



UltraSynth: Insights of a CGRA Integration into a Control Engineering Environment

Dennis Wolf¹ · Andreas Engel² · Tajas Ruschke¹ · Andreas Koch² · Christian Hochberger¹

Received: 2 May 2020 / Revised: 27 October 2020 / Accepted: 18 January 2021 / Published online: 27 February 2021
© The Author(s) 2021

Abstract

Coarse Grained Reconfigurable Arrays (CGRAs) or Architectures are a concept for hardware accelerators based on the idea of distributing workload over Processing Elements. These processors exploit instruction level parallelism, while being energy efficient due to their simplistic internal structure. However, the incorporation into a complete computing system raises severe challenges at the hardware and software level. This article evaluates a CGRA integrated into a control engineering environment targeting a Xilinx Zynq System on Chip (SoC) in detail. Besides the actual application execution performance, the practicability of the configuration toolchain is validated. Challenges of the real-world integration are discussed and practical insights are highlighted.

Keywords Reconfigurable accelerator · Coarse grained reconfigurable architecture · Control engineering · System integration · Scalability

1 Introduction

In [16] we have presented a CGRA-based hardware accelerator that was integrated into a Realtime (RT) target for control engineering purposes. There are many requirements to a computing system in the field of control engineering. Hard real time, low jitter, computational power and the ability to incorporate sensors and actuators are the major demands. When control cycles below 10 μ s

are required, Field Programmable Gate Arrays (sFPGAs) are often suggested, but engineers would like to test and evaluate different choices of parametrization or varying control algorithms with virtually no delay and preferably with a real test rig. Since most control algorithm provide at least decent parallelism, the use of a CGRA can be a good choice. As the target technology is a Xilinx Zynq FPGA, we use the CGRA as an overlay architecture in order to reduce synthesis time while being able to benefit from the computational potential of a CGRA. The targeted applications in [16] are modeled with CAMEL-View, which is a design environment for mechatronic systems [8].

Hence, the CGRA is confronted with an integration into a real-world industrial environment. The resulting challenges for the accelerator that are not covered with usual (often synthetic) benchmark suites for accelerators. This contribution introduces the CGRA integration and gives a detailed analysis on how the system performs as well as the insights that can be drawn. It enhances previous research [16] especially by investigating the scalability of the tool and application runtime with the CGRA size and by proposing possible micro-architecture improvements allowing to implement larger CGRAs on the same FPGA.

Serving as an introduction, Section 2 discusses related work and Section 3 gives a brief explanation of the complete system including the CGRA. The communication between the CGRA and its surroundings is detailed in

✉ Dennis Wolf
wolf@rs.tu-darmstadt.de

Andreas Engel
engel@esa.tu-darmstadt.de

Tajas Ruschke
ruschke@rs.tu-darmstadt.de

Andreas Koch
koch@esa.tu-darmstadt.de

Christian Hochberger
hochberger@rs.tu-darmstadt.de

¹ Computer Systems Group (RS), TU Darmstadt, Darmstadt, Germany

² Embedded Systems and Applications Group (ESA), TU Darmstadt, Darmstadt, Germany

Section 4. The integration of the configuration toolflow into the CAMEL-View IDE is presented in Section 5. Afterwards, Section 6 evaluates the CGRA with a focus on the scalability of the CGRA performance with the number of PEs. Then, the tool and application execution time for the given benchmarks are discussed. Finally, Section 7 analyses challenges arising from configuration and data memory handling and extensively discusses the lessons learned and their impact on further research activities.

2 Related Work

There are a few commercial solutions integrating a hardware-accelerator in a general purpose design or an environment for engineering control applications. For example, the Mathworks MATLAB and National Instruments Labview environments can be extended by FPGA-based front-ends like the dSpace DS5203 [3]. However, those accelerators either come along with a fixed set of predefined functionality (e.g. for signal conditioning), or they are limited to a small synthesizable block-set. In the latter case, changing the accelerated kernel requires a significant amount of time and the licenses for the FPGA vendor tools.

Besides commercial products, numerous CGRA architectures have been proposed in literature. Most of them are evaluated in a stand-alone fashion or based on simulation.

In [7], an array of functional units (DySER) is integrated in the execution stage of an OpenSPARC processor. A compiler detects compute-intensive code regions and maps those onto the array. Except for simple control flow structures, most of the control flow is handled by the OpenSPARC processor. Memory access has to be handled by the processor as well, and in every loop iteration each local variable has to be written to and read from the computation slices. The CGRA is completely integrated, but not suitable for real-time applications.

Fricke et al. present an automated toolflow that utilizes a CGRA as an overlay architecture on FPGA [5]. The authors also investigate the toolflow usability in terms of tool runtime [6]. However, it appears that the architecture does not support control flow and is lacking a verified realization on a chip.

A CGRA that is primarily constructed for mobile phones with the capability to process floating-point operands is contributed in [10]. The design is implemented with 130 nm CMOS technology and tested with JPEG and physics engine kernels. While achieving a significantly higher performance than an ARM9 for the given benchmark, the architecture is neither coupled with an actual host processor nor an automated tool for software integration is provided.

Karunaratne et al. focus on the dynamic reconfigurability of the interconnect between the PEs of the CGRA [9]. Using

a statically scheduled configurable crossbar switch, this architecture allows to transfer data between far apart PEs without routing through intermediate PEs. The performance evaluations is however limited to a rather small 4×4 CGRA configuration, thus the scalability of the concept to larger configurations can not be judged. However, even for this small configuration, tool runtimes larger than 1000 s are reported for some applications, which would not be acceptable for the application scenario targeted by UltraSynth (see Section 3).

Cong et al. present a fully pipelined CGRA [2]. It shows that incorporating pipelined operations can lead to significant increase in performance. However, the system lacks of applicability as a general purpose accelerator, since only innermost loops can be executed and control flow heavy applications cannot be mapped.

In [1], a framework for CGRA microarchitecture and scheduling research is presented. It introduces an architecture description language and interpreter to separate the microarchitecture design and configuration from its physical implementation. Furthermore, the CGRA-ME framework reads the application kernels to be mapped to the CGRA as abstract dataflow graphs. Regarding these two features, CGRA-ME follows a similar concept as UltraSynth (see Section 5). However, in contrast to UltraSynth, CGRA-ME does not consider the context of the CGRA, e.g. the communication with sensors, actuators, external memory and Microcontroller Unit attached to the CGRA on SoC. Moreover, the performance evaluation provided in [1] is limited to small arithmetic kernels with tens of operations. For UltraSynth, the acceleration of complete control engineering applications is investigated.

Only a few examples of existing CGRA research can be summarized here. However, recent surveys provide a broad overview on various CGRA concepts and discuss their performance and usability. For example, Liu et al. explicitly identify the necessity to improve the programmability and productivity of CGRAs [11]. Podobas et al. draw the same conclusion and encourage the research community to focus on more complex (real world) applications rather than studying simple and small CGRA kernels [12]. This work addresses both of these findings by describing the integration of the required CGRA tooling into a control engineering environment and by investigation scalability effects on tool runtime and CGRA performance.

3 Requirements and System Outline

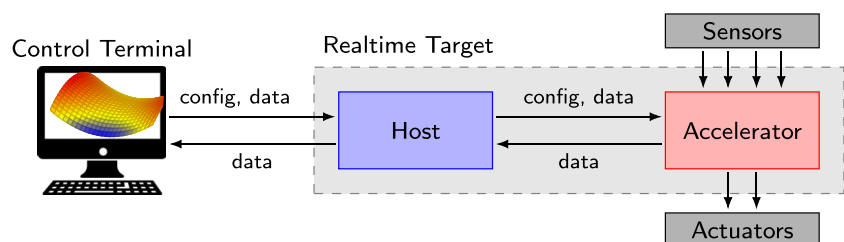
In CAMEL-View mechatronic systems are modeled as a hierarchy of interconnected subsystems with inputs, outputs, parameters, and internal states. Instead of explicitly resolving differential equations describing complex

physical behavior, the engineers formulate how the outputs and the first derivative of internal states are computed from the inputs, parameters, and the internal states. When generating code for executing these models either as simulation or on an embedded processor as part of the test rig, numerical integrators are required to derive the values of internal states after a certain time step from their current value and derivative, e.g. $x(t + \Delta t) = x(t) + \dot{x}(t) \cdot \Delta t$. Those integrators are not explicitly represented in the CAMEL-View models.

