elements (CEs). The focus of [13] is on emulation of systolic schedule for GR on REDEFINE and hence synthesis of systolic array on REDEFINE. The major disadvantage of [13] is the high computational complexity, coupled with high latency problems on REDEFINE's NoC, as matrix size increases.

III. Overview of the Column-wise GR algorithms

In classical GR [4], zeroing out one element is done by applying a *rotation* locally i.e. multiplying with a constructed *Givens matrix* of size 2×2, such that the chosen element – part of a local 2×2 submatrix – becomes zero, then updating the rest

978-1-4799-6016-3/14/$31.00 ©2014 IEEE

of the matrix to compensate for this multiplication and conserve the information of the annihilated element. The construction of this Givens matrix involves square-root and division operations, whereas in SFG [4] and SDFG [5], these architecturally complex operations are omitted by increasing computational complexity by using more additions, subtractions and multiplications. Although this simplifies hardware implementation, it still does not allow a parallel implementation, because a new Givens matrix generation for zeroing out a new element requires completion of the updates of the matrix elements of the affected rows.

In [6] it is shown that by merging the effect of several Givens matrices in a large set of operations, several elements can be zeroed out at once, affecting several rows. The significant advantage of this approach is that this larger set of operations for completing a large rotation is highly parallel, especially the updates on several rows. When mapped onto highly parallel architectures, significant efficiency is gained although the amount of computation is increased.

Without delving into mathematical details, the following example illustrates how the Column-wise versions of the SFG and SDFG work. An interested reader can check the mathematical background in [4] [5] and [6]. The updated matrix Q_1X is shown for SFG and SDFG in Eq. 1 and 2 respectively, after applying cummulative Givens matrix Q_1 zeroing out the subdiagonal elements of the first column of a 4×4 input matrix and the necessary updates for each affected row.

Example: Taking input matrix $X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}$ and applying one iteration of CSFG yields

$$Q_1X = \begin{bmatrix} \frac{p_3}{x_{41}} & \frac{x_{11}x_{12}+s_{11}}{x_{41}} & \frac{x_{11}x_{13}+s_{21}}{x_{41}} & \frac{x_{11}x_{14}+s_{31}}{x_{41}} \\ 0 & x_{12} - \frac{x_{11}}{p_2}s_{11} & x_{13} - \frac{x_{11}}{p_2}s_{21} & x_{14} - \frac{x_{11}}{p_2}s_{31} \\ 0 & x_{22} - \frac{x_{21}}{p_1}s_{12} & x_{23} - \frac{x_{21}}{p_1}s_{22} & x_{24} - \frac{x_{21}}{p_1}s_{32} \\ 0 & x_{32} - \frac{x_{31}}{x_{41}}x_{42} & x_{33} - \frac{x_{31}}{x_{41}}x_{43} & x_{34} - \frac{x_{31}}{x_{41}}x_{44} \end{bmatrix} \quad (1)$$

Similarly, applying one iteration of CSDFG on X yields

$$Q_1X = \begin{bmatrix} p_3 & x_{11}x_{12} + s_{11} & x_{11}x_{13} + s_{21} & x_{11}x_{14} + s_{31} \\ 0 & x_{11}s_{11} - x_{12}p_2 & x_{11}s_{21} - x_{13}p_2 & x_{11}s_{31} - x_{14}p_2 \\ 0 & x_{21}s_{12} - x_{22}p_1 & x_{21}s_{22} - x_{23}p_1 & x_{21}s_{32} - x_{24}p_1 \\ 0 & x_{42}x_{31} - x_{41}x_{32} & x_{43}x_{31} - x_{41}x_{33} & x_{44}x_{31} - x_{41}x_{34} \end{bmatrix} \quad (2)$$

where

$$p_1 = x_{41}^2 + x_{31}^2; \quad p_2 = p_1 + x_{21}^2; \quad p_3 = p_2 + x_{11}^2$$
$$s_{12} = x_{31}x_{32} + x_{41}x_{42}; \quad s_{11} = x_{21}x_{22} + s_{12}$$
$$s_{22} = x_{31}x_{33} + x_{41}x_{43}; \quad s_{21} = x_{21}x_{23} + s_{22}$$
$$s_{32} = x_{31}x_{34} + x_{41}x_{44}; \quad s_{31} = x_{21}x_{24} + s_{32}$$

For an $n \times n$ input matrix, the complexity of the column-wise version in terms of additions and multiplications is as follows, including divisions for CSFG.

