2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 978-1-6654-2701-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASAP52443.2021.00030

# CGRA-ME: An Open-Source Framework for CGRA Architecture and CAD Research (Invited Paper)

Jason Anderson, Rami Beidas, Vimal Chacko, Hsuan Hsiao,
Xiaoyi Ling, Omar Ragheb, Xinyuan Wang, Tianyi Yu
Dept. of Electrical and Computer Engineering, University of Toronto
Email: janders@eecg.toronto.edu, rabeidas@gmail.com,
{vimal.chacko, julie.hsiao, xiaoyi.ling, omar.ragheb, xinyuan.wang, litalemon.yu}@mail.utoronto.ca

***Abstract*—Coarse-grained reconfigurable arrays (CGRAs) are programmable hardware platforms that can be used to realize application-specific accelerators for higher performance and energy efficiency. A CGRA is a 2D array of configurable logic blocks & interconnect, where the logic blocks are typically large & ALU-like, and the interconnect is word-wide. CGRA-ME is a software framework that enables the modelling and exploration of CGRA architectures, as well as research on CGRA CAD algorithms. With CGRA-ME, an architect can specify a CGRA architecture at a high level of abstraction. A set of applications can be mapped onto the architecture to assess the mappability, power, performance and cost. CGRA-ME also allows one to generate synthesizable Verilog RTL for the modelled CGRA, permitting its implementation as an ASIC or FPGA overlay. In this paper, we describe the CGRA-ME framework [5] and overview its capabilities and current limitations. We discuss ongoing and prior research conducted with the framework, as well as outline future plans. We believe CGRA-ME will be a valuable contribution to the community, enabling new research on CGRA CAD & architectures.**

## I. Introduction

Coarse-grained reconfigurable arrays (CGRAs) are programmable hardware devices that have large configurable logic blocks (often resembling ALUs) and datapath-style word-wide configurable interconnect. In comparison with field-programmable gate arrays (FPGAs), where the long-standing programmable element is a look-up-table (LUT) and where interconnect is configurable at the granularity of an individual logic signal, CGRAs are less flexible and dedicate correspondingly less silicon area overhead to programmability. The value proposition of CGRAs is that they are closer to custom ASICs from the perspectives of power, performance & area. While at the same time, retaining *some* of the programmability of fine-grained FPGAs.

While the concept of CGRAs emerged about 25 years ago [10], their industrial impact was initially modest. CGRAs, however, are garnering renewed attention (e.g. [4], [21], [22], [24]), which is a direct result of Moore's Law and Dennard scaling becoming obsolete [11]. These trends have stalled performance improvements in standard processors, leading to the proliferation of compute accelerators, where customized hardware is used to raise application performance and power efficiency. CGRAs are one available means of realizing such compute accelerators. Moreover, the increasing ubiquity of machine learning (ML) has been enabled by specialized hardware. Deployment of ML in edge or cloud environments typically requires bespoke hardware, and recent years have seen a large number of startup companies, as well as large multinationals developing custom hardware for ML inference and training. Such custom hardware closely resembles CGRAs.

We describe CGRA-ME [5], an open-source software framework for CGRA research under active development at the University of Toronto. CGRA-ME aims to facilitate research in two areas: 1) the modelling and exploration of CGRA architectures, and 2) CAD/mapping algorithms for CGRAs. The genesis of the CGRA-ME project was the observation that the majority of prior work on CGRAs had been point solutions. CGRA researchers typically proposed a specific CGRA architecture and an associated toolchain, and then demonstrated the architecture's efficacy to improve performance and/or energy consumption for a suite of applications. CGRA architectures were seldom compared against one another, owing to there being no open-source architecture evaluation framework, such as those that exist for standard processors [2], FPGAs [23], and GPUs [1]. Moreover, research on CGRA CAD was impeded by a lack of an open-source baseline toolchain where new innovations could be implemented and evaluated. The CGRA-ME project was launched with the purpose of providing such an architecture evaluation and CAD framework to the research community.

The paper is organized as follows: In Section II, we provide a high-level overview of CGRA-ME, its components and capabilities. Then, Sections III and IV describe recent research conducted using the framework on CGRA CAD & architectures, including mapping algorithms, compact area/performance modelling, implementing CGRAs on top of FPGAs as overlays, hybrid systems having a RISC-V processor and CGRA accelerator, and dynamically reconfigurable CGRAs. Section V provides a roadmap for future CGRA-ME research to raise its adoption and improve its flexibility. Section VI concludes the paper.

