Design and Analysis of Layered Coarse-Grained Reconfigurable Architecture

Zoltán Endre Rákossy[†], Tejas Naphade[‡] and Anupam Chattopadhyay[†]
[†]*Institute for Communication Technologies and Embedded Systems (ICE), RWTH University Aachen, Germany*
Email: {rakossy,anupam}@ice.rwth-aachen.de
[‡]*Department of Electrical Engineering, Indian Institute of Technology Bombay, India*
Email: tejasnaphade@iitb.ac.in

***Abstract*—Coarse-grained reconfigurable architectures (CGRAs) represent an important class of programmable accelerators with a significant performance advantage for data-driven, systolic algorithms. In this paper, we present a novel CGRA where data access, data transport and execution are separately layered into dedicated, independent structures. The proposed architecture concept allows for independent control and optimization on each layer to address the storage access bottleneck, faced by state-of-the-art CGRAs. The architecture is programmable and the implementation is derived from a high-level language specification, allowing fast design exploration, debugging and simulation. Up to 50% run-time performance improvement and 5× area-time-energy product gain of the layered CGRA over a non-layered one is demonstrated with 2 case studies from demanding linear algebra applications.**

***Keywords*-Coarse-Grained Reconfigurable Architecture (CGRA); LU decomposition; Matrix Multiplication; Layered Architecture;**

I. Introduction

The increasing complexity of embedded applications and rising manufacturing costs prompted designers to look for architectural solutions, which offer both flexibility and performance. Application-specific accelerators are increasingly used to balance the performance-flexibility trade-off and increasingly used in modern System-on-Chips. Coarse-Grained Reconfigurable Architectures (CGRAs) represent an important class of application-specific accelerators, which are shown to be efficient [1]. In contrast to general-purpose Field-Programmable Gate Arrays (FPGAs), CGRAs are designed specific to application(s) with dedicated functional units, application-specific bit-width and interconnects. Numerous CGRA designs have been proposed in recent times, both in academia [2][3] and industry [4][5]. Despite significant advancements in the CGRA design, there are several major limitations to its widespread acceptance. Two serious challenges for CGRA design are **1)** to increase the utilization of the parallel functional units, one needs to provide data at a very high bandwidth which is not always possible due to cost or architectural limitations and **2)** to provide stable tools for modeling, programming, simulation and debugging.

We address both the aforementioned challenges in this paper. For the first challenge, a layered CGRA design is advocated. The multiple, independent layers allow for increasing the data bandwidth by simultaneous execution and data transport. In *data computation* layer, a processing element (PE) array with customizable interconnect will deliver the means for execution of parallel but also sequential computation. The *communication* layer manages, transports, times and stores any data on the way to or from the data computation layer, aiming to keep it under maximum load. *Memory access* layer and *control flow* layer are responsible for managing memory loads and stores in the most efficient way coupled with fine-grained layer component activation to implement application control flow.

We tackle the second challenge by a high-level modeling and exploration framework for CGRA. While such exploration platforms have been proposed already in the literature, we differentiate by coupling with a commercial high-level accelerator design framework. This not only provides us with a ready tool-flow for simulation and implementation, it also makes it possible for a wider research community to exploit the same flow. As this paper focuses on the architectural aspects of the first challenge, we will not delve into details of the high-level modeling here.

The paper continues with discussing related work, then introduces target application details and mapping in Section III. Section IV provides architectural details of the proposed multi-layered CGRA. Evaluation is conducted in Section V followed by conclusions.

II. Related Work

The performance and flexibility advantage of CGRAs has been long established. ADRES[2] views CGRA as an extension to a VLIW front-end, where control-intensive parts are executed on the VLIW part and compute-intensive applications are mapped on the CGRA by use of modulo-scheduling. Another view on CGRA modeling focuses on compilation techniques for both computation and data aspects of CGRAs mapping[6]. rASIPs (reconfigurable Application Specific Instruction-set Processors) are designed by use of a high-level architecture description language helping to co-design of the base processor, the reconfigurable fabric and its interface in a unified manner [7], resulting in a synthesizable RTL code and configuration bitstream. These approaches avoid having the assembly programming view, delivering ready configurations or RTL from high level input, but the importance of programmability pushes companies to address this problem, too[8][9].

The promised performance of CGRA remains elusive without having sufficient bandwidth. Early experiments on data coherence and leveraging parallelism of CGRAs were conducted in [10]. The ADRES design relied on a wide-issue VLIW architecture while allowing the CGRA to access registers and the memory hierarchy directly. An Integer Linear Programming (ILP) modeling of CGRAs has been advocated in [6][11] to make the application mapping on CGRA with high data locality. Many CGRA designs are proposed to increase data locality by providing dense interconnects [12] or by providing local registers inside the processing elements [4]. While the programming and mapping solutions fail to guarantee optimality, the architectural solutions compromise area-efficiency in order to reduce the execution runtime. Despite these efforts, maximizing CGRA efficiency to exploit parallelism remains an important research challenge. We explore an alternative architectural solution by orthogonalization of data access and computation problems.