$$M_{CSFG} = \frac{2n^3 + 3n^2 + n}{3}; \qquad A_{CSFG} = \frac{4n^3 - 3n^2 - n}{6}$$
$$D_{CSFG} = \frac{n(n-1)}{2} \quad (3)$$

$$M_{CSDFG} = \frac{2n^3 + n}{3}; \qquad A_{CSDFG} = \frac{2n^3 + 3n^3}{6} \quad (4)$$

where M_{CSFG}, A_{CSFG} and D_{CSFG} represent multiplications, additions and divisions in CSFG while M_{CSDFG} and A_{CSDFG} represent multiplications and additions in CSDFG, which has lesser multiplications than the non-column version in [5], while of CSFG the complexity does not change. It is also interesting to note how the p and s terms accumulate over the rows and how these are shared in the rows and columns, creating parallelism and also input data locality.

IV. The *Layers* Architecture

The *Layers* concept is centered around the philosophy of separating computation, communication, memory access and control into dedicated and optimized architectural structures. By specializing parts of the architecture for these tasks, higher efficiency and lower energy can be achieved when compared with standard CGRAs. Moreover, programmability is greatly enhanced by such separation, allowing to exploit data locality, memory access optimization and addition of control flow to a CGRA and thus simplifying the two most important problems in CGRA design: memory bandwidth and programmability.

In this paper, we propose a new architecture designed from the ground up, exploiting high-level exploration and modeling techniques proposed in [14]. Our goals were to provide a modular and scalable architecture, which covers a large portion of the design space and is able to efficiently exploit the parallelism provided by our algorithms described in Section III. The proposed architecture does not make assumptions about memory bandwidth nor is it restricted by a fixed array size for the CGRA and it supports any kind of processing elements, including floating-point arithmetic, opposite to the proposal in [15]. Furthermore, this architecture employs a *reconfigurable control path*, which can be adapted to the control flow of different applications and features an interface for system level integration. With modularity in structure, interconnect and programmable control flow, the architecture can be easily adapted into a domain-specific accelerator, by customizing the generic structures to the target domain.

A. Architectural overview

A detailed view of the architecture is shown in Fig. 1, divided into 4 pipeline stages: pre-fetch, fetch, q-decode and layers. Data flows from left→right (control and configuration, main pipeline) and top↔bottom (layered data flow). The *pre-fetch* and *fetch* stage serve only to forward the instruction word to the *q-decode* stage, where the reconfigurable control path is implemented. In *q-decode*, the current execution state is stored and updated, also data path configurations are decoded and forwarded.

The *layers* stage implements a reconfigurable data path in a waterfall-like manner and is divided into three layers dedicated to each operation class: memory, communication and execution. Each layer can be configured to work at different speed ratios $r(L0 : L1 : L2) = r_0 : r_1 : r_2 | r_0, r_1, r_2 \in 2^n$, to maximize efficiency for each layer for a given application: e.g. $r = 1 : 8 : 4$ is tuned for slow execution, fast communication and medium memory access speed. The control layer speed is always $max(r_0, r_1, r_2)$. Here we use $r = 1 : 8 : 8$, based on the slow fp units and application requirements.

The topmost layer (*SoC*) implements an interface which allows system level integration and control, details of which are out of the scope of this paper. The architecture is completely described in the high-level LISA architecture description language [16], which allows automatic generation of RTL code and a set of simulation and programming tools.

B. The computation layer (L0)

L0 is comprised of a scalable and customizable square array of size $N \times N$ of processing elements (PE) interconnected with a mesh network of configurable bit-width (other geometries possible). Each PE has its own pipeline, is replaceable and modular in design, allowing the designer to plug in custom RTL