## II. Overview of CGRA-ME

Fig. 1 provides a block diagram of the CGRA-ME framework. Green blocks represents inputs to the framework, comprising an application benchmark (1), a CGRA architecture

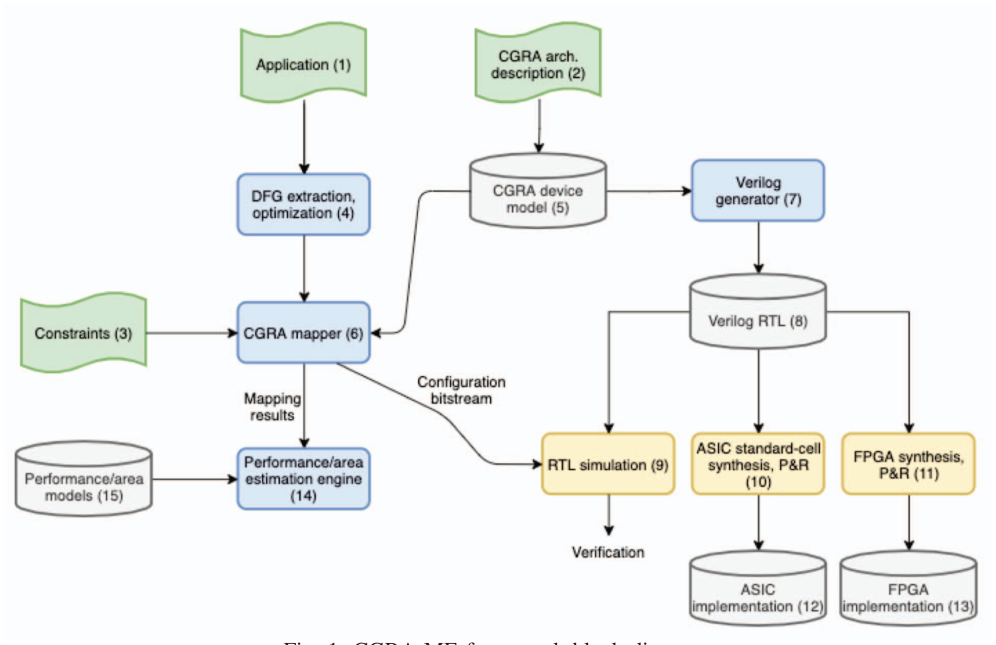


Fig. 1: CGRA-ME framework block diagram.

description (2), and constraints (3). The application is expressed in the C language annotated with pragmas to indicate the compute kernel designated for the CGRA – typically a compute-intensive loop body. The target CGRA architecture can be specified in one of two forms: 1) using a C++ API, or 2) using an XML-based language. Constraints presently supported include allowing a user to pre-place/lock components of the application (e.g. I/Os) to specific CGRA resources, as well as to set a target initiation interval (II) for the kernel's CGRA implementation.

The data-flow graph (DFG) for the compute kernel is extracted in block (4) using an LLVM-based compiler pass [14]. The extracted DFG is represented a .dot file. As such, applications can also be constructed by directly writing .dot files. The architecture description (2) is parsed to create an in-memory device model (5). The device model, constraints, and DFG are then fed into the CGRA mapper (6).

The CGRA-ME mapper, elaborated upon in Section III-A, is architecture agnostic. Meaning, the mapping algorithm is not tied to a *specific* CGRA, but rather, is able to map to the user-provided architecture. A configuration bitstream is produced for a DFG that can be mapped to the architecture. Additionally, the mapping results are an input to the area/performance estimation engine (14) (c.f. Section III-B). The area/performance estimates produced for the application (mapped onto the CGRA) are informed by area/performance models of the CGRA's primitive components (15).

The CGRA device model can also be fed into a Verilog RTL generator (7) to produce RTL for the modelled CGRA. Yellow boxes in the figure reflect third-party tools that are not part of the CGRA-ME framework. The generated CGRA RTL is application-agnostic and must be configured with the bitstream to realize the application. The "configured" RTL can then be simulated (e.g. with ModelSim) for cycle-accurate functional verification (8). Alternately, the CGRA RTL can be input to a standard-cell ASIC toolflow (10) or an FPGA toolflow (11). In our work, we have used Synopsys Design Compiler, Cadence Innovus, and PrimeTime/PrimePower for the ASIC flow. We have targeted both Xilinx and Intel FPGAs to realize CGRAs as *FPGA overlays*. Overlays are a programmable hardware fabric (the CGRA) that is implemented on top of a lower-level programmable hardware fabric (the FPGA).

A wide range of CGRAs can be modelled with CGRA-ME, including *multi-context* CGRAs, which are dynamically reconfigurable and concurrently hold multiple configurable bitstreams that are cycled among according to a prescribed recipe. Each such bitstream is called a context, and the CGRA's logic and routing may differ in each context.

## III. CAD and Architectural Modelling

### A. Mapping Algorithms

The input application kernel is represented using the generated DFG, with vertices for operations and edges for value dependencies. The target CGRA architecture is itself represented using a graph, referred to as a *modulo routing resource graph* (MRRG) [16], where nodes in the MRRG represent hardware resources and edges describe the connectivity between those resources across both time and space. Mapping associates the application DFG's operations and data dependencies with MRRG nodes and edges, respectively.