When executing control models on a test rig, CAMEL-View runs on a control terminal (e.g. an industrial PC), which is connected to a RT target over a wired network link as shown in Figure 1. An embedded system is used as RT target to fulfill the system requirements on jitter and fast peripheral I/O in the microsecond range. It consists of a host processor for the network communication stack and other basic computations, as well as an accelerator for compute-intensive application kernels and communication with peripheral devices (i.e. sensors and actuators). In this scenario CAMEL-View is also the user interface of the test rig capturing change requests for run-time parameters and visualizing data received from the RT target as charts or animations.

CAMEL-View control algorithms are processed in a loop with a fixed frequency. In each period, sensor inputs and/or data from the host processor is loaded, processed according to the control algorithm and the resulting values are sent to the actuators or back to the host processor. The accelerator can either execute the whole CAMEL-View model, such that the host processor is just used as a communication gateway. Alternatively, a control algorithm can also be distributed over the CGRA and the host processor, where the control cycle of the host processor can be an integer-multiple of the accelerator control cycle. This requires an efficient and well synchronized exchange of run-time data. More specifically, the accelerator has to distinguish between high priority inputs produced by the host in every control cycle and low priority parameter updates sporadically generated by the control terminal. In the reverse direction, accelerator results directly processed by the host have a high priority but typically a low volume, while computed results to be displayed at the control terminal have a lower priority but require a higher bandwidth, when lots of intermediate signals are to be observed.

Figure 1 Application scenario.



During the development of a test rig, structural modifications of the control models require the regeneration and redeployment of the configuration for the host processor and the accelerator. Besides the run-time data, the host processor thus has to be able to send application-specific configurations to the accelerator. The round trip time of such modification and reconfiguration cycles has to be in the orders of (tens of) seconds to ensure an acceptable and efficient workflow for the control engineers, which typically need to perform several test runs in a row. Thus, instead of using an High-Level Synthesis toolflow to translate and synthesize CAMEL-View models into bitstreams for FPGA-based hardware accelerators within several hours, a CGRA-based accelerator is exploited. Besides its fast mapping from abstract application descriptions to corresponding CGRA configurations, this approach is independent from any FPGA or ASIC synthesis tools and their required licenses, which was another main requirement within the UltraSynth project.

Finally, a CGRA Application Programming Interface (API) is required for the host processor to encapsulate all data and configuration transfers as well as the execution synchronization. Besides improving the portability to other target devices, this API hides CGRA-specific optimizations such as the rearrangement or duplication of data transfers (see Section 4.2) from the CAMEL-View backend developer.

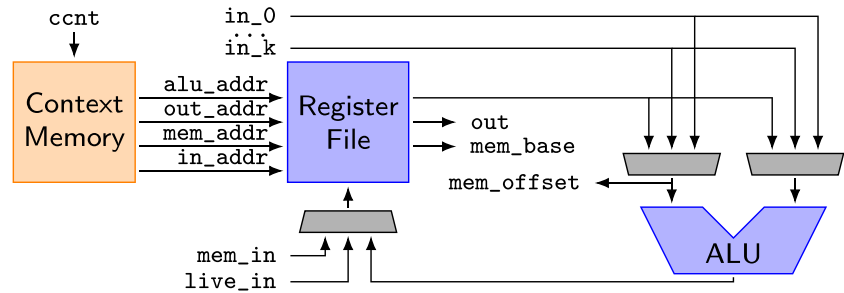
4 Hardware Integration

The micro-architecture of the proposed CGRA decouples the core computation (i.e. interconnected PEs and control flow modules) from the hardware interface to the host processor. As this paper focuses on the CGRA integration into the RT target, the core architecture is described only briefly. More details can be found in [14].

4.1 Micro-Architecture of the CGRA Core

The core of the CGRA is composed of PEs that process data and a Condition Box (C-Box) and Context Control Unit (CCU) that handle control flow. A PE consists of an Arithmetic Logic Unit (ALU), a Register File (RF), and a Context Memory (CMem), as shown in Figure 2. The

Figure 2 PE with access to external memory.



context indexed by the current context counter (i.e. *ccnt*) is loaded from the CMem in each clock cycle. Besides multiplexer settings, e.g. ALU operand selections, a context mainly defines the addresses for the read and write ports of the RF as well as the operation that is carried out by the ALU. Thus, one operation can be triggered per clock cycle per PE.

The ALU has a modular structure. Operands are selected by multiplexers in front of the ALU. They select either data from the RF or from inputs (i.e. $in_0 \dots in_k$) driven by the out-port of neighboring PEs. These are directly connected with zero latency. The multiplexer in front of the RFs selects either results from the ALU, data from an external memory or setup parameters (i.e. *live_in*). The setup mechanism is detailed in Section 4.2.

The CGRA is able to process kernels with heavy control flow and nested loops by using speculative computing and predicated stores in the PEs. The C-Box combines status signals from comparison operations and drives the store predication signals, as shown in Figure 3. Furthermore, branch selection signals drive the CCU, which performs branching. Consequently, all CMem of the PEs are controlled by the CCU with a common *ccnt* signal.

4.2 Peripheral Communication

To utilize the CGRA core as a hardware accelerator, it has to be properly integrated into a processing system. In this section, the integration of the CGRA core into a Xilinx Zynq SoC is described. An alternative system integration of the same CGRA core can be found in [15].

As described in Section 3 the accelerator has to pull sensor data, push actuator data, receive configurations and

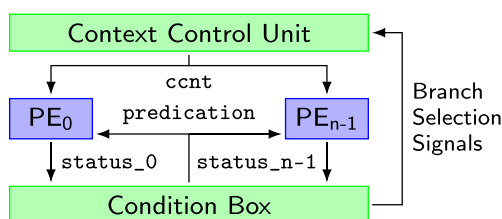


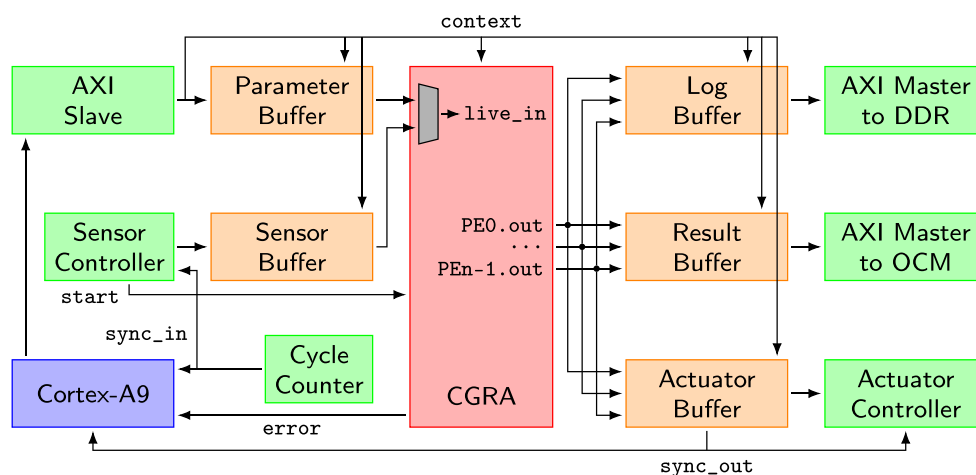
Figure 3 CGRA core overview.

run-time data from the host processor, and push back different kinds of computed results to the host processor. To support these data transfers within the Zynq SoC, additional hardware modules have to be arranged around the CGRA core as shown in Figure 4. Just like the PEs inside the CGRA, those modules include CMem to be configurable for a specific application. An Advanced eXtensible Interface Bus (AXI) interconnect is used to transfer configuration and run-time data from the host processor to the CGRA. Therefore, all CMem are memory mapped into the AXI address space by an appropriate *AXI slave* module.

To ensure the accurate timing of the periodic control loops (typically ranging between 1 and 10 kHz), a configurable hardware *cycle counter* provides a heartbeat for the overall system in terms of periodic *sync_in* pulses. Upon this trigger, the *sensor controller* captures new values from its attached peripherals and writes those samples into the BRAM-based *sensor buffer*. The sensor controller then signals the CGRA to start executing a control cycle. During this execution, the CGRA can read values from the sensor buffer. The required read address is provided by the CMem within the sensor buffer. The sensor data is transferred into the RF of one (or more) PEs via the *live_in* path (see Figure 2). The sensor sampling cannot be interleaved with a control cycle, as the scheduler has no information about the time required to capture specific samples. This sensor sample delay may even vary between successive control cycles.