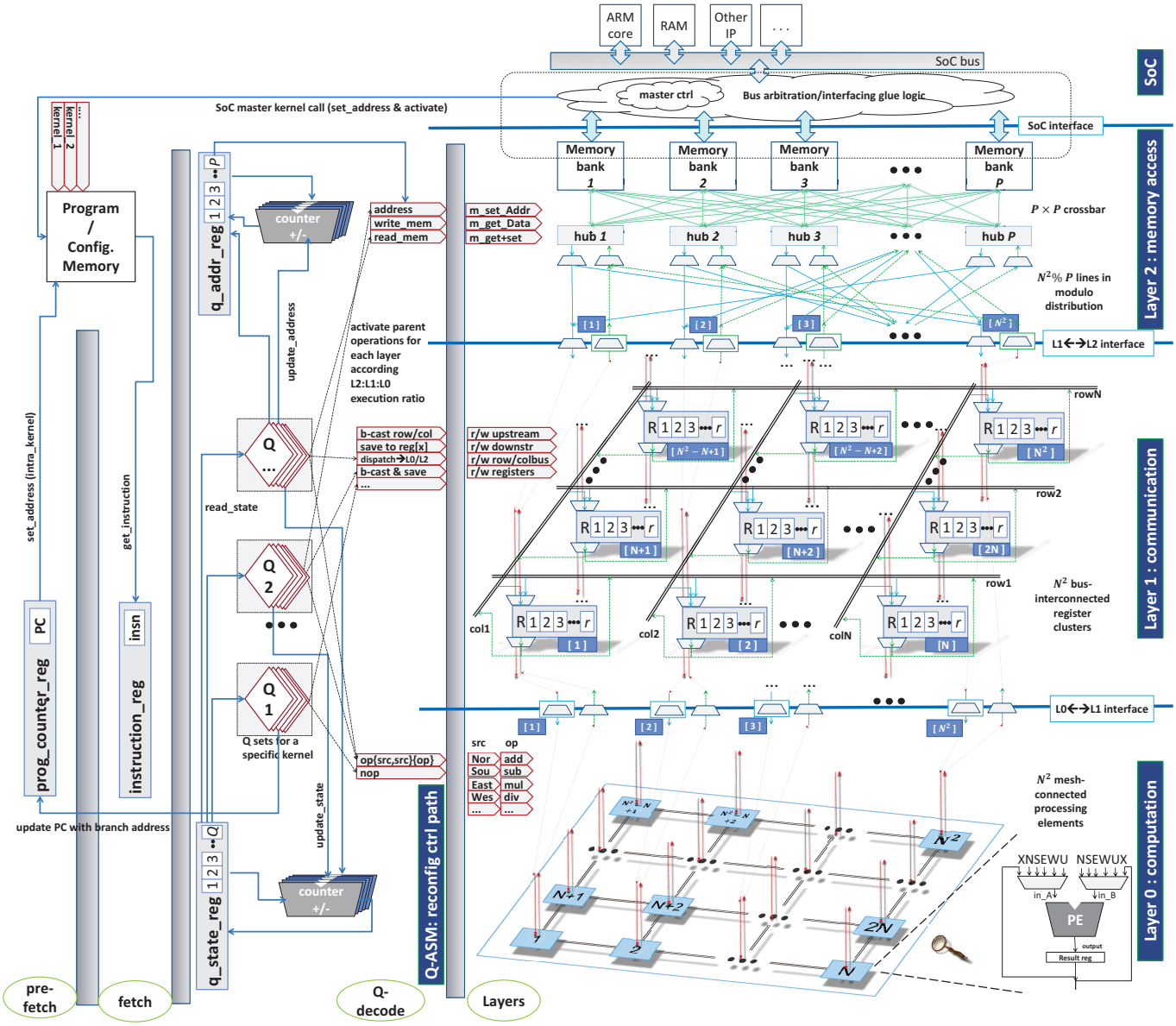

Fig. 1. The *Layers* architecture: scalable and modular layers dedicated for memory access, communication and computation are managed by a reconfigurable control flow stage.

components. PE capability is defined in a group of *operations* while the interconnect provides a number of *sources* to each PE input port. Operations and sources are encoded in groups to create an access interface to the *q-decode* stage.

$$op(PE_n) = \{+, -, *, \dots\};$$
$$src(PE_n^{ports}) = \{North, South, East, West, Up, Self, \dots\};$$
$$n \in \overline{1..N^2};$$

In this paper we use DesignWare floating point (fp) modules provided by Synopsys, $op(PE_n) = \{+, -, *\}$. One pipelined fp divider is added to `PE0` for the architecture targeted at the CSF algorithm, providing one 32-bit fp division result in 4 cycles. Each PE reads input data from 6 sources for each input port $src(PE_n^{a,b}) = \{N, S, E, W, U, X\}$ and outputs results into a register.