CGRA-ME currently implements two architecture-independent mapping solutions utilizing integer linear programming (ILP): an exact fully generic solution [6], and a heuristic connectivity-centric solution [27].

The exact mapper is motivated by the limited routing resources present in most CGRA architectures, which decreases the probability of finding feasible solutions using heuristic approaches. The mapping problem in its entirety is described by an ILP formulation, defining binary variables for the placement of operations onto functional units and the use of routing resources by values. For example, consider a binary variable in the ILP, $F_{p,q}$, that if true, indicates that DFG node $q$ is placed in CGRA functional unit, $p$. Then, one of the constraints for the feasible mapping solutions is defined as:

$$\sum_{p \in FuncUnits} F_{p,q} = 1, \forall q \in Ops$$

forcing every DFG operation $q$ to be mapped to a single functional unit. The functional unit exclusivity is forced by:

$$\sum_{q \in Ops} F_{p,q} \leq 1, \forall p \in FuncUnits$$

The objective function for the ILP formulation is defined to minimize the number of used CGRA interconnect resources in the mapping and the interested reader is referred to [6] for the complete formulation.

The exact mapper was compared to a simulated annealing-based mapper and managed to find 30% more feasible solutions under various application and architecture scenarios. Runtime was reasonable with most tested scenarios finishing in under one hour. However, as is the fate for many optimizers that rely on generic ILP solvers, scalability remained an issue, which motivated the heuristic approach.

Still guided by the limited routing resources of typical CGRAs and variety of their architectures, our second mapper utilizes an ILP formulation with constraints similar to the ones listed above, but minimizes the search space with more practical constraints on routing. It assumes data can only flow to a destination within its neighborhood, using one of a pre-computed set of paths connecting the neighbors. To speedup the optimization, the mapper breaks the problem into iterations of placement followed by routing, by explicitly modelling hard routing constraints during relaxed placement then checking feasibility in a detailed routing step [27].

The heuristic solution was evaluated over extended scenarios of various applications, architectures, and context counts and compared with the exact solution. It managed to find a feasible mapping in more than 93% of tested cases, with the exceptions being highly constrained single context problems, while delivering up to 2 orders of magnitude speedup in runtime. At this speed, the mapper is useful not only in single application mapping, but also for architecture design space exploration. Table I highlights the geometric mean runtimes for the exact and heuristic ILP mappers across 28 benchmarks and two CGRA architectures, ADRES [17] and HyCUBE [13]. The improved mapper performs better across all contexts for both architectures.

### B. Power and Area Modelling

While the RTL generated by CGRA-ME can be input to an ASIC or FPGA synthesis/place/route flow to assess CGRA area and performance, the run-time required for such flows can be lengthy, taking hours or even days for the largest designs. Such run-times are detrimental to rapid architectural exploration, where one wishes to explore and evaluate a broad range of points in the architecture space. What is needed are compact models that allow one to compute area/performance with low run-time and reasonable accuracy, without invoking the full ASIC/FPGA toolflow.

TABLE I: Average runtimes (across 28 benchmarks) and speedup between 2 ILP-based mappers for ADRES and HyCUBE having different #s of contexts.

| Arch. | # Contexts | Exact ILP [6] | Heuristic ILP [27] | Speedup |
|---|---|---|---|---|
| ADRES | 1 | 4.4 | 1.5 | 3.0 |
| HyCUBE | 1 | 79.6 | 4.3 | 18.6 |
| ADRES | 2 | 310.2 | 7.1 | 43.5 |
| HyCUBE | 2 | 8422.3 | 13.5 | 625.9 |
| ADRES | 3 | 1303.7 | 15.6 | 83.5 |
| HyCUBE | 3 | 12788.1 | 23.2 | 551.7 |

The approach in [19] is to pre-characterize the area/delay of architectural primitives available within CGRA-ME that one uses to compose a complete CGRA. This includes multiplexers of a variety of sizes, registers and register files, ALUs having different capabilities and so on. Each of these primitives was implemented in standard cells using a commercial ASIC toolflow. The primitives were implemented in both an area-optimized and delay-optimized manner, by supplying constraints to the ASIC tools. The two optimization styles allow one to assess area/performance of area-optimized as well as timing-optimized CGRAs. A characterization database was constructed containing area/delay values for each primitive.

With the characterization database, the area of an arbitrary CGRA can be estimated by aggregating the area of each of its constituent primitives. Performance estimation is more involved, however, as it involves analyzing the critical path delay within the portion of a CGRA *used* by a given application. For this task, CGRA-ME incorporates an open-source static timing analysis (STA) tool, Tatum [18]. After application mapping in CGRA-ME, the CGRA resources used by the application are traversed and a timing-graph is constructed in Tatum. Delays from the characterization database are annotated onto the timing graph, permitting Tatum's STA to be invoked and the critical path delay reported.