The Layers approach is similar in principle to the clustered VLIW architecture, which promotes data locality among parallel functional units. Similarity can also be drawn between architectures with multiple configuration[13] or physical[14] layers, or the ideal RISC machine, the Transport Triggered Architecture (TTA) [15], which directly *moves* the operands among the input and the output ports of different functional units. In Layers, data communication layer makes use of similar data movement mechanisms as TTA, but instead of isolated PEs and central buses, the underlying execution structure is a true mesh-connected array allowing a wider movement range. Furthermore, the basic processing elements in our approach are reconfigurable as in a CGRA.

Lately, designs motivated towards multi-layered computing have also started to emerge [16], in order to exploit the capability of 3D IC manufacturing. To the best of our knowledge, this is the first work towards the feasibility and performance analysis of a multi-layered programmable CGRA, however there have been experiments with layered configurations of FPGAs before[13].

III. Application Mapping

This section defines architectural constraints and presents pre-implementation algorithm analysis and mapping.

A. Architectural Constraints and Algorithm Analysis

In this paper we explore two 32-bit fixed-point (Q-format) CGRA-based architectures at high-level, with a set of constraints typical to embedded systems, in order to trigger typical design problems. Under given constraints maximum performance evaluation of the two design approaches is targeted. As a case-study application, we chose block LU decomposition of arbitrary matrix size from the family of dense linear algebra, which makes CGRAs a perfect implementation target, given the large amounts of parallelism in the algorithm.

We targeted a maximum area of 500kGE which is a usual constraint for embedded accelerators, translating into one divider and at least a 4×4 PE array. We limited storage to four 32-bit wide memory banks with two combined read/write ports.

B. Application mapping: Matrix multiplication

Matrix multiplication is a high complexity but parallelizable core component for many linear algebra algorithms. Efficient mapping and execution plays a critical role for applications that use it. For our architecture, we draw inspiration from optimal mappings for systolic arrays[17][18][19] and memory partitioning and partial storage for FPGAs [20].

Optimal execution of a square matrix multiplication of size n on $n \times n$ array with Multiply-accumulators is n cycles as shown in Fig. 1. *Optimal load access* can be achieved with this mapping, every element being loaded only **once**. If the PE array size ($n \times m$) is smaller than input matrix size ($i \times j, j \times k$), saving input and output data in FIFO registers reduces data re-load necessity. In Fig. 1 an example for 4×4 inputs on a 2×2 case is depicted. PE array slides over the first cube slice over 4 cycles in the given order, $m \times \frac{k}{n}$ (i.e. 2) registers store input b and $\frac{i}{m} \times \frac{k}{n} \times m \times n = i \times k$ (i.e. 16) registers store the output. Input b is reused from FIFO in cycles 3,4 while a is used for 2 cycles. Outputs are saved every cycle to FIFO, and are accumulated every slice. Load optimality can be traded off with storage, by choosing the direction in which the PE plane slides through the slice or cube to minimize FIFO storage, especially for non-square inputs. It is important to note that left and right matrix multiplication have different memory access patterns, which can stress memory access in an uneven way or can violate access constraints.

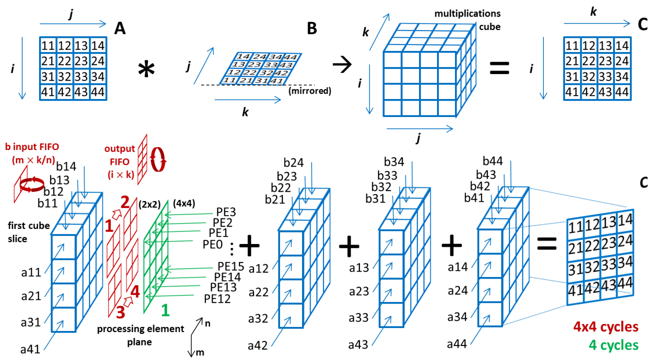

Figure 1. Matrix multiplication mapping on a MAC capable processing element array. If the PE array is smaller than matrix size, FIFO registers reduce data reload.

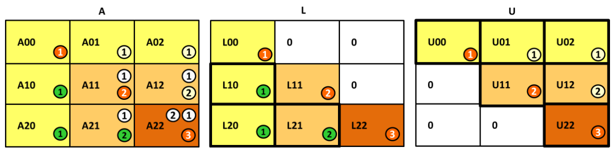

Figure 2. Block LU progression through $3q \times 3q$ matrix. Red circles = S1, green = S2L, pale yellow = S2U, white = S3, iteration number inside.