C. The communication layer (L1)

The main role of L1 is to serve as a staging area and preparation network for the input data of L0 coupled with transporting results from the L0 result registers upstream. It is organized in register clusters of parameterizable size for each of the N^2 elements, which are interconnected by two bus-like structures topologically on row and column. Additionally, the upstream interface to L2 and the downstream interface to L0 are buffered and act like a pipeline register between the layers. Formally, in every cycle, each L1 cluster can perform a combination of core operations of the form $op(targets)$, with the condition that it does not violate architectural laws (e.g. creating loops, double write to same target, etc):

$$op = \{read, write, nop\}$$
$$targets = \{downstream, upstream, rowbus, colbus, reg\,[k]\}$$

Using these core operations, useful constructs can be grouped and made available at the q-decode stage: e.g. $save(downstr\,[reg]\,, upstr\,[reg]\,, rbus\,[reg]\,, cbus\,[reg])$, where each `reg` can/must be a different register from the cluster allowing several simultaneous reg-writes from downstream, upstream and buses; $rowbc{+}save(downstr\,[reg]\,, upstream\,[reg]\,, cbus\,[reg])$, where one of downstream, upstream or column bus are

selected exclusively as a source by getting a non-zero `reg` parameter, broadcast onto the row bus, while taking the parameter as the register index where the source is to be saved. Such compound operations can be application-tailored or one can use a default set. The buses are access-guarded wires with variable cluster span. In case of large arrays, long-distance and short-distance bus structures can be added as necessary.

D. The memory access layer (L2)

This layer provides access structures to a variable number of memory ports P and is used to distribute these ports to N^2 L1 structures downstream. Distributing data across a number of memory ports allows for higher load/store bandwidth. To avoid the necessity of a full crossbar from N^2 elements to P ports and ensure scalability, P *hubs* are introduced in-between. Thus each hub has access to each port, needing only a $P \times P$ crossbar, which is reasonable since usually memory ports are scarce ($N^2 \geq P$). From the hubs, a static modulo $N^2\%P$ distribution is employed, uniformly distributing hubs across downstream elements, e.g. if $PE(n)\%P = 0$, the n-th PE is connected to hub 0, each hub having $\lceil \frac{N^2}{P} \rceil$ connections. Another role of the hub is to use the memory's protocol to forward access requests and select the correct port based on the desired data address. A reduced set of possible operations are formed, such as *setAddr, getData, getD+setA, setA+setD* and made available at the q-decode stage.

E. The control layer (Q-ASM)

The q-decode stage implements the control layer of the architecture. Using *qualifier*-based Algorithmic State Machine (ASM) concepts proposed in [14], the control flow of our algorithms are implemented, also providing a configuration decode framework for assembly programming. Qualifiers implement control flow components such as `for` loops, `if-else` branches and keep the current execution state in q_registers. Formally, q-ASM-based control flow can be reduced to a set of components, which when combined, realize the state machine of the algorithm. The q-decode stage is comprised of two sets of registers and counters and a set of algorithm-specific q-operations which combine the resource sets. The q-operations also *activate* the macro-structures formed by the elementary operations in the layers pipeline stage for each layer, effectively controlling execution, interconnect and memory.

$$q_{addr} = \{q\_addr\_regs(\overline{1..P}), q\_addr\_counters(\overline{1..P})\}$$
$$q_{state} = \{q\_s\_reg(\overline{1..q}), q\_s\_count(\overline{1..m}), q\_compare(\overline{1..k})\}$$
k, m, q are definable parameters

At this stage the instruction/configuration word is formed, for which an assembler and linker is automatically generated by the LISA tools, with arbitrary syntax defined in each macro-op.

$$qual(param)|L2op(1..P)|L1op(1..N^2)|L0op(1..N^2)$$

Based on the ratio r of the architecture, partial loading of the instruction word can be optimized, e.g. L0 fetches only every r_0 cycles although at this time LISA tools do not support multiple program memories. A set of $s = max(r_0, r_1, r_2)$ instructions is called a *macro-cycle*, where L2 and L1 prepare the data for one L0 cycle of execution, enhancing programming and algorithm mapping.