## IV. Architectures and Circuits

### A. Implementing CGRAs as Overlays on FPGAs

The work in [25] is centered on implementing CGRAs with FPGAs, i.e. as FPGA overlays. The generic Verilog produced by CGRA-ME is used to produce first-cut overlay implementations, where ADRES [17] and HyCUBE [13] CGRAs are implemented on Xilinx UltraScale and Intel Stratix 10 FPGAs. The area and performance is analyzed as a means of driving FPGA architecture-specific optimizations for the overlays.

Three main enhancements are made to optimize a CGRA overlay's area/performance on FPGAs. First, as CGRAs contain large numbers of multiplexers to realize configurable interconnect, [25] proposes a lightweight implementation that

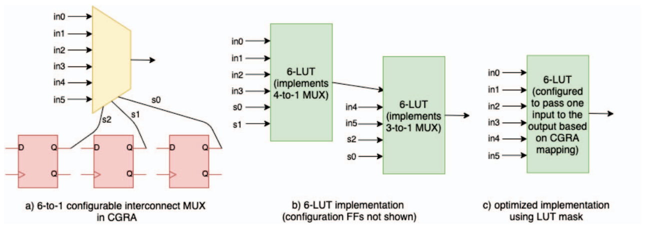

in0
in1
in2
in3
in4
in5
s0
s1
s2
D
Q
6-LUT (implements 4-to-1 MUX)
6-LUT (implements 3-to-1 MUX)
6-LUT (configured to pass one input to the output based on CGRA mapping)
a) 6-to-1 configurable interconnect MUX in CGRA
b) 6-LUT implementation (configuration FFs not shown)
c) optimized implementation using LUT mask

Fig. 2: CGRA interconnect multiplexer in FPGA-overlay implementation.

TABLE II: Resource and performance results for a $4 \times 4$ HyCUBE CGRA implemented on Xilinx UltraScale+ FPGA.

| | Unopt. | Unopt. FP | Opt. | Opt. FP |
|---|---|---|---|---|
| LUTs | 17770 | 17673 | 11230 | 11069 |
| DSPs | 48 | 48 | 32 | 32 |
| LUTRAM | 64 | 64 | 64 | 64 |
| Fmax (MHz) | 187.2 | 188.6 | 297.8 | 307.6 |

leverages the ability to set LUT masks in an FPGA logic block. Fig. 2 illustrates the idea. Part (a) shows a configurable 6-to-1 interconnect multiplexer, with the select values held in flip-flops. Part (b) shows a conventional FPGA implementation, requiring two 6-LUTs, where the left LUT selects between in0-in3, and the right LUT selects between the output of the left LUT, or in4-in5. Part (c) shows the optimized implementation, requiring a single LUT whose function (mask) is set based on the CGRA mapping results. For example, if after CGRA mapping input in4 was selected to be passed through the interconnect multiplexer, then the function of the LUT in Fig. 2(c) would be set to pass in4 to the output. The CGRA overlay can be pre-compiled for the FPGA and the FPGA synthesis tools can be invoked in "incremental" mode to simply update the LUT masks according to the CGRA mapping results. The optimization improves the overlay's area and performance.

The second and third enhancements to the overlay implementation include changes to exploit FPGA DSP blocks for implementing CGRA ALUs, and also an efficient multi-ported register file design that takes advantage of the ability to use LUTs to implement small RAMs. Overlay performance was further improved by floorplanning to ensure the physical layout of CGRA rows and columns aligned with the logical layout.

Results highlights are shown in Table II for a representative $4 \times 4$ HyCUBE CGRA implemented as an overlay on a Xilinx UltraScale+ FPGA. The first three rows of the table show the resource usage of LUTs, DSP blocks and LUTRAMs. The last row of the table gives the performance in MHz. Columns labeled "Unopt." give results for unoptimized overlays; columns labeled "Opt." give results for overlays incorporating the optimizations mentioned above. The columns labeled "FP" reflect overlay implementations that are also floorplanned. Observe that LUT resource usage drops from ∼17K in the unoptimized implementations to ∼11K in the optimized. Performance increases from ∼188MHz in the unoptimized to over 300MHz in the optimized. The results demonstrate the importance of considering FPGA architectural features in realizing efficient CGRA overlays on FPGAs.