Besides the sensor samples, run-time parameters sent from the host processor via AXI are used as inputs for the CGRA processing. Those parameters can be categorized into *constants* (loaded once for each application), *initial values* of internal states (loaded after each application reset), *run-time parameters* (updated sporadically upon user interaction at the control terminal), and *inputs from the host processor* (updated before every control cycle). In any case, those parameters must not be updated while a control cycle is executed, as the computed results would depend on the scheduled operator ordering and the exact time of the parameter update. The latter cannot be predicted accurately enough due to latency variations on the host processor and the AXI interconnect. Thus, a BRAM-based *parameter*

Figure 4 Communication between the CGRA and its peripherals (sensors, actuators, and host processor) on the Zynq SoC.



buffer is memory mapped into the AXI address space to delay all parameter updates until the end of the current control cycle. Besides this delay mechanism, the parameter buffer also contains a configurable table to map parameter indices (derived from the AXI write address) to the targeted PE index and the targeted address within the corresponding RF. As soon as the current control cycle is done, the buffered parameters are written to the RFs via the *live_in* path (see Figure 2). If a certain parameter has to be written to multiple RFs within the CGRA to reduce data copy operations at run-time, multiple AXI writes to the parameter buffer are generated by the host processor, as detailed in Section 5.3.

The parameter buffer can be configured such that the execution of the next control cycle is delayed until the expected number of inputs from the host processor has been received and transferred into the RFs. This mechanism is exploited to synchronize the execution of CGRA and the host processor when a control algorithm is distributed over both of them. However, the dynamic delay until all required inputs are received adds up to the possible uncertainty from the sensor sampling stage, which might cause the overall application period to be exceeded. To detect such timing violations at run-time, an *error* is signaled to the host processor if the CGRA receives another *sync_in* pulse while still executing the current control cycle. The accuracy of the control algorithm is not affected by a delayed start of the control cycle computations, as the actual sensor sampling is triggered independently by the *sync_in* pulse with fixed periodicity.

The *out* signal of the PEs (see Figure 2) are not only used to drive the *in* signals of their neighboring PEs, but also to push calculated results to the *actuator controller* and the host processor, as shown in Figure 5. Within the *actuator buffer*, the *buffer context* loaded for the current *ccnt* selects the correct PE output to be buffered (see Figure 5). For the last actuator sample generated within a control cycle, a

sync_out pulse is derived from the actuator buffer context to let the *actuator controller* transfer the buffered values to the attached peripheral devices.

To transfer CGRA outputs to the host processor, two different kinds of outputs are distinguished, as described in Section 3. *Result* outputs have to be available at the host processor before the next control cycle is started. Those results are typically intermediate values of a control loop distributed over CGRA and the host processor. As shown in Figure 6, one PE output per clock cycle can be pushed into the *result buffer*. To reduce the scheduling constraints, the results can be buffered out of order. The *output context* then forwards the results in order to the On-Chip Memory (OCM) within the Zynq processing system via a high priority *AXI master*, from where they can be accessed by the host processor. The output context cannot be indexed by the continuously incremented *ccnt*, as the AXI master must be able to stall the transfer as long as the AXI slave is not ready to receive more data. Thus, a *burst controller* drives the output context such that an AXI burst is generated, as soon as the slave is ready and all burst data is available in the result buffer.

The other type of PE outputs to be transferred to the host processor is referred to as *log data*. These are typically values to be monitored by the user at the control terminal or to be captured over a long run to retrace the overall

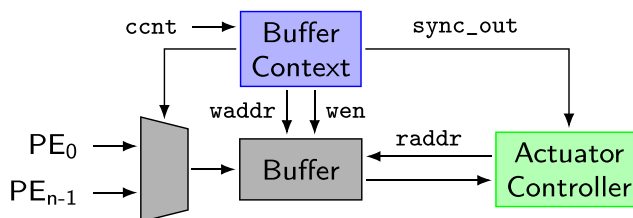
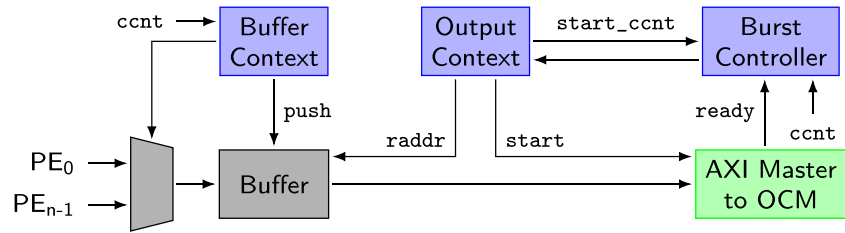


Figure 5 Actuator buffer.

Figure 6 Result buffer.



calculation. Compared to the result data discussed before, the *log data* transfer is not time critical, as the host processor does not have to respond immediately. On the other hand, the amount of *log data* is typically much larger than the amount of result data. Using the same approach for *log data* as for results would result in a bottleneck, as only one log datum could be buffered in every cycle. As most *log data* is generated near the end of a control cycle, this output generation could not be interleaved completely with other computations, and thus would increase the overall schedule length. To overcome this bottleneck, the input stage of the *log buffer* consists of a double buffer for each PE, as shown in Figure 7.

The buffers for write (PE to buffer) and read (buffer to AXI) access are switched after each control cycle. This allows to buffer more *log data* at once and furthermore, an entire control cycle can be used to transfer the log buffer content to the host processor. This is actually done by writing the *log data* and the index of the current control cycle into a circular buffer in the SoC-external Double Data Rate (DDR) memory via a low priority *AXI master* to not interfere with the transmission of the result data. The host processor is thus relieved from immediately processing the received *log data*.

5 Toolflow Integration

The CGRA-specific toolflow partially integrated into CAMEL-View is shown in Figure 8. The main toolflow is implemented in Java, as portability and reusability are considered more important than the tool run-time at the

moment. It is divided into two interconnected toolchains for generating the CGRA composition bitstream (i.e. Steps 1 to 3, described in Section 5.1) and the application-specific CGRA configuration (i.e. Steps 4 to 10, described in Section 5.2). The latter can be executed on the control terminal from within CAMEL-View. The final download of the configuration is part of the API used by the host processor and described in Section 5.3.

5.1 CGRA Model and Bitstream Generation

Based on the micro-architecture described in Section 4.1, a framework was implemented to model arbitrary CGRAs. These can have an irregular interconnection and heterogeneous provision of operations in PEs. Also, an arbitrary number of PEs is supported. A model is used to represent a CGRA, which can be passed to a Verilog generator to generate HDL code. The same model is also used for scheduling. This allows to systematically research CGRA-structures and analyse the impact on scheduling quality without requiring actual hardware. The discussed CGRA can be also used in another system [15]. In order to support both integrations, the model of CGRA only represents its core (as described in Section 4.1). In order to actually use a CGRA (model), one has to use the instance for a specific system-integration. These are nothing but inheritances of the core model, containing additional information on communication interfaces and control modules.

CGRA instances can be modeled either manually by providing a corresponding JavaScript Object Notation (JSON) description, or automatically optimized for a set of applications, which is however beyond the scope of this

Figure 7 Log buffer.