C. Application mapping: LU decomposition

LU decomposition is important in solving linear equation sets of type $Ax = b$, where nonsingular $n \times n$ matrix A is decomposed to an upper (U) and lower (L) triangular matrix such that $Ly = b, Ux = y; \Rightarrow Ax = LUx = Ly = b$. For large matrix sizes, the block LU algorithm splits the matrix into small blocks of $q \times q$ where the basic LU algorithm can be directly applied (Fig. 2). Only relevant parts for architectural mapping are presented, leaving LU decomposition details to excellent references[21].

The block LU algorithm can be split into 3 stages, and the progression is shown in Fig. 2 for a $3q \times 3q$ case. Each stage has a different computational core, which requires different types of architectural optimizations. The basic LU algorithm is similar to the block LU shown below, with the difference that blocks are replaced by elements and matrix inversion is replaced by division.

```
Block LU algorithm:
for ($x = 0$; $x < \frac{n}{k}$; $x$++)
[S1: update main diagonal]
    apply LU: $A_{xx} = L_{xx}$ and $U_{xx}$
    calculate $L_{xx}^{-1}$ and $U_{xx}^{-1}$
[S2U: update current row]
    $U_{xi} = L_{xx}^{-1} \times A_{xi} \mid i > x$
[S2L: update current column]
    $L_{jx} = A_{jx} \times U_{xx}^{-1} \mid j > x$
[S3: update all sub-blocks]
    $A_{ij} = A_{ij} - (L_{jx} \times U_{xi}) \mid i, j > x$
```

For S1, division becomes critical when applying LU on the $q \times q$ block, however due to area constraints and data dependency multiple dividers can not be considered. Thus, the LU part has been completely unrolled and highly pipelined. Moreover, inversion of the resulting L and U triangular

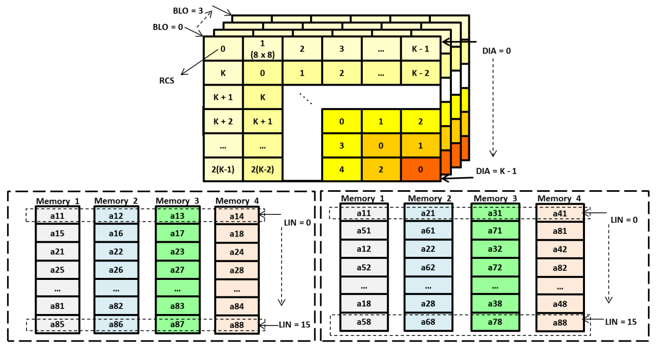
Figure 3. Global memory organization tailored for 8×8 block on a 4×4 PE array, with straight/transpose block detailed.

matrices is also sequential and requires division. Inversion of the $q \times q$ triangular matrices is further split into two $\frac{q}{2} \times \frac{q}{2}$ inversions and multiplications, partly parallelizable. A tight S1 scheduling is critical for maximizing resource use and performance, but with increasing block size q, manual optimization is limited by complexity. S2 and S3 rely on $q \times q$ matrix multiplications.

Fig. 3 depicts memory organization. Input matrix is divided into blocks of size $q = 8$, sub-diagonal blocks are stored in a transpose format to those above the diagonal, to accommodate left and right matrix multiplication load patterns, which alleviates the bank-balancing problem presented in [11]. BLO offset determines starting addresses of A, L and U for simplicity, but A can be overwritten with the resulting L and U. DIA offsets each step in the main iteration, and points to the first block on the diagonal of the sub-matrix during block LU execution. RCS offsets the block within the current row/column in S1 while LIN points to the 1×4 element half-row of an 8×8 block. One block is distributed across 16 LIN addresses and 4 memory banks. S2 and S3 parallel execution is limited only by the memory bandwidth of the modules. 4×4 PE array with $q = 8$ is chosen due to smaller block size yielding low PE array load. Higher block size violates the 2 accesses per bank constraint. One $q \times q$ matrix multiplication requires $\leq 2q$ concurrent loads every cycle, more if next matrix load overlaps with stores.

IV. The *Layers* Concept and Architecture

The central philosophy of the Layers concept is to design an architecture with structures dedicated to *data computation (L0)*, *data communication (L1)*, *memory access (L2)* and *control flow (SA)*. As such, a layered architecture aims to gain high energy efficiency and performance by using most suitable structures for each of the above categories. In our view, an architecture is *efficient* when computational resources for the respective task have near-optimal load. To attain this efficiency, the remaining structures must provide all the data it requires, at the right time, by whatever means necessary. To highlight the flexibility and advantages of layers, we have also created an architecture devoid of the communication layer at the end of this section.