V. Mapping the Algorithms

Application mapping still remains one of the most challenging problems when designing and using CGRAs. For *Layers*, mapping is greatly enhanced by two factors: 1) algorithm/hardware co-design for scalability and 2) high-level design space exploration and tools generation. We consider a

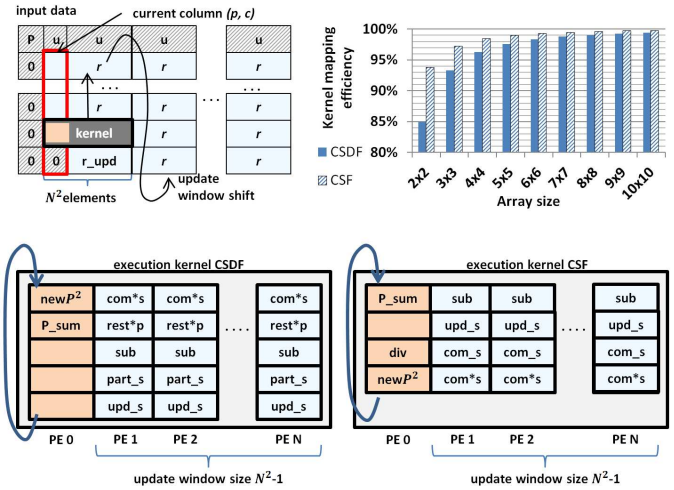

Fig. 2. Mapping CSDFG and CSFG: algorithm execution over input data (top left), kernel mapping (bottom) and mapping efficiency (top right).

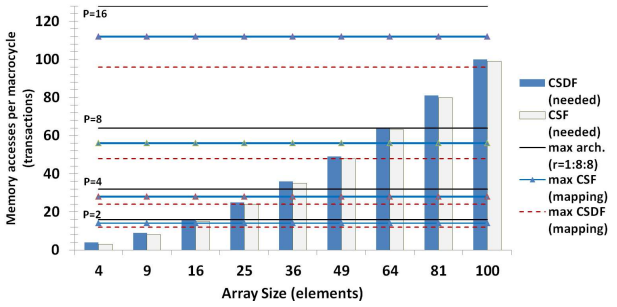

Fig. 3. Architectural memory bandwidth limits, mapping-based limits vs. actual algorithmic bandwidth requirements with varying N^2 and P.

mapping *optimal* when all the computational resources are fully utilized every cycle with meaningful tasks.

This is hard to achieve, since CGRAs are inherently starved for data and often application dependencies in data and control flow or architectural properties limit our options (e.g. fp divider has 4 cycles latency). The derivation of the mapping took into consideration several architectural and algorithmic parameters, such as available bandwidth per cycle, architectural computation:memory speed ratio, array size, dependency, common data and progress through the data. Based on the ratio r, ports P and size N^2, the available memory bandwidth and data requirements per cycle change. Either of these parameters can be used as the optimization target when mapping. *Layers* allows smooth scaling for exploring the optimal point for a given application, trading off mapping optimality and parallelism for resources.

Carefully analyzing CSDFG and CSFG, we could identify terms with special properties: "p", "common", "rest" and "s"-terms. For each zeroed element, its row must be updated with effect of current previous zeroed elements, accumulated in the "p" and "s" terms, which required addition of squared terms for "p" and accumulation of one multiplication for "s". During the calculation of partial "p" and "s" terms, data is broadcast on the row, which represents the "common", and is consumed together with the local term for each column, representing the "rests".

Fig. 2 shows how the algorithm progresses through the input matrix and the mapping of the kernels of each algorithm and their respective efficiency. We used modulo-scheduling concepts to unroll the bottom→top kernel progress over one column to extract the common data points and pipelined the computation such that all PEs are busy, independently. Each PE is responsible for one column vector, allowing scalability with the window size $N^2 - 1$, reserving one PE to execute

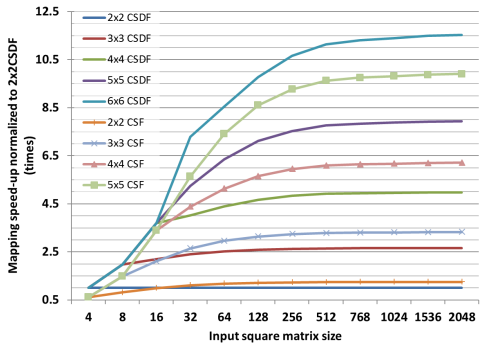

Fig. 4. Cycle speed-up gained when using a larger array.

computation for "p" and division in case of CSF. In the kernel for both algorithms, the "common" and "rests" terms are used twice, once for "s" calculation and once for the updates, in subsequent rows, allowing us to temporarily save these terms in L1 and distributing them as needed. This saves on one hand memory load bandwidth, but also the memory read latency on the other hand. Both algorithms use no more than 4 registers per PE in L1, the current architecture being configured for up to 7 L1 registers.