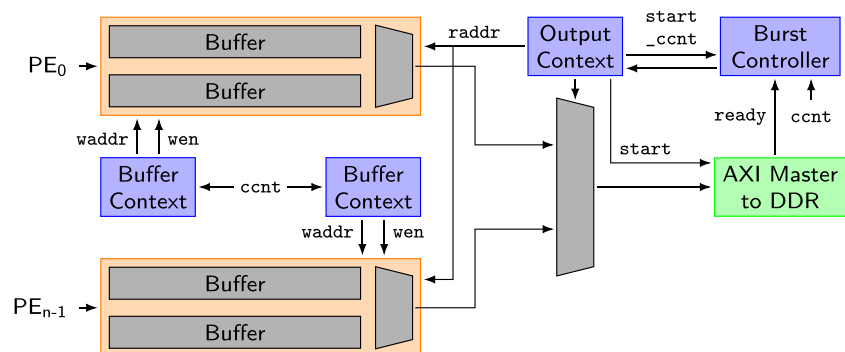
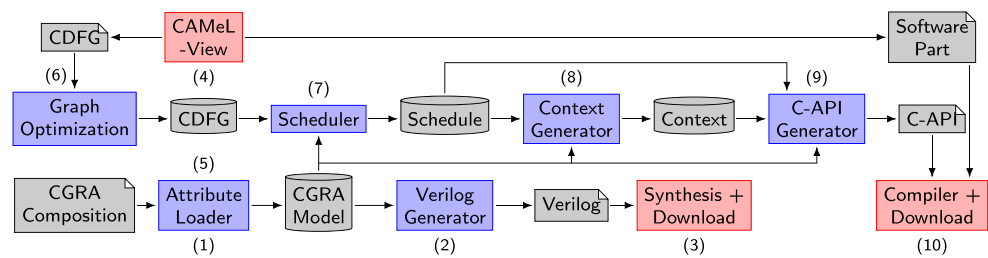


Figure 8 Toolflow for generating a Verilog bitstream from a CAMEL-View composition (1 to 3) and CGRA contexts for a specific CAMEL-View application (4 to 10). Third-party tools (red) are interconnected by the CGRA-specific tools (blue) implemented in Java.



paper. After parsing the composition into the CGRA model by the *attribute loader* (Step 1), corresponding Verilog code is generated (Step 2). The resulting hardware description and constraints also include all required modules for interfacing the CGRA to the host processor and to peripheral sensors and actuators as described in Section 4.2. Finally, the bitstream is generated and downloaded to the Zynq device using the Xilinx Vivado tool-chain (Step 3).

5.2 Application Scheduling and CGRA Configuration

The second toolchain starts with the application modeled in CAMEL-View (Step 4). Within this environment, the submodules of the control algorithm to be accelerated by the CGRA can be selected. These computational kernels are then exported as a Control and Data Flow Graph (CDFG), which is represented in JSON along with other settings like the numerical integrator details. All remaining parts of the application are exported as C-functions to be executed on the host processor. Furthermore, the user selects a specific RT target (i.e. Zynq device), which is already programmed with the synthesis result from the bitstream toolchain. The corresponding CGRA composition associated with the selected RT target is then loaded, again as Java object model (Step 5).

Afterwards, the application CDFG is parsed and optimized into a Java object model (Step 6). Those optimizations comprise typical compiler passes (e.g. common sub-expression elimination, constant propagation, dead code elimination, and bitwidth optimization) and other steps required to adjust the CAMEL-View execution model to the CGRA execution model. For instance, selection operations (e.g. $a = b ? c : d$) have to be transformed into predicated store operations (e.g. $\text{if } (b) \ a = c; \text{ if } (!b) \ a = d$). Furthermore, update operations for the CDFG state variables have to be inserted depending on the user-selected integrator type. Currently, first and second order integrators (i.e. Euler and Heun [4]) are supported. All integrator steps rely on the calculation of the time derivative for the internal states, which is already part of the CDFG generated by CAMEL-View.

The optimized CDFG and the CGRA model are then fed into the *scheduler* (Step 7), which is based on list scheduling with additional constraints to cope with routing resources

and inter-PE data transfers [13]. This approach produces very good results in short time [14]. Since the CAMEL-View models are based on reading and writing physical inputs and outputs, additional input and output operations need to be scheduled. The input operation allows a PE to read a value from the sensor buffer, as shown in Figure 5. Output operations write the actuator, result, or log buffer, or even to multiple of them at once. As the result and actuator buffer may only be written by one PE at a time, the scheduler has to avoid write conflicts on these buffers. Furthermore, data structures required to map the buffered values to their respective CDFG nodes are generated.

Based on the scheduled input, output and arithmetic operations as well as the targeted CGRA model, the context information required to let the CGRA execute the current application is generated (Step 8).¹ The CMem content for PEs, C-Box, CCU, and the peripheral buffers described in Section 4.2 is generated as a Java object model at this stage. In Step 9, this context information is exported into C structures as part of the CGRA API for the host processor (see Section 5.3). Besides the context data, the *C-API generator* exports more scheduling details about the RF allocation to hide the mapping of inputs, parameters, and constants to certain PEs from the host application.

Finally, the C-API is combined with the software part of the CAMEL-View application and fed into the ARM compiler (Step 10). The resulting binary is downloaded to the host processor and executed on one ARM Cortex-A9 core. The other core is interfacing the control terminal and does not need to be reprogrammed when modifying the target application.

5.3 CGRA API

The software API for the host processor provides methods to configure the CGRA, to register interrupt handlers, to start and stop the periodic execution of the accelerated application kernel, to transfer run-time data to the CGRA, and to read result and log data from the OCM and DDR memory. All transfers from the host processor to the CGRA exploit the Cortex-A9 Direct Memory Access controller.

¹On the basis of step 7 and 8 it can be seen that the use of a generic CGRA model allows the use of any feasible CGRA composition

Listing 1 Minimal usage example for the CGRA API.

```

1 volatile uint8_t nextCycle = 0;
2 void handler(void *data) {nextCycle = 1;}
3 void main() {
4   cgra_setupData.logLowerAddr = 0x14000000;
5   cgra_setupData.logUpperAddr = 0x147FFFFFFF;
6   cgra_setupData.logIncrement = 0x1000;
7   cgra_setupData.mainPeriod = 100000;
8   cgra_setup();
9   cgra_onSyncIn(handler)
10  cgra_changeStateRun();
11  while (1) if (nextCycle) {
12    nextCycle = 0;
13    cgra_updateRuntimeParameters();
14    for (int i=0; i<CGRA_LOG_RESULT_COUNT; i++)
15      printf("%f\n", (float*) cgra_readLogData(i));
16    cgra_stepLogWindow();
17  }
18 }

```

They are combined to AXI burst transfers as far as possible. Both mechanisms are transparent to the API user. The API is executed bare-metal on the ARM.

A minimal usage example is shown in Listing 1. After defining the boundaries of the circular log buffer in the DDR memory region (Lines 4 to 6) and the number of clock cycles per control cycle (Line 7), all configurations (i.e. context data and initial RF values) are transferred to the CGRA in Line 8. In Line 9, the interrupt handler for the `sync_in` pulse is registered and the control cycle is started in Line 10. After each `sync_in` pulse, all run-time parameters modified by the control terminal (i.e. on the second ARM core, not shown in Listing 1) are transferred to the CGRA in Listing 13. A dirty flag mechanism is used to avoid superfluous parameter transfers. Furthermore, one parameter update might result in multiple AXI transfers, if this parameter is scheduled to multiple PEs inside the CGRA. The mapping of parameters to (multiple) AXI addresses is based on the tables exported by the C-API generator. Finally, after reading the log data received from the CGRA during the last application cycle (Line 15), the API-internal pointer to the current window in the circular log buffer has to be forwarded (Line 16).

6 Evaluation

This section gives a detailed evaluation of the CGRA tool and performance. First, the run-time of the tool and the application execution time are analyzed for a single CGRA composition for six given benchmarks. Afterwards, the impact of the number of PEs (ranging from 4 to 49) on the tool run-time, synthesis results and execution time of the the most stressing benchmark (i.e., *Half Axle dynamic*) are analyzed. This evaluates the scalability of the presented concept. The analyzed CGRAs (and its peripherals described in Section 4.2) were synthesized for a Xilinx XC7Z045-2 SoC using Vivado 2019.2.

For this evaluation, the accelerator is generated (as described in Section 5.1) to process single precision floating-point numbers, as it is required for the targeted application domain. When drafting and evaluating their algorithms, control engineers are not able to focus on numerical stability issues arising from fixed-point arithmetic or hardware-optimized trigonometric operations. This should explicitly demonstrate real-world applicability and its challenges.

Table 1 summarizes all common parameters for all CGRAs. They are chosen such that all benchmark applications used for the performance evaluation in Section 6.4 can be mapped on the same CGRA. As shown in Figure 9, each PE is connected with its eight neighbors in a matrix structure with horizontal and vertical wrap-around. This matrix star toroidal interconnect was chosen as the best performing solution while keeping the number of PE inputs treatable. It might be obvious that an irregular interconnect and heterogeneous operator provision in the PEs potentially improves the CGRA performance, but this is beyond the scope of this paper.