1) ***The architecture:*** The complete architecture is shown in Fig. 4, divided into 4 parts: (SA) and (EX).L0, (EX).L1 and (EX).L2, and it is completely defined in the LISA language [22] using the complete simulator, debugger, assembler, RTL description generator flow. (SA) and (EX) members are divided by the pipeline register forming a 2-stage pipeline, while the layers are divided by an intra-layer interface, with upstream and downstream channels.

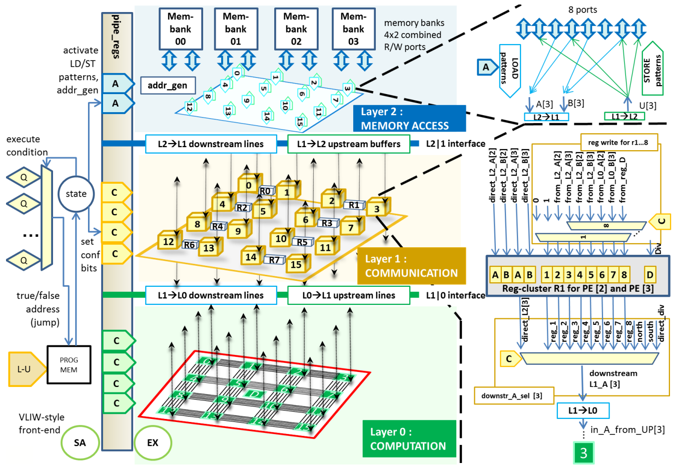
Figure 4. The layered architecture with a zoom-in on the communication and memory layer.

The EX stage is in complete sync, intra-layer and inter-FU communication always passes at least one register, as detailed at the end of the section.

2) ***Control flow:*** The "state automaton" (SA) stage incorporates program control, configuration bit setting and execution state supervision. (SA) is based on a finite state machine coded via several *qualifiers* for next state decision, which is loaded from the program memory in the form of a very long instruction word.

Thus it assembles a full VLIW front-end, reading one instruction word divided into the *qualifier* instruction, jump address pairs and layer-specific configuration words with direct instructions to the pipeline register. The qualifiers determine next execution step and update execution context, like state registers, activation bits, counters and address generator seeds which-after all information is dumped into the pipeline registers, for each of the layers, along with a partitioned configuration word, which was fetched from memory. A VLIW-style fetch construct was decided to be the best way to maintain accurate control, synchronized delivery and fine-grained partitioning of all configuration and activation data. Full flexibility is retained allowing transformation from configuration bits to instruction decode bits, word reordering, configuration override by qualifier control and fine-grained deactivation of parts for power saving.

The qualifiers can, for instance, override one routing segment or one PE function by just overwriting the configuration in the pipeline register of the respective structure. No additional VLIW-specific fetch logic is present such as branch prediction, but it is subject of further investigation along with qualifier-based self-configuration. The assembly is created by exploiting a high-level formalism, based on which spatial and/or temporal assignment of PEs is done derived from input/output locations, dependencies or available PEs. This will create instruction words with a combination of qualifier/no qualifier, memory access pattern activation (if any), configuration word or empty configuration (NOP - the array freezes). Fig. 5 shows two examples of the CGRA assembly word. First qualifier syntax is followed by 2 addresses (true/false path depending on outcome), then each layer with specific instructions and configurations. Often repeating configuration patterns like 5,4,2 (link up,link east,subtract) can be coded into a more readable format if necessary, by grouping the syntax into an assembler symbol (e.g. sub_UW) instead of having fine-grained assembly, but

```
q_mlin_lu    39   39 \ ;qual and true/false addr
    L2: l2a l2rcw8 \   ;addr_gen and pattern
    L1: l1luexe0 \     ;instruction
    L1conf: \   ;routing conf: downstream x2, upstream
        e0:14,0,13 e1:3,12,0 e2:3,12,13 e3:14,0,0 \
        e4:14,0,13 e5:3,12,0 e6:3,12,13 e7:14,0,0 \
        e8:14,0,13 e9:3,12,0 e10:3,12,13 e11:14,0,0 \
        e12:14,0,13 e13:3,12,0 e14:3,12,13 e15:14,0,0 \
    L0conf: \  ;source A, B and opcode (7=mul, 2=sub)
        e0:5,4,2 e1:5,5,7 e2:5,5,7 e3:5,2,2 \
        e4:5,4,2 e5:5,5,7 e6:5,5,7 e7:5,2,2 \
        e8:5,4,2 e9:5,5,7 e10:5,5,7 e11:5,2,2 \
        e12:5,4,2 e13:5,5,7 e14:5,5,7 e15:5,2,2 \
        ss:0 div:0,0,0,0
```

```
uncond      56   56 \
    L2: l2na l2nop \
    L1: l1uiop10 \
    L1conf: \
        e0:0,0,0 e1:0,0,0 e2:0,0,0 e3:0,0,0 \
        e4:11,0,0 e5:0,0,0 e6:3,14,0 e7:14,0,0 \
        e8:0,0,0 e9:11,12,0 e10:15,0,0 e11:1,0,0 \
        e12:0,0,0 e13:12,0,0 e14:3,10,0 e15:3,0,0 \
    L0conf: \
        e0:0,0,0 e1:0,0,0 e2:0,0,0 e3:0,0,0 \
        e4:5,1,1 e5:0,0,0 e6:5,5,1 e7:5,2,1 \
        e8:0,0,0 e9:5,5,1 e10:5,3,1 e11:5,2,7 \
        e12:4,1,2 e13:5,1,1 e14:5,5,7 e15:5,6,7 \
        ss:0 div:0,0,0,0
```