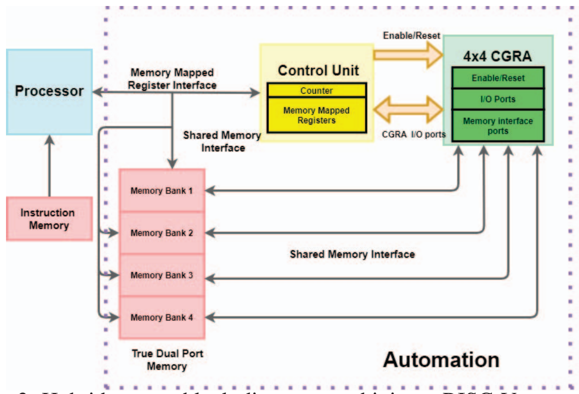


Fig. 3: Hybrid system block diagram combining a RISC-V processor with a CGRA,

## B. Hybrid CGRA/Processor Systems

CGRAs are often used for compute acceleration alongside processors and towards supporting this, [15] extends CGRA-ME to permit the generation of hybrid systems that combine a CGRA and a RISC-V processor. The RISC-V processor developed as part of the PULP platform was selected [9]. The hybrid system can be automatically generated with our framework and any CGRA that is modelled in CGRA-ME can be attached to the processor. In this hybrid system, the RISC-V processor controls and transfers data to/from CGRA and the CGRA performs the compute-intensive parts of the application.

Fig. 3 shows the block diagram of a hybrid system example with a 4×4 CGRA connected to the processor. All parts except the processor and its instruction memory can be automatically generated. The architecture of the control unit and the number of memory banks change corresponding to the architecture of the CGRA incorporated into the hybrid system. The processor and CGRA communicate through memory-mapped addresses connected to the I/O ports on the CGRA and also through shared memory between the processor and CGRA. The processor and the control unit invoke the starting and stopping of the CGRA. The CGRA starts by receiving a "start" signal from the processor via a memory-mapped address. The processor can stop the CGRA using a counter located in the control unit. The counter is pre-loaded with the desired number of CGRA execution cycles, based on the application latency and CGRA kernel trip-count. The counter commences counting down when the CGRA is enabled and stops the CGRA when the count reaches 0. The memory block, as seen in the block diagram, is divided into four memory banks to fully utilize the multiple memory ports of the CGRA.

Four applications were used to evaluate the performance of the hybrid system: matrix multiply ($MM$), K-nearest neighbor ($KNN$), a finite impulse response ($FIR$) filter and polynomial evaluation ($POLY$) via Taylor expansion of $e^x$. These applications are also implemented on RISC-V running software alone and a RISC-V with a vector unit [20]. For the hybrid systems, we compare two systems with different sizes of CGRA: a 4×4 CGRA and an 8×8 CGRA. For the vector

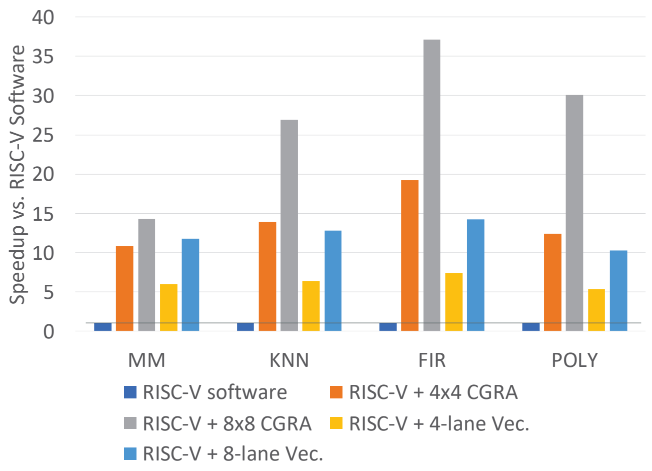


Fig. 4: Cycle-latency speedup of benchmarks executing: 1) on software on RISC-V, 2) on two types of hybrid RISC-V+CGRA systems, 3) on a RISC-V processor with vector extensions.

processor, we evaluate two systems with different numbers of lanes: 4 lanes and 8 lanes. Fig. 4 shows cycle-latency speedup results for the applications running in the various scenarios. The speedups are computed relative to software running on the RISC-V. From Fig. 4, we observe order-of-magnitude speedups compared to the RISC-V processor alone. The speedup arises from the spatial and pipeline parallelism of the CGRAs, branch-free operation and memory-access parallelism compared to RISC-V processor alone. For the processor with a vector unit, the hybrid system still offers speedups because of its spatial and pipeline parallelism.

## C. Multi-Context CGRAs

The concept of *multiple contexts* can be seen in some of the academic and industrial CGRAs (e.g. [13], [26]), where, multiple configuration bitstreams are stored in the CGRA instead of just one. The CGRA switches between these *contexts* in a predefined pattern. This allows a small CGRA to implement a larger application than its hardware permits by time-multiplexing the CGRA hardware. The work in [3] compares different circuit implementations of the multi-context feature in CGRAs from the power, performance and area (PPA) perspectives.

Fig. 5 presents an example of the multi-context concept for a 2×2 CGRA with 2 contexts. The example shows that the lower right block performs an addition on two external inputs in context 0, and in context 1, it performs a logical OR operation on an external input and the value stored in a register in context 0. In this way, the functionality and the connectivity of the elements in the CGRA can change in each context.