Table 1 Common CGRAs settings. Operators are annotated with their single precision floating-point latency (cycles).

Operators	ADD(8), SUB(8), MUL(8), DIV(16), OR(1), NEG(1), ABS(1), SGN(1), SQRT(54), IFLT(4), IFGT(4), SIN(36), COS(36), ASIN(37)
PE Interconnect	Matrix Star Toroidal
RF	256 entries (32 bit each)
C-Box Size	64 entries
CMem Size	8192 entries
I/O Buffer Size	32 entries

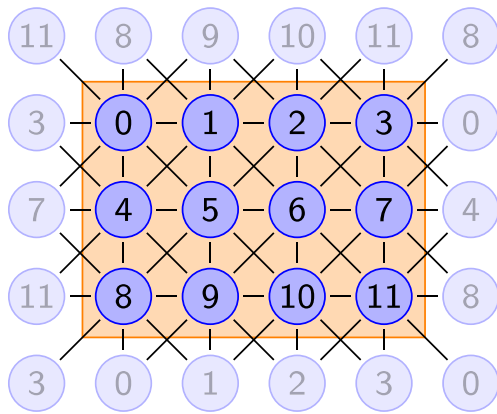


Figure 9 Interconnect topology: 3×4 matrix star toroidal.

6.1 Benchmarks

Six representative benchmarks were picked from the given set of benchmarks typically used in CAMEL-View. *Single Pendulum* calculates the behaviour of a swinging Pendulum—and *Double Pendulum* vice versa for a pendulum with two joints. *DC Motor Control* computes a control algorithm for a DC motor. These three represent small applications with less than 500 nodes. *Half Axle minimal* is a medium sized application with 1624 nodes. It simulates the behaviour of a suspension on one side of an axis of a vehicle. It is a physically simplified version of the big application *Half Axle dynamic*. The CDFG of that challenging application is about 500 times larger (in terms of nodes) than the simple academic examples (like the *Single Pendulum*). The *FIR 1000* is a finite impulse response filter of the order 1000 and a medium sized benchmark.

6.2 Tool Execution Time

As described in Section 3, one goal of this project is to minimize the time required to map control algorithms to the accelerator. To evaluate the fully automated tool run-time, the six chosen CAMEL-View applications have been mapped to the 5×5 CGRA. This composition was chosen as it performs best in terms of application run-time for the largest benchmark (see Figure 15) and as it is one of the largest composition actually fitting on the target device (see Figure 12).

As shown in Figure 10, the CDFGs to be handled by the scheduler range from several dozen nodes to more than 10 000 nodes. The Java tools described in Section 5.2 were executed on an Intel i7-8700k running at 3.7 GHz. The presented run-times are the average over 10 executions. For the smaller CDFGs, the file I/O dominates, but the overall run-time is neglectable. The run-times of the scheduler and

context generator increase significantly with the CDFG size. However, providing a complete configuration within 81 s for the largest example is still two orders of magnitude faster than generating a corresponding FPGA bitstream. It should be mentioned that the run-time of the tool holds notable potential to be reduced. The model, Verilog generator and the scheduler are designed to be easily adjustable by using object oriented hierarchies and modular code structures. In general, the coding style prefers understandability and maintainability over high performance. E.g. the scheduler has a modular structure to allow different and new heuristics for the attraction-criterion (see Ruschke et. al. [13]).

6.3 Application Execution Time

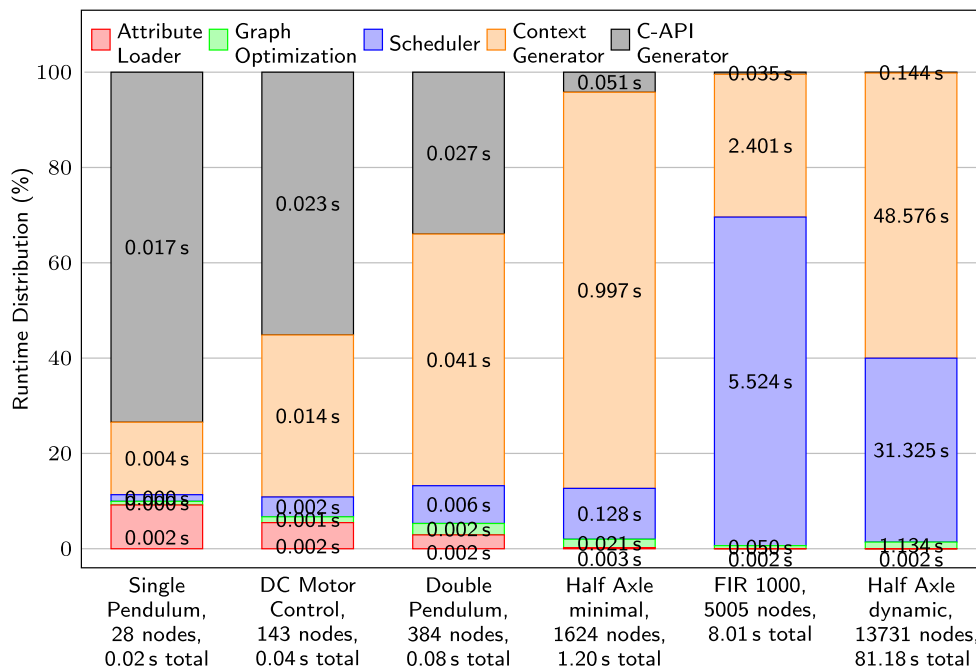
The time required to execute a single control cycle is limiting the integration step size and is thus considered the main performance metric. The average execution time of one cycle on the ARM Cortex-A9 processor (running at 800 MHz) *after* cache initialization is taken as the performance baseline. The six benchmarks are run out of CAMEL-View, which uses the GCC compiler with an O2 optimization. Those applications are not using sensors or actuators, as this comparison should focus on the computational performance of the architecture itself. The actual inputs and outputs of the execution cycles are provided and received by the host processor instead. The speedups shown in Figure 11 relate to the execution time of the benchmark applications on the 5×5 CGRA composition running at around 75 MHz (see Section 6.4.1).

It is important to note that the RF entries were set to 256 for all benchmarks to present one CGRA that allows mapping of all benchmarks (as specified in Table 1), while *Half Axle dynamic* is the only actually requiring that size. So application-specific CGRA compositions would yield much better speedups mainly for the smaller kernels (see Section 7.3.3), but this would contradict the main goal of supporting fast switches between the applications.

6.4 Scalability

The performance of the CGRA does not stand out in contrast to the ARM core. This can be explained by some observations besides the clock frequency of below 100% and non-optimal latencies of some operators (e.g. 54 clock cycles for an SQRT). The easiest ones are the fact that the ARM cores clock speed is about eight times higher and the combination of its dual issue architecture. Additionally, small to medium kernels provide relatively low parallelism. For instance, the scheduling length for *Single Pendulum*, *Double Pendulum* and *DC Motor Control* does not improve for CGRAs with more than 9 PEs. An analysis tool integrated into CAMEL-View could be used

Figure 10 5 × 5 CGRAs tool run-time: The specified node value indicate the size of the CDFG.



to decide whether kernels should be run on the Central Processing Unit or on the CGRA. In contrast, *Half Axle dynamic* and *FIR 1000* on the other hand provide a lot of instructions that can be computed in parallel for most of the time of the schedule. However, due to the large amounts of data, many of them are information for the control terminal, need to be written back from where they have been modified to their initial PE and RF address of a control cycle. This overhead makes up to 20% of the schedule.

The following analysis concentrates on *Half Axle dynamic*, since it benefits from an increasing number of PE even beyond the capabilities of the target FPGA. This leads to insights which can not be observed with perfectly suited (smaller) applications. All CGRAs used for the discussion of the different compositions are still based on the parameters shown in Table 1.