Figure 5. Two CGRA assembly words: regular parallel execution during LU decomposition (left), sequential code execution in U inversion (right)

that adds decode logic.

*3) **Memory access layer L2**:* The main task of L2 and the interface to L1 is to connect to the memory, address generation and carry out memory-centric and destination-aware rearrangement and routing. L2→L1 interface features 32 downstream lines and 16 upstream buffers which must be linked to 8 ports in a well defined fashion. L2 has no configuration bits, having only a set of patterns (< 20) activated via L2 instructions which also initiate memory transfer. Dedicated operations for single/double read/write are activated by the respective transfer pattern.

*4) **Communication layer L1**:* L1 plays a crucial role in Layers, for it is responsible to arrange, time and provide the data to L0. It is comprised of a cascade of routing resources which crosses a clustered register file additional to an independent upstream (L2) to downstream (L0) link. The register cluster comprises of 8 registers and a special divider buffer register. All 8 registers can be simultaneously written by using `regwrite` operations, which guarantee write exclusivity. Rewrite sources encompass upstream, downstream, divider buffer and a set/reset value. The configurations for the regwrite multiplexers are generated internally by application specific *movement patterns*, coded as an instruction called from assembly. Register clusters are shared between pairs of PEs, thus broadcasts can happen locally. Additionally Read Across-2 for vertical clusters is added to exploit scheduling mobility during S1 execution as shown in [23], albeit manually. Especially for S1, where critical path included the divider, cluster broadcast and sharing are exploited. To achieve maximum efficiency, S1 code was completely unrolled and scheduled using VLIW register access concepts and scheduling mobility, which after the `regwrite` configuration bits were derived and grouped into L1 instructions.

L1→L0 downstream data is defined via the downstream mux for each interface wire (2 for each PE), configuration bits forwarded directly from assembly. In case of upstream, similar source select muxes are employed to push any register contents or data from L0 into L2 upstream buffers. For highly parallel S2 and S3, broadcasts to register clusters are used to avoid data reload, while the same data passes through the downstream buffers. For S3, special partial loading is employed to hide loading of the minuend. L1 uses a homogeneous structure, which restricted some flexibility (e.g. matmult could use a hop-2 cluster broadcast), but this simplified design and allowed the use of templated operations. 192 configuration and coding bits are necessary to fully configure L1 instructions and routing.

*5) **Computation layer L0**:* L0 is a generic 2D mesh of reconfigurable FUs, detailed in Fig. 6, noting that instead of the memory double-buffer, the units connect to the L0↔L1 interface, via direct wires to their corresponding L1 structures. The structure of execution layer is based on Q-format fixed-point arithmetic-logic units (ALUs) with multiply-accumulate (MAC) capability. Outputs of each element can be stored in one of the two result buffers.A single-cycle

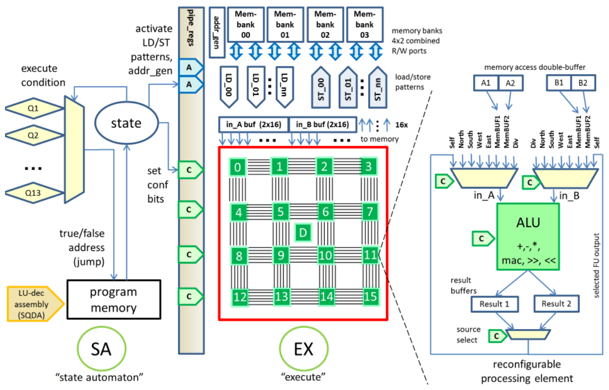

Figure 6. The simple CGRA architecture, with a zoom-in on one of the processing elements.

Q-format divider connects to certain elements besides the memory buffers.

One PE can take input data from 16 sources, 8 for each input, as shown in Fig. 6. Besides the 4 neighboring PEs (north, south, east, west), one PE can also take its own output register as the source, or take data from upstream. The output wires for each element are directly connecting the output registers of the respective neighboring nodes (creating an explicit 2D pipeline in the array), or to upstream structures. Communication via L0 structures have always priority over L1 or L2 communications, exploiting data locality.