Three different circuit alternatives for implementing multi-context support in CGRAs are presented and evaluated in [3]. The first circuit cycles through contexts using a circular shift register; the second approach uses a one-hot "context" counter and AND/OR logic to select configuration bits based on the current context; and the third one uses a binary counter and multiplexer-based logic.

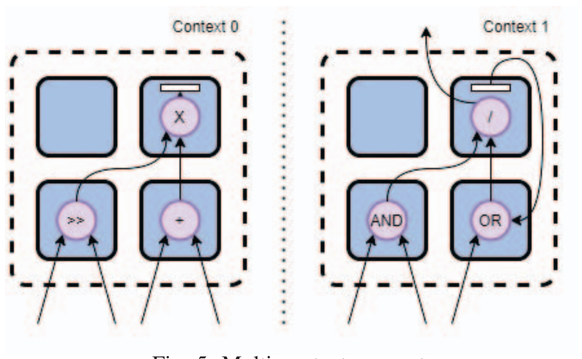


Fig. 5: Multi-context concept.

TABLE III: Area, performance and power results of HyCUBE CGRA of size 4×4 for the candidate implementations for multi-context support.

| Design | Total Contexts (N) | HyCUBE Area (μm²) | HyCUBE Timing (ns) | HyCUBE Power (mW) |
|---|---|---|---|---|
| Single Context | 1 | 237493 | 4.27 | 1.78 |
| Circular Shift Register | 2 | 258087 | 4.33 | 3.64 |
| | 4 | 279966 | 4.34 | 2.75 |
| | 8 | 333246 | 4.27 | 3.31 |
| One-hot Counter | 2 | 255531 | 4.48 | 2.11 |
| | 4 | 274907 | 4.45 | 1.94 |
| | 8 | 323400 | 4.47 | 1.64 |
| Binary Counter | 2 | 253273 | 4.55 | 2.10 |
| | 4 | 276915 | 4.62 | 2.01 |
| | 8 | 326180 | 4.69 | 1.77 |

Table III shows the area, performance and power results presented in [3] for the different implementations presented in the paper along with that of a single context CGRA for comparison. The results show that the implementation of multi-context support in CGRAs has significant impact on their power, performance and area (PPA). It also shows that the binary and one-hot counter approaches for multi-context implementation have the best PPA trade-off when compared to the circular shift register approach.

## D. Double-Pumped Interconnect

Double-pumping is a circuit-design approach where a sub-circuit within a larger circuit is clocked at 2× the system clock frequency. In [28], we use CGRA-ME to model various CGRA architectures derived from the ADRES [17] and HyCUBE [13] architectures and evaluate the area reductions afforded by double-pumping and halving the width of the interconnect. The interconnection styles explored are categorized into: 1) whether the interconnect is *coupled* (e.g. PE can either process or transfer data, like in ADRES) or *decoupled* (e.g. PE can process and transfer data simultaneously, like in HyCUBE) from the PE, and, 2) whether it is *registered* (e.g. PE is isolated from the interconnect via registers) or *non-registered* (e.g. either the input or output of a PE is allowed to connect directly to the interconnect).

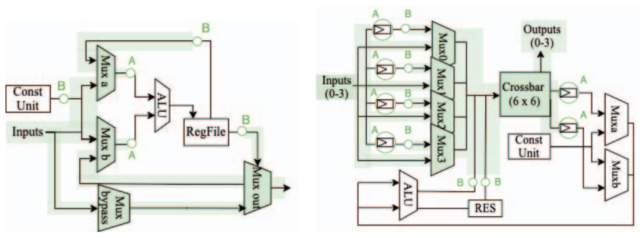
Across different styles of 32-bit word-width CGRAs and optimization goals in the synthesis tool, the full-width interconnect area ranges from 14% to 27% of the total circuit area, representing a non-trivial area cost. By halving the width and double-pumping the interconnect, an area reduction of up to 16% can be achieved, with higher radix interconnects having higher area-reduction potential. Larger area reductions were observed when timing-optimized synthesis was applied, due to the synthesis tool's selection of larger and higher-drive-strength cells for the baselines. Synthesis and place and route were done targeting a 45$n$m standard-cell ASIC implementation. Application-level functional verification and performance analysis were done on CGRA-ME-generated benchmark application bitstreams.

Fig. 6 shows how double-pumped-interconnect is incorporated in the *non-registered* ADRES and HyCUBE PEs. The structure of the full-width PE is shown in black; the green-shaded components represent targets for area reduction and are halved when double-pumped. Additional converters are inserted throughout the PEs to enable double-pumping. 32:16 converters (serializing) are instantiated at each point `B`, and 16:32 converters (deserializing) are instantiated at each point `A`. By sending different halves of the 32-bit word through the interconnect multiplexers at different times, data transfers can be completed with the half-width interconnect. The tradeoffs associated with different converter designs are discussed in more detail in [28]. The technique uses both levels of the system clock, meaning that the double-pumped CGRA does not require additional faster clocks.