6.4.1 Scalability—Synthesis

Figure 12 illustrates the Lookup Table (LUT) and BRAM consumption and Figure 13 shows the minimum clock period possible for each individual CGRA. As expected the

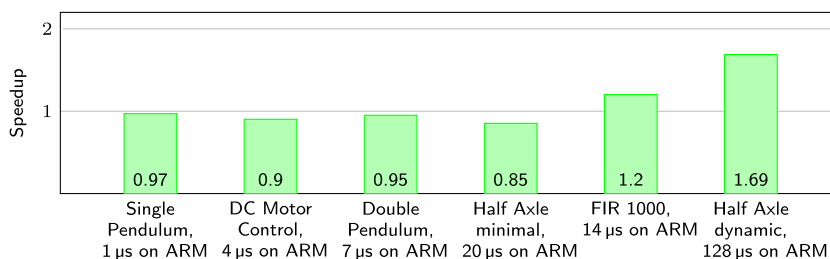
resource utilization increases linearly with the number of PEs. Up to 26 PEs can be implemented on the Xilinx Zynq FPGA before running out of LUTs. The figure also shows the resource utilization even beyond 26 PEs to estimate some trends, e.g. BRAM resources are still not limiting at 36 PEs.

The clock period for CGRAs with more than 26 PEs cannot be derived since they cannot be implemented due to the mentioned over-utilization. The default strategy for synthesis and implementation were used. The clock period is below 13.4 ns (i.e. 75 MHz) for all CGRAs except of the 5 × 5 instance, that is slightly above. As expected, the maximum clock frequency decreases with rising number of PEs, yet it does not drop significantly, which is a pleasing observation.

6.4.2 Scalability—Mapping of *Half Axle Dynamic*

The most interesting metric is the execution time of an application when analysing scalability. Figure 14 shows the execution time in clock cycles and Figure 15 shows the absolute execution time in μs .

Figure 11 5 × 5 CGRA application execution time: Speedup compared to ARM Cortex-A9.



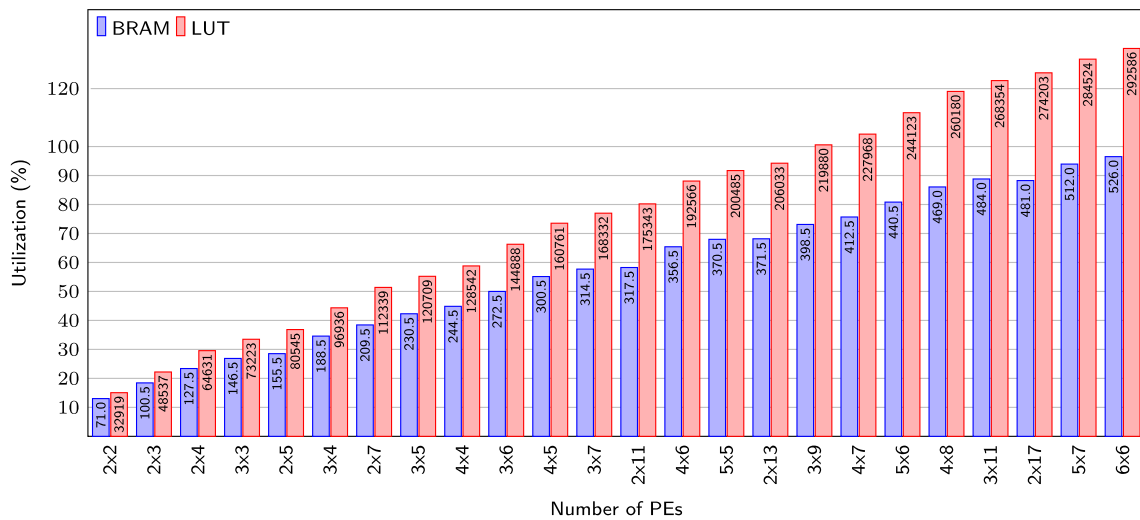


Figure 12 CGRA LUT and BRAM consumption for instances between 4 and 36 PEs.

The required clock cycles clearly show a decrease, meaning there is still unexploited parallelism even for CGRAs with up to 49 PEs and beyond. In general, it can be seen that CGRAs that have a squared structure perform better than longish structures. This is probably due to routing congestion, since squared instances allow better data locality which leads to shorter schedules. When a CGRA is meant to be regular and homogeneous, squared structures are the better choice.

The best performing instance in terms of absolute execution time is the 5 × 5 PE composition, since it is one of the largest compositions still fitting onto the FPGA. Also the larger (but longish) 2 × 13 PE composition can also be implemented, it yields a significantly worse scheduling length.

6.4.3 Scalability—Tool Run-Time

In order to verify that the simple upscaling of PE numbers is a valid option to increase performance, the run-time of the toolchain has to be analyzed as well. By increasing the number of PEs, the search space for the scheduler becomes larger. This increases the run-time of scheduling and context generation.

Figure 16 shows the total run-time of the toolchain over the number of PEs. Fortunately, the run-time increases fairly linear. The reason is most likely the attraction criterion used during scheduling. Not all PEs are considered to place a node, but only PEs in the neighborhood of the PE to which the predecesing node was mapped [13]. Peaks can be observed for instances that have a longish structure.

Figure 13 CGRA minimum clock period for instances between 4 and 36 PEs.

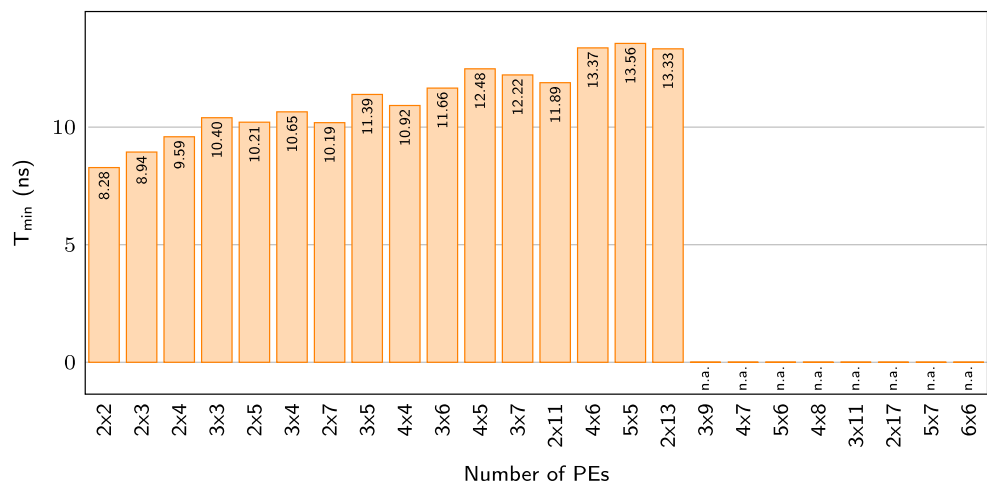
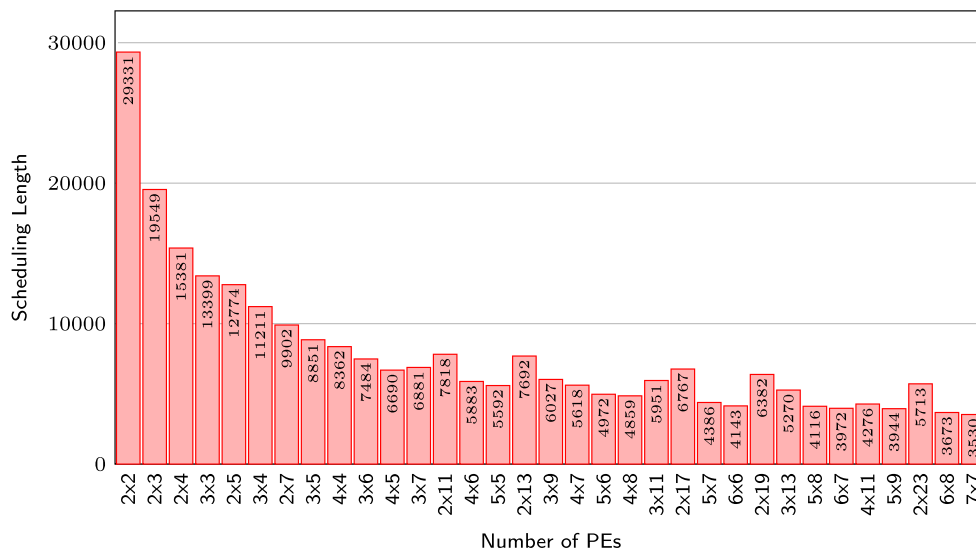


Figure 14 Execution time in clock cycles of *Half Axle Dynamic* for up to 49 PEs.