The scheduling was optimized for L0 PEs in 5 macro-cycles for CSDFG and 4 macro-cycles for CSFG, taking the array size as a parameter achieving $> 99\%$ mapping efficiency for array sizes $N > 5$, and $> 90\%$ for $N = 2..4$, excepting the 2×2 CSDFG for which only 85% could be achieved, as shown in Fig. 2. Optimal usage could not be achieved without breaking regularity and scalability. The first PE recalculates the "p" terms for each update window shift, since storing input matrix column size number of "p" terms would be infeasible for large matrix sizes. In case of CSFG the divider latency (4 cycles) could be pipelined with the calculation of "p" terms.

Once the scheduling is defined, necessary bandwidth can be calculated for every macro-cycle and mapped against available bandwidth for varying combination of P, N^2 and r, eventually trading off mapping efficiency. Fig. 3 shows available bandwidth, required algorithm bandwidth for $N = 1..10$, $r = 1 : 8 : 8$ and maximum mapping efficiency. Lower r would require more than $2 \times P$, or would halve the mapping efficiency, introducing NOPs. Thus, any of the variables can be used as a constraint, from which others can be derived, allowing a truly scalable mapping and resource trade-off.

VI. Evaluation and Results

A. General considerations

Layers has been coded completely in the LISA ADL of Synopsys Processor Designer, completely parametrized, then customized for the two algorithms with the necessary PEs and qualifier parent operations. Simulations have been conducted for random non-singular square input matrices of size 4..2048, for different combinations of $P = 2..16$ and $N = 2..6$, values limited by the current 32-bit implementation of the LISA tools when it exceeded the 4GB memory usage, at $N = 7$ for CSDFG and $N = 6$ for CSFG or input data size $>$2048. 64-bit binaries of the Synopsys tools would remove this non-architectural limitation. The assembly programs for each algorithm have been coded in a scalable way, encompassing 17 macro-cycles for CSDFG and 28 macro-cycles for CSFG including prolog and epilog of the kernels.

For these configurations RTL code has been generated and synthesized with DesignCompiler H-2013 for Faraday $65nm$ technology library, using PowerCompiler and backward switching activity files for power estimation. Using more ports than the minimal amount yielded greater power and area usage with no advantages, the power gained from emptier macro-cycles did not compensate for the power and area used for additional structures, therefore we omitted those results for clarity. A speed-up of $11\times$ can be gained by increasing N shown in Fig. 4 for each algorithm.

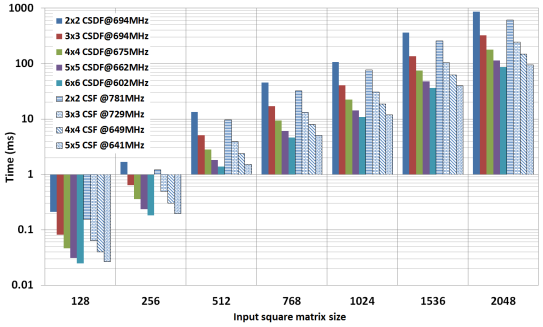

Fig. 6. Time in ms required for different input matrix sizes by each architecture/algorithm. Different configurations span an order of magnitude.

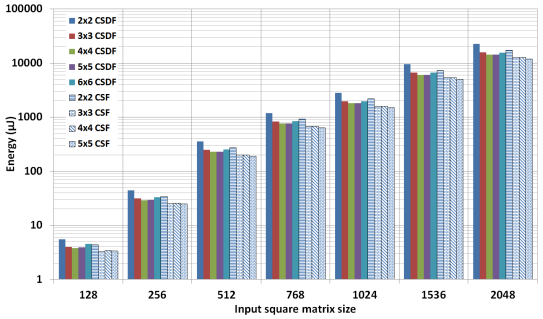

Fig. 7. Similar amount of energy in μJ required for different input matrix sizes by each architecture/algorithm. Smaller matrix sizes omitted for clarity.

B. Time and energy

Very interesting results are provided in Fig. 6+7, where the overall time and energy values are depicted for each configuration and input matrix size. Execution time spreads over several orders of magnitude with varying input data size, while an order of magnitude speed-up can be maintained between the smallest and largest array for large input data sizes. In the case of CSFG, a smaller N achieves about the same execution time as a larger one for CSDFG. In terms of energy, it is very important to note how the overall energy remains comparable for a given input matrix size, letting designers to directly trade off execution speed with area, without worrying about energy. Here, CSFG remains superior for large input data, needing $\approxeq$10-20% less energy than CSDFG which is more suitable for small matrix sizes, breaking even around 32×32.