Table IV shows the area and timing performance for the CGRA architectures whose PEs are double-pumped as per Fig. 6, assuming all data communications are between PEs that are immediate neighbors. Note that the ADRES presented in this table uses NSEW + diagonal (8-way) interconnects, and the "size" column represents the number of PEs in each row and column of each CGRA. Overall, area improvements were observed with some performance trade-off for these non-registered interconnect architectures. For registered interconnect architectures in [28] not illustrated here, area improvements can be observed with minimal performance overhead, and in some cases performance improvements.

TABLE IV: Area and minimum possible periods for baseline and double-pumped HyCuBE and ADRES CGRAs.

| Architecture | Size | Baseline | | Double-pumped | |
|---|---|---|---|---|---|
| | | Area ($\mu m^2$) | Min Period ($ns$) | Area ($\mu m^2$) | Min Period ($ns$) |
| **HyCUBE** | 4 | 148700 | 3.48 | 125163 | 3.90 |
| | 8 | 576764 | 3.49 | 483804 | 3.90 |
| **ADRES** | 4 | 112943 | 3.68 | 100997 | 4.50 |
| | 8 | 445767 | 3.76 | 395508 | 4.60 |



(a) ADRES PE. (b) HyCUBE PE.

Fig. 6: Double-pumped ADRES and HyCUBE PEs with full-width modules shown in black; components shaded in green are halved when double-pumped. Circle enclosing registers at points `A` indicate a replacement of registers with 16:32 converters.

## V. Roadmap

Having highlighted recent research directions above, we now turn towards the future and point to research planned and underway.

**Hybrid Flow Compiler Enhancements:** The hybrid toolflow refers to the high-level software/hardware partitioning of applications into a RISC-V and CGRA. In this flow, the serial portion of a application would be designated to the RISC-V and the compute-intensive portion of the application would be realized on the CGRA. The partitioning of the application into hardware and software parts would be automated in the compiler and a bitstream/binary executable would be generated to program both the RISC-V and the CGRA, respectively. Optimizations in the compiler would account for the spatial and pipeline parallelism of the CGRA, as well as presence of multiple memory banks/ports. In the work described in Section IV-B, such optimizations were done manually.

**Floating Point:** Presently, CGRA-ME supports functional units that can perform integer and logical operations. Floating point (FP) is needed for many applications, e.g. scientific computing or ML training. We are presently adding FP primitives from the FloPoCo project [7] into CGRA-ME. The FloPoCo cores offer configurable precision for the mantissa and exponent, as well as configurable operator latency.

**Predication:** Software programmers naturally make heavy use of if-else statements, leading to branches in the executable code. CGRA-ME presently requires a DFG which is "branch free", containing no control flow. This requires compiler techniques (e.g. [8]) to collapse control flow into straight-line code. Such collapsing also requires certain operations in the optimized code to *only* execute under certain conditions – predicated code. We are actively working to enhance CGRA-ME to support such predicated execution, by allowing Boolean predicates to be routed through the CGRA on a separate configurable interconnection network.

**Elastic (Latency Insensitive) CGRAs:** CGRA-ME currently uses static scheduling to map the application to the architecture wherein, the specific clock cycle in which each operation occurs is fixed by the scheduler. However, there are limitations to static scheduling as the latency of reading and writing to memory may be unpredictable. Also, static scheduling may suffer from workload imbalance and insufficient parallelism. A new architecture is proposed, called an elastic CGRA [12], that introduces a latency-insensitive design in which execution is input-dependent and dynamic and obviates scheduling – computations only trigger when their inputs are ready. We are actively working to support elastic behavior within CGRA-ME via an expanded library of elastic circuits primitives, including elastic buffers, forks, and joins.

**High-Level Cycle-Accurate Simulation:** For functional verification and to assess the cycle latency of an application, one can simulate the CGRA-ME-generated Verilog RTL, configured with the bitstream of a mapped application. The same Verilog can be used for ASIC/FPGA synthesis, and consequently, it contains the *complete* CGRA specification down to the level of individual logic signals, clock and reset. RTL simulation times can be lengthy for large CGRAs and/or large applications. Rather than simulating the complete RTL, an alternative, higher-level cycle-accurate specification can be generated for verification and performance assessment, perhaps using SystemC.

**Expanding the Library of Available Primitives:** When composing a CGRA with CGRA-ME, the architect uses a set of primitives drawn from a "toolbox", including ALUs with varied capabilities, I/O ports, memory ports, registers and register files, as well as interconnect primitives for multiplexers, crossbars, and direct connectivity. We look to expand the set of available primitives to broaden the space of CGRAs that may be modelled and explored in the framework. For example, we are currently lacking primitives for DMA and data marshalling that would be valuable additions to the toolbox.