Since List-Scheduling is used, the run-time of mapping also correlates with the increased scheduling length for these longish compositions.

In conclusion, one can summarize that upscaling the number of PEs improves performance.

7 Insights

Section 6 indicates that upscaling is a valid choice to improve performance. However, this leads to drawbacks and several challenges, which also hinder a more significant increase in performance. These need to be tackled to provide a fully capable reconfigurable accelerator. Following, these challenges and the main lessons learned about the practicable application of CGRAs are discussed.

7.1 Requirements of Real-World Applications

Many CGRA designs focus on integer arithmetic due to their efficient FPGA implementation and short latencies yielding a good exploitation of the fast reconfiguration. Besides the presented work, [10] is one of the rare publications explicitly stating that floating-point operations are supported. More specifically, [12] reports that floating-point support is clearly under-represented. Less than 25% of the published CGRAs support floating-point computations. However, floating-point arithmetic and complex operations have to be supported, especially if a CGRA wants to qualify as a multi-purpose accelerator. An integer or fixed-point based CGRA could not have been used for the presented integration, since it would not provide enough precision to ensure that the discussed control engineering

Figure 15 Run-time of *Half Axle Dynamic* in clock cycles for up to 49 PEs.

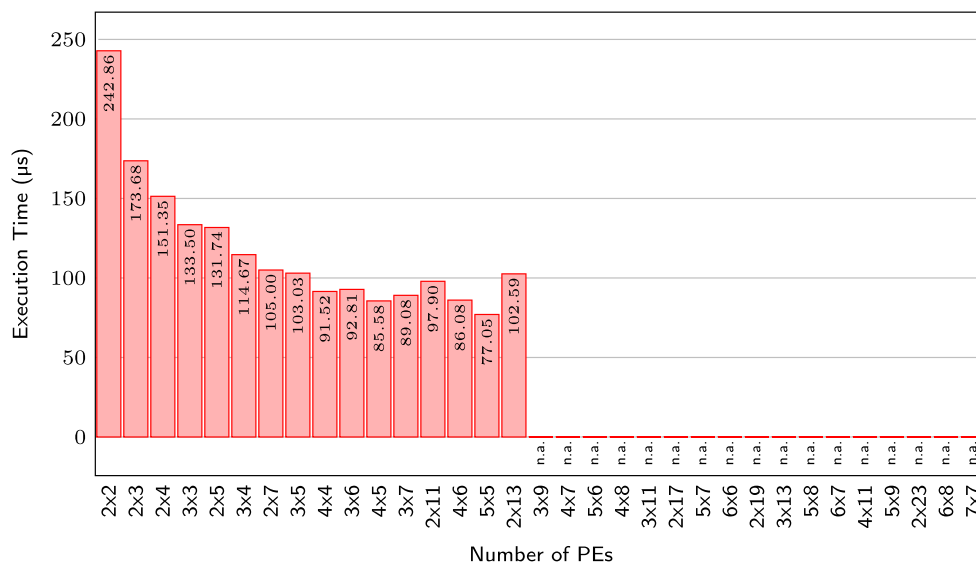
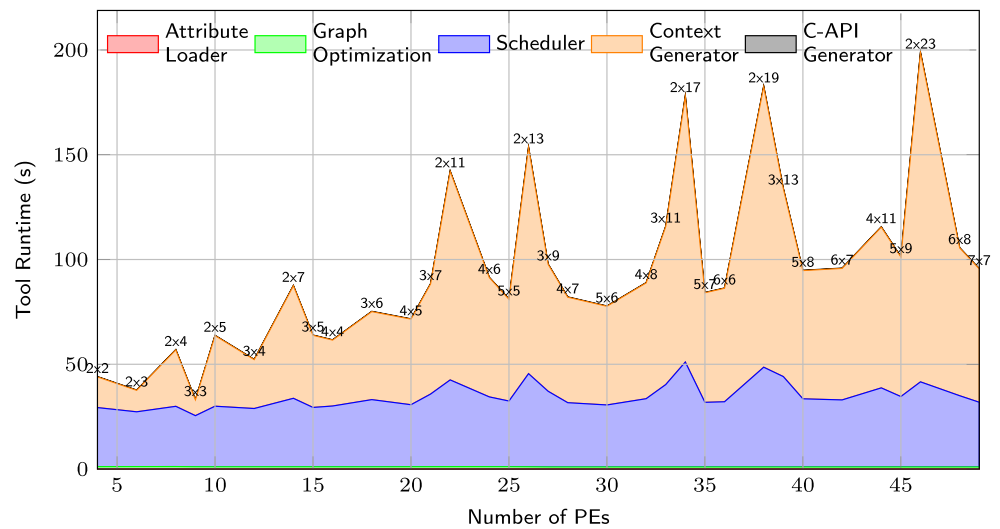


Figure 16 Run-time of the toolchain for the *Half Axle Dynamic*.



applications run stable without exploding rounding errors. Additionally, each of the discussed applications requires complex operations like division, square root or at least one trigonometric operation. Complex algorithms promise parallelism to be exploited by the accelerator, but lead to challenges.

On software or profiler level, an automated integration is inevitable for the native use of the accelerator. A shortcut would be the use of pragmas or a manual incorporation of the accelerator. Yet, many CGRAs are not even integrated into a system [1, 9, 10]. In contrast, the presented accelerator is an integration that works automatically out of the CAMEL-View IDE.

7.2 Impact of Real-World Applications

7.2.1 Long-Running Operations

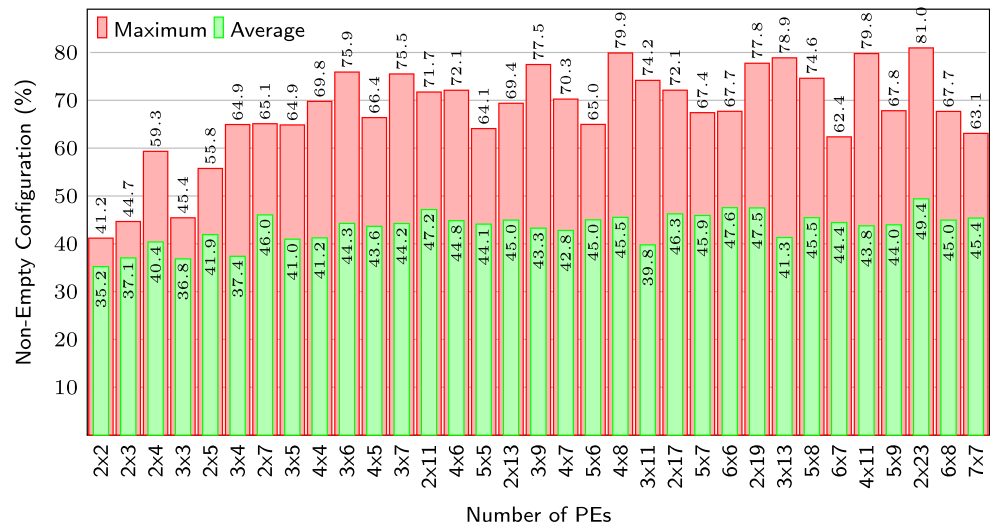
Floating-point arithmetic as well as complex (e.g. trigonometrical) operations require up to dozens of clock cycles, as shown in Table 1. This challenges the initial CGRA concepts of simple PEs processing data in parallel. Currently, only one operator can be active at any time in each PE. This works for computation using integer operations, since they usually have a latency of one or two cycles. As a consequence, long-running operations result in a poor PE utilization, since a PE is blocked until the computation has finished. Emulating complex operations as combinations of integer operations on CDFG level (like math libraries do on software processors) increases the schedule length significantly. Therefore, they have to be supported directly. Not supporting these operations at all limits the applicability of the CGRA to academic applications as we have argued.