Four configuration bits are allocated for each source multiplexer and four bits for the opcode selector. PEs is described at high level as template `OPERATION<>`s, which takes the data on input wires `in_A` and `in_B` and outputs to the output register, considering the opcode for the execution. The 8 interconnect links for each source are modeled also with template `OPERATION<>`s which are activated based on the configuration bits. Thus, to completely configure one element, 12 configuration bits are needed, resulting in 196 configuration bits for the PE array of 4×4 elements. Adding divider source routing and PE output source-select switching (Result1+2) gives 210 total configuration bits for the array. These bits are stored directly in clusters of 4-bit pipeline registers, then directly linked to relevant multiplexer selectors to avoid early (and large) decoding. A `0` configuration bit for any structure maintains current state and storage to exploit temporal scheduling as well.

*6) **Timing and synchronization**:* Fig. 7 shows the timing model of the architecture. The main pipeline divides into the SA and EX stage, while the EX stage is an explicitely controlled 3D pipeline, parallelizing each layer and each FU. Data is always latched at every FU output in L0 and every upstream or downstream pass is latched at least once in the intra-layer interface or L1 local store, leading to a fully synchronous model with controllable data-flow latency.

*7) **The simple CGRA**:* Fig. 6 shows the architecture without the communication layer, in order to investigate the full contribution of L1. Some L2 memory access patterns needed to be rewritten and complete rescheduling, optimizing and re-write of the assembly and SA qualifier structures (now different) have been done to code the applications. Due to LISA high-level design, applications were running on sCGRA within a week.

The PE array is connected to the memories by double buffers, a technique common in traditional CGRA, all loads and stores happening through these buffers while keeping

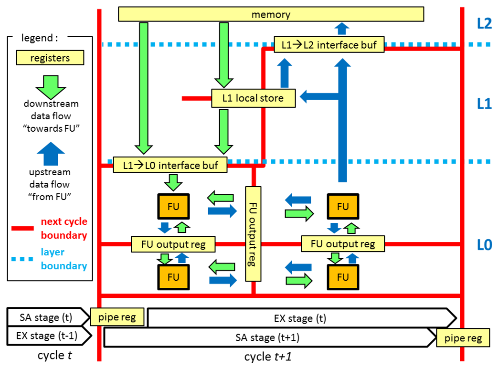

Figure 7. The timing model of the architecture

Table I
IMPROVEMENTS BY USING LAYERS (APPLICATION BLOCK SIZE q=8)

Alg. stage	Operation	Cycles/iteration sCGRA	Cycles/iteration Layers	CRU sCGRA	CRU Layers	Speedup
S1	LU	204	46	10.29%	45.65%	4.43×
	U^{-1}	54	40	27.54%	30.78%	1.35×
	L^{-1}	40	30	17.18%	35%	1.33×
S2U	MatMult	52	46	69.23%	69.56%	1.13×
S2L	MatMult	52	46	69.23%	69.56%	1.13×
S3	MM w/sub	56	48	71.42%	75%	1.16×

the memory-centric patterns from L2. It remains unmodified otherwise. The complete instruction word holds now qualifier encoding (6 bits) immediate true and false addresses (2×12 bits), the load and store encoding (7 bits) and the configuration word (210 bits) resulting in a 247 bit instruction word.

V. EVALUATION

In this section, the layered CGRA is compared against an unlayered one. sCGRA has a LISA description of 4.5k lines for a total of 141 operations, configuration word length of 247 bits, which generates 53k lines of Verilog code. Layers required 10k lines of code in LISA for 197 operations, requires an configuration word of 381 bits and has 95k lines of generated Verilog code. Assembly code size is 235 words for Layers, 352 for sCGRA. Simulations have been conducted up to a matrix size of 2048, limited by the 32-bit implementation of Synopsys PD [22]. It is important to note here that compared with a traditional HDL coding and RTL simulation approach, the high-level debugging environment provided fast and easy debugging and simulation.

The designs support fixed-point operations, which makes it difficult to benchmark against state-of-the-art implementations. However, we note that the runtime (in cycles) achieved by sCGRA is comparable with those reported in [20][24][25], making it a benchmark for evaluating the multi-layered CGRA.