## VI. Conclusions and Future Work

CGRA-ME is an open-source software framework for research on the *modelling and exploration* of CGRA architectures, as well as CAD tools for CGRAs. The framework is under active development, with recent thrusts on scalable CGRA mapping algorithms, rapid assessment of CGRA area/performance, CGRAs implemented as FPGA overlays, hybrid systems comprising a processor and a CGRA, multi-context CGRAs, and interconnect optimizations for CGRAs. While the framework supports the modelling of a wide range of CGRAs, and mapping thereto, much more work needs to be done. Specific future directions include improved compiler techniques for targeting hybrid systems with a CGRA and a processor, a broader library of primitives, including floating point and dataflow-style cores, and support for predication. To follow the project and learn more, visit: `https://cgra-me.ece.utoronto.ca`.

## References

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed gpu simulator," in IEEE ISPASS, 2009, pp. 163–174.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," SIGARCH Comput. Archit. News, vol. 39, no. 2, 2011.

[3] V. Chacko and J. Anderson, "Power, performance and area consequences of multi-context support in cgras," in IEEE ASAP, 2021.

[4] G. Charitopoulos, D. N. Pnevmatikatos, and G. Gaydadjiev, "Mc-def: Creating customized cgras for dataflow applications," ACM Trans. Archit. Code Optim., vol. 18, no. 3, Apr. 2021.

[5] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A unified framework for CGRA modelling and exploration," in IEEE ASAP, 2017, pp. 184–189.

[6] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in ACM/IEEE DAC, 2018.

[7] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," IEEE Design and Test of Computers, vol. 28, no. 4, pp. 18–27, 2011.

[8] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," IEEE Trans. on Computers, vol. 30, no. 7, 1981.

[9] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable iot endpoint devices," IEEE Trans. VLSI, vol. 25, no. 10, pp. 2700–2713, 2017.

[10] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: a reconfigurable architecture and compiler," Computer, vol. 33, no. 4, pp. 70–77, 2000.

[11] W. Haensch, "Scaling is over — what now?" in Annual Device Research Conference (DRC), 2017, pp. 1–2.

[12] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in ACM FPGA, 2013, p. 171–180.

[13] M. Karunaratne, A. K. Mohite, T. Mitra, and L. Peh, "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," in ACM/IEEE DAC, 2017, pp. 1–6.

[14] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in IEEE CGO, 2004, pp. 75–86.

[15] X. Ling, T. Notsu, and J. Anderson, "An open-source framework for the generation of RISC-V processor + CGRA accelerator systems," in Euromicro Conference on Digital System Design, 2021.

[16] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in IEEE/ACM DATE, 2003, pp. 296–301.

[17] ——, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in FPL, 2003, pp. 61–70.

[18] K. E. Murray and V. Betz, "Tatum: Parallel timing analysis for faster design cycles and improved optimization," in 2018 International Conference on Field-Programmable Technology (FPT), 2018, pp. 110–117.

[19] K.-P. Niu and J. H. Anderson, "Compact area and performance modelling for CGRA architecture evaluation," in IEEE FPT, 2018, pp. 126–133.

[20] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "RISC-V$^2$: A scalable RISC-V vector processor," in IEEE ISCAS, 2020, pp. 1–5.

[21] G. Peng, L. Liu, S. Zhou, S. Yin, and S. Wei, "A 2.92-Gb/s/W and 0.43-Gb/s/MG flexible and scalable CGRA-based baseband processor for massive MIMO detection," IEEE JSSC, vol. 55, no. 2, pp. 505–519, 2020.

[22] A. Podobas, K. Sano, and S. Matsuoka, "A template-based framework for exploring coarse-grained reconfigurable architectures," in IEEE ASAP, 2020, pp. 1–8.

[23] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: Architecture and cad for fpgas from verilog to routing," in ACM FPGA, 2012, p. 77–86.

[24] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "OpenCGRA: An open-source unified framework for modeling, testing, and evaluating cgras," in IEEE ICCD, 2020, pp. 381–388.

[25] I. Taras and J. H. Anderson, "Impact of FPGA architecture on area and performance of CGRA overlays," in IEEE FCCM, 2019, pp. 87–95.

[26] T. Toi, N. Nakamura, T. Fujii, T. Kitaoka, K. Togawa, K. Furuta, and T. Awashima, "Optimizing time and space multiplexed computation in a dynamically reconfigurable processor," in IEEE FPT, 2013, pp. 106–111.

[27] M. J. P. Walker and J. H. Anderson, "Generic connectivity-based CGRA mapping via integer linear programming," in IEEE FCCM, 2019, pp. 65–73.

[28] X. Wang, T. Yu, H. Hsiao, and J. Anderson, "Double-pumping the interconnect for area reduction in coarse-grained reconfigurable arrays," in IEEE ASAP, 2021.