C. Comparisons with related work

Table I and Fig. 5 summarize the results and provides some comparison data. The frequency of each architecture is limited by the control flow complexity in the q-decode stage for small N, and by L1 critical path for larger N at $r = 1 : 8 : 8$. Choosing lower r, the fp PEs limit overall frequency, while sacrificing memory bandwidth, especially when requiring a divider. When comparing to the clean version of *Layers*, it is interesting to note that the reconfigurable control path slows the architecture down, although by not operating near maximum frequency has great advantages in power consumption. In the case of CSFG, the simpler control path allows a higher operating frequency and when considering the fact that the area does not differ significantly from CSDFG, makes the CSFG version superior.

TABLE I. COMPARISON WITH SIMILAR ARCHITECTURES IN THE LINEAR ALGEBRA DOMAIN

Architecture	Area (kGE or mm^2)	Frequency (MHz)	Power (mW)	GFlops/W	GFlops/mm^2	TechLib
30 LAC cores (sim. estim.) [17]	115mm^2	1400	N/A	30-55	6-11	45nm
2×2 Layers r=1:8:8 (clean)	84kGE (0.12mm^2)	990	17.74	27.95	5.84	65nm
2×2 Layers r=1:4:4 (clean)	84kGE (0.12mm^2)	990	17.74	55.91	11.68	65nm
2×2 Layers r=1:2:2 (clean)	84kGE (0.12mm^2)	990	17.74	111.83	23.37	65nm
GSGR on DSP [11]	N/A	1200	9313	8.24	N/A	40nm
QR Decomp. ASIC [18]	36kGE	278	48.2	N/A	N/A	130nm
Lin. solver (120 PEs) [19]	N/A	200	18000	2.61	N/A	65nm, FPGA
Column-GR on REDEFINE [6]	N/A	374.1	N/A	N/A	N/A	90nm
Systolic array [7]	N/A	139	N/A	N/A	N/A	65nm, FPGA

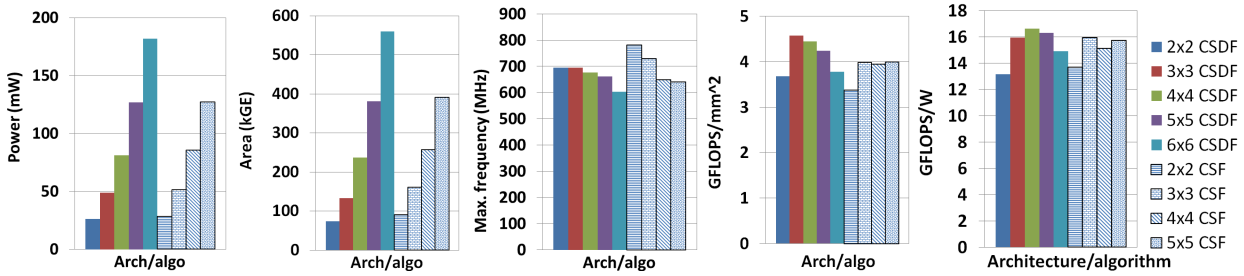

Fig. 5. Power, area, maximum frequency, GFlops/mm^2 and GFlops/W for the 9 *Layers* architectural variants tailored for Givens rotation.

Comparing with other architectures is difficult for a variety of reasons, mainly because of different algorithms and platforms, making comparisons often unfair. In the linear algebra domain, the LAC CGRA [17], shows impressive numbers, but lacks application information for a comparison and heavily relies on estimates for the complete system, whereas our implementation shows post-simulation and post-synthesis results. In [11], details about a 64×64 inversion with GSGR on a modern DSP platform are discussed, requiring 2.2ms, which is two orders of magnitude slower than the respective triangularization on the slowest variant of our architecture, however the former handles inversion. Comparing with T-GR [8] for a 8×8 matrix the execution time of its FPGA implementation at 51.9MHz yielded 0.000365ms which is 5.8× slower than the slowest CSFG version. In the 2D-systolic design [7] a 12×12 matrix is processed in 0.0101ms on an FPGA implementation at 139MHz, which is 25× slower than execution of a 16×16 matrix in the slowest of our designs.