A. Comparing architectural performance

For studying architectural performance, we present a metric based on Computational Resource Utilization (*CRU*). *CRU* is defined as the ratio between active computational resources and the total available computational resources, averaged over the complete execution time. In principle, a higher *CRU* indicates that the handling of data bandwidth is better due to more meaningful computation happening. Table I compares the cycles used and *CRU* of the two architectures, coupled with the speed-up data. Varying the LU algorithmic block size yielded low *CRU* for a 4×4 PE array, however larger blocks violated our memory access constraints. High *CRU* in S1 is bottle-necked by the divider

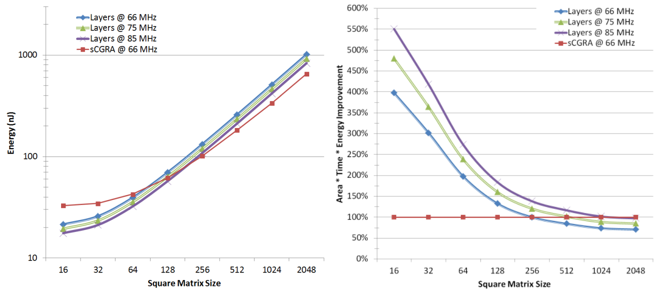

Figure 8. Scaling of energy with matrix size and A-T-E product improvement.

(one Q-format division per cycle), while S2 and S3 are limited by the memory.

The block size $q = 8$ was chosen after evaluating different block sizes, reaching a memory bandwidth utilization for 65-75% of S2 and S3 execution time, for both architectures. Remaining bandwidth is unusable due to read-after-write latency of the memory, forcing idle cycles and decreasing *CRU*. Therefore, with $q = 8$, matrix multiplication on the 4×4 PE array balanced memory traffic and PE load.

Table I reveals that the layered approach gives significant speedup, requiring up to 77% less cycles during sequential execution (most of S1), by exploitation of data reuse and movement in the communication layer. Even for the highly parallel S2 and S3, a speedup can be achieved, while sCGRA is slowed by memory access. Local storage in L1 was fully exploited to minimize loads and stores, thus some improvement could be achieved against the non-layered approach. Although the local store quantity of 64 words was covering the required one for the optimal execution time (40 words), limited broadcast capability prevented achieving optimality, requiring partial re-load of data from memory. A closer analysis revealed that adding inter-cluster broadcast as suggested by Gangwar[23] in L1 would significantly increase S2 and S3 *CRU* in the case of layers, but requires further investigation.

B. Area, timing and power

For the proposed designs, RTL descriptions have been generated by Synopsys PD G-2012.06-SP1, using same options for both architectures, and synthesized both designs with Synopsys DC F.2011-SP3. **Faraday 90nm** technology library was targeted, with clock-gating and operand isolation power optimizations enabled. Switching activity for Power Compiler has been annotated by simulating at RTL level. The area, timing and power consumption figures for both the architectures are summarized in Table II.

The 3D pipeline approach in Layers yielded a higher maximum frequency of 85MHz, while for sCGRA maximum frequency is at 66MHz, requiring a memory bandwidth of 5.44Gbit/s and 4.23Gbit/s, respectively. We observe more area in Layers due to the additional logic and storage in the communication layer. This also results in higher power when compared with sCGRA. Investigating timing reports revealed that the Q-format divider severely impacted timing, underlining the need to replace it with a pipelined divider. The critical path is not impacted at all by the multi-layered concept.

Although Layers has better timing and *CRU*, in terms of energy the gained execution time advantage is negated by higher power consumption as matrix size increases (Fig. 8). This trade-off can be best seen using improvement of area, time and energy product which decreases with matrix size

Table II
SIMULATION AND SYNTHESIS RESULTS, FARADAY $90nm$

Arch.	Max. Freq (MHz)	Area (kGE)	Power (mW)	Matrix size	Time (ms)
sCGRA	66	205.8	8.76	16	0.15
				64	3.10
				256	118.58
				512	850.46
				1024	6287.28
				2048	48953.33
Layers	66	221.5	15.94	16	0.05
				64	1.582
				256	84.93
				512	663.24
				1024	5246.24
				2048	41741.25
Layers	85	250.7	16.85	16	0.04
				64	1.228
				256	65.94
				512	514.99
				1024	4073.55
				2048	32410.86

to 93% of sCGRA for $n = 2048$ (Fig. 8). Reason for this is that for large matrix sizes, S2 and S3 execution in LU decomposition accounts for more than 95% of the time, coupled with the fact that the small speedup in matrix multiplication cannot cover the difference in power (both architectures using the same 4×4 PE array). ATE product is improved up to $> 5\times$ where sequential execution is dominant and the matrix dimensions are small.

The power consumption overhead in layered architecture more than offsets the runtime improvement, resulting in higher energy for some cases. In the experiments we did, this is found to be subjective - depending on the matrix dimensions, amount of functional parallelism and the architectural flexibility. For example, further optimizing S3 section of LU decomposition by specializing the communication layer (L1) can compensate for the energy overhead at the expense of area.

VI. CONCLUSION AND OUTLOOK

In this paper we have introduced the concept of layered coarse-grained reconfigurable architectures and compared it against a traditional design using computationally demanding linear algebra kernels. Our experiments show that the layered concept improves run-time performance over a non-layered design by up to $4\times$ for sequential parts and up to 16% in highly parallel parts, improving area-time-energy product by up to $5.5\times$. We have explored an ADL-based modeling solution for CGRA, which significantly improves design productivity.

In future, we would like to thoroughly benchmark the layered architecture with existing efficient GPGPU, FPGA, DSP and ASIP architectures for linear algebra applications. As an immediate follow-up step, we would port the architecture to support floating point operations and further boost the resource utilization by adding broadcast mechanism in the multi-layered CGRA. In the long run, we would also like to explore efficient scheduling and mapping techniques for layered CGRA.

ACKNOWLEDGMENTS

This research was funded by the UMIC Cluster of Excellence at RWTH Germany. DAAD Exchange Scholarship funded the research visit of the co-author.

REFERENCES

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

[2] B. Mei, A. Lambrechts, J. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *Design & Test of Computers, IEEE*, 2005.

[3] X. Chen, A. Minwegen, Y. Hassan, D. Kammler, S. Li, T. Kempf, A. Chattopadhyay, and G. Ascheid, "FLEXDET: Flexible, Efficient Multi-Mode MIMO Detection Using Reconfigurable ASIP," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 69–76, IEEE, 2012.

[4] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pp. 157–166, IEEE, 1996.

[5] "http://www.tabula.com/". [Online], Tabula.

[6] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Memory access optimization in compilation for coarse-grained reconfigurable architectures," *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 16, no. 4, p. 42, 2011.

[7] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid, "A design flow for architecture exploration and implementation of partially reconfigurable processors," *IEEE Trans. VLSI Syst.*, vol. 16, pp. 1281–1294, oct 2008.

[8] "http://www.menta.fr". [Online], Menta.

[9] "http://www.recoresystems.com/". [Online], Recore Systems.

[10] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, *et al.*, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[11] Y. Kim, J. Lee, A. Shrivastava, J. Yoon, and Y. Paek, "Memory-aware application mapping on coarse-grained reconfigurable arrays," *High Performance Embedded Architectures and Compilers*, pp. 171–185, 2010.

[12] T. von Sydow, M. Korb, B. Neumann, H. Blume, and T. Noll, "Modelling and quantitative analysis of coupling mechanisms of programmable processor cores and arithmetic oriented eFPGA macros," in *Reconfigurable Computing and FPGA's, 2006. IEEE International Conf. on*, pp. 1–10, IEEE, 2006.

[13] X. Zhang, H. Rabah, and S. Weber, "Auto-adaptive reconfigurable architecture for scalable multimedia applications," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pp. 139–145, IEEE, 2007.

[14] J. Portilla, T. Riesgo, and A. De Castro, "A reconfigurable fpga-based architecture for modular nodes in wireless sensor networks," in *Programmable Logic, 2007. SPL'07. 2007 3rd Southern Conference on*, pp. 203–206, IEEE, 2007.

[15] H. Corporaal, *Microprocessor Architectures: from VLIW to TTA*. John Wiley & Sons, 1997.

[16] W. Davis, E. Oh, A. Sule, and P. Franzon, "Application exploration for 3-D integrated circuits: TCAM, FIFO, and FFT case studies," *IEEE Trans. VLSI Syst.*, vol. 17, no. 4, pp. 496–506, 2009.

[17] V. Kumar and Y. Tsai, "On synthesizing optimal family of linear systolic arrays for matrix multiplication," *IEEE Trans. Comput.*, vol. 40, no. 6, pp. 770–774, 1991.

[18] I. Milentijevic, I. Milovanovic, E. Milovanovic, and M. Stojcev, "The design of optimal planar systolic arrays for matrix multiplication," *Computers & Mathematics with Applications*, vol. 33, no. 6, pp. 17–35, 1997.

[19] G. Li and B. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, vol. 100, no. 1, pp. 66–77, 1985.

[20] S. Campbell and S. Khatri, "Resource and delay efficient matrix multiplication using newer FPGA devices," in *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 308–311, ACM, 2006.

[21] G. Golub and C. Van Loan, *Matrix computations*, vol. 3. Johns Hopkins Univ Pr, 1996.

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

[23] A. Gangwar, M. Balakrishnan, and A. Kumar, "Impact of intercluster communication mechanisms on ILP in clustered VLIW architectures," *ACM Trans. on Design Autom. of Electronic Systems*, vol. 12, no. 1, p. 1, 2007.

[24] G. Wu, Y. Dou, J. Sun, and G. Peterson, "A high performance and memory efficient LU decomposer on FPGAs," *IEEE Trans. Comput.*, no. 99, pp. 1–1, 2012.

[25] Z. Nikolic, H. Nguyen, and G. Frantz, "Design and implementation of numerical linear algebra algorithms on fixed point DSPs," *EURASIP J. on Advances in Signal Proc.*, vol. 2007, no. 2, pp. 13–13, 2007.