VII. CONCLUSIONS

Exploiting high-level design methodologies, in this paper we meticulously explored the design space and highlighted the advantages of scalable algorithm-hardware co-design over 9 architectural variants for two column-wise versions of Givens Rotation algorithms. Besides providing excellent trade-off space for area, speed and energy, we highlight the advantages of a programmable and scalable CGRA architecture and high-level design methodologies. In the future, we seek to create a library of highly optimized, scalable linear algebra kernels and provide SoC integration and (3D silicon) place&route details.

REFERENCES

[1] M. Alle *et al.*, "REDEFINE: Runtime reconfigurable polymorphic ASIC," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 2, pp. 11:1–11:48, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1596543.1596545

[2] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array," in *Reconfigurable Computing: Architectures, Tools and Applications*. Washington, DC, USA: Springer Berlin Heidelberg, 2007, vol. 4419, pp. 1–13.

[3] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[4] G. H. Golub and C. F. Van Loan, *Matrix computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[5] J. Götze and U. Schwiegelshohn, "A square root and division free givens rotation for solving least squares problems on systolic arrays," *SIAM J. Sci. Stat. Comput.*, vol. 12, no. 4, pp. 800–807, May 1991. [Online]. Available: http://dx.doi.org/10.1137/0912042

[6] F. Merchant, A. Chattopadhyay, G. Garga, S. K. Nandy, R. Narayan, and N. Gopalan, "Efficient QR Decomposition Using Low Complexity Column-wise Givens Rotation (CGR)," in *VLSI Design*, 2014, pp. 258–263.

[7] X. Wang and M. Leeser, "A truly two-dimensional systolic array FPGA implementation of QR decomposition," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, pp. 3:1–3:17, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1596532.1596535

[8] M.-W. Lee, J.-H. Yoon, and J. Park, "High-speed tournament givens rotation-based QR Decomposition Architecture for MIMO Receiver," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, may 2012, pp. 21 –24.

[9] M. Hofmann and E. J. Kontoghiorghes, "Pipeline Givens sequences for computing the QR decomposition on a EREW PRAM," *Parallel Computing*, vol. 32, no. 3, pp. 222 – 230, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819105001638

[10] L. Ma, K. Dickson, J. McAllister, and J. McCanny, "QR Decomposition-Based Matrix Inversion for High Performance Embedded MIMO Receivers," *IEEE Trans. Signal Process.*, vol. 59, no. 4, pp. 1858–1867, 2011.

[11] Fonseca, Ma Nilma and Klautau, Aldebaro, "High speed GSGR matrix inversion algorithm with application to G. fast systems," in *Microwave & Optoelectronics Conference (IMOC), 2013 SBMO/IEEE MTT-S International*. IEEE, 2013, pp. 1–5.

[12] R. Döhler, "Squared givens rotation," *IMA Journal of numerical analysis*, vol. 11, no. 1, pp. 1–5, 1991.

[13] P. Biswas, K. Varadarajan, M. Alle, S. K. Nandy, and R. Narayan, "Design space exploration of systolic realization of QR factorization on a runtime reconfigurable platform," in *Embedded Computer Systems (SAMOS)*, 2010, pp. 265–272.

[14] Z. E. Rákossy, A. Acosta Aponte, and A. Chattopadhyay, "Exploiting architecture description language for diverse IP synthesis in heterogeneous MPSoC," in *Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 2013, pp. 1–6.

[15] Z. E. Rákossy, T. Naphade, and A. Chattopadhyay, "Design and analysis of layered coarse-grained reconfigurable architecture," in *Reconfigurable Computing and FPGAs (ReConFig)*, 2012, pp. 1–6.

[16] "http://www.synopsys.com/Systems/BlockDesign/processorDev", [Online] Synopsys.

[17] A. Pedram, R. A. van de Geijn, and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Trans. Comput.*, vol. 61, no. 12, pp. 1724–1736, 2012.

[18] Mahdi Shabany and Dimpesh Patel and P. Glenn Gulak, "A Low-Latency Low-Power QR-Decomposition ASIC Implementation in 0.13um," *IEEE Trans. Circuits Syst.*, vol. 60-I, no. 2, pp. 327–340, 2013.

[19] W. Zhang, V. Betz, and J. Rose, "Portable and scalable FPGA-based acceleration of a direct linear system solver," *ACM Trans. on Reconf. Techn. and Sys.(TRETS)*, vol. 5, no. 1, p. 6, 2012.