7.2.2 (Re)configuration

Reconfiguration in every clock cycle allows a simple design of the CCU. This is reasonable as long as a new operation can be carried out in almost every clock cycle. This does not apply in the case of floating-point arithmetic. Medium sized kernels like *Half Axle minimal* easily require thousands of CMem entries per PE, as CAMEL-View eliminates control flow by unrolling all loops besides the main numerical integration step. A fictive (yet representative) control flow algorithm calculated in 100 μ s on the CGRA requires 10 000 clock cycles assuming a clock frequency of 100 MHz. A 5×5 CGRA with 32 bit wide CMems requires 1 MB of configuration data just for the PEs. Besides occupying a significant part of the BRAM resources, transferring a large amount of configuration data becomes time critical for systems frequently switching between different accelerator kernels at run-time. Amortizing the configuration time (which might also include just-in-time scheduling) is only possible for long-running and/or frequently executed accelerator kernels. In contrast to the kernels generated by CAMEL-View, applications with (nested) loops significantly improve the ratio between kernel run-time and configuration data.

None the less, this overhead is neglectable for the presented application scenario, because the configuration overhead appears only once for a hardware in the loop system that might run many minutes or even hours. Moreover, the CGRA accelerator can satisfy certain timing constraints of control algorithms (especially regarding low jitter) that the ARM processor might not satisfy at all, thus expanding the addressable application space.

Figure 17 Naive configuration compression—analysis for 4–49 PE for *Half Axle Dynamic*.



7.2.3 Data Management

Usually, the required size of the RF correlates with the size of an application. 256 RF slots per PE are required for the *Half Axle dynamic* based on a variable lifetime analysis. Since the RF is read combinatorially and lies on the critical timing path of the CGRA, enlarging the RF decreases the maximum clock frequency.

7.3 Consequences

7.3.1 Long-Running Operations

Techniques like pipelining and interleaving operations promise an increase of the PE utilization. Furthermore, this would reduce the number of unused CMem entries (i.e. empty NOP cycles) thus also improving the CMem utilization. However, a new PE concept and a notably more complex scheduling is required for these techniques. Although this contradicts the requirement of fast application mapping, an improved scheduling is mandatory to be competitive on the performance level.

7.3.2 (Re)configuration

If PEs do not require reconfiguration in every cycle, i.e. loading empty configuration cannot be avoided, a novel strategy for skipping the PE context switches is demanded. One approach could be an individual counter for each PE stalling its local `cnt` increment, but not the ALU operation. Obviously jump addresses would need to be computed locally for each PE. This technique would require a synchronization of PEs which is hard to realize for applications with a data dependent control flow. The amount

of configuration data can thus be kept manageable for applications exploiting (nested) loops.

In order to get an estimation of how many configuration can be spared, the amount of non-empty configuration are illustrated in Figure 17. It can be seen that over half of the configuration data could be spared on average for *Half Axle Dynamic*. The (red) *Maximum* bars indicate the configuration utilization of the busiest PE. Regarding the 5×5 PE composition evaluated in Section 6.3, the occupied CMem space would be reduced from 5592 to 3585. Without compression, the CGRA would require 16k CMem entries to load all presented benchmarks at once. Being compressed, 8k entries would be sufficient to hold all presented benchmarks simultaneously.

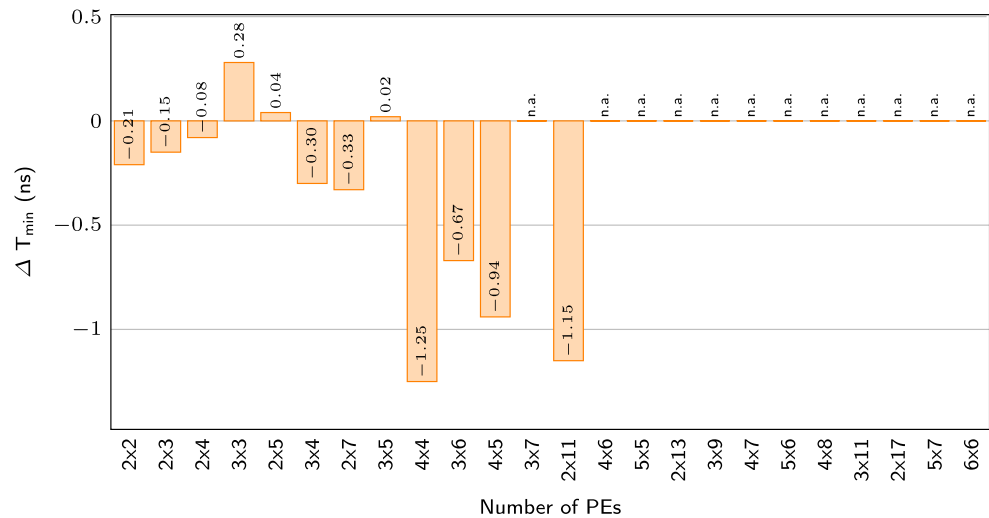
The impact of the reduction of the memory size from 16k to 8k on the CGRA performance is shown in Figure 18. The clock period of some composition can be improved by over 1 ns. Additionally, CGRAs with 16k CMem entries and more than 22 PEs do not fit into the FPGA anymore, since LUT resources are over-utilized.

This compression analysis considers only configuration that have absolutely no content, but there is much more potential. For instance, the write address for the RF needs to be loaded separately, if the operation requires more than one cycle. Instead, the RF write address could be loaded with the opcode and stored within operators and passed to the RF once the operation has finished.

7.3.3 Data Management

Aside from a suitable PE interconnect, local data management needs to be considered. On the hardware level, BRAMs can be attached to each PE to support a spill and fill strategy. Using a stack could also be an alternative.

Figure 18 Decrease of the clock period if the CMem size is reduced from 16k to 8k—RF size is kept at 256.



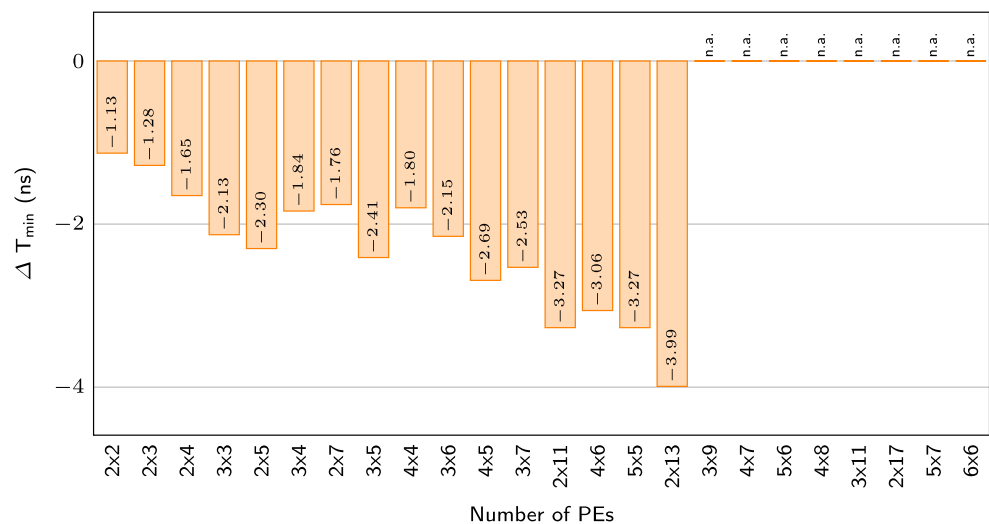
The memory transfers might enlarge the schedule, but could be outweighed by a higher clock frequency. For example, reducing the RF size from 256 to 64 slots would reduce the clock period by about 15% to 30% for the presented CGRAs, as shown in Figure 19. This would allow to run the CGRA at 100 MHz and above. As an alternative, scheduling could be improved to use lifetime analysis on the flight rather than afterwards during context generation. However, this drastically worsens the tool run-time.

Many cycles at the end of a schedule are currently used to copy global state variables back to their initial location (i.e. PE and RF slot) for the next control cycle. Large numbers of PEs further amplify this effect. The same effect can be observed with local variables in loop structures. Besides the already mentioned resource utilization, these are two additional reasons why increasing the number of PEs causes drawbacks to be overcome with further micro-architecture and tooling improvements.

8 Conclusion

This contribution presents an evaluation and insights of a CGRA-based hardware accelerator for control engineering applications. Real-world applications have hard demands and challenge the concept of CGRAs. They majority of the data and configuration management needs to be handled differently than most CGRA concepts suggest. The demand for large amounts of configuration data calls for new configuration mechanisms in order to spare cycles, which do not require (re)configuration. Bigger configuration memories result in a clock frequency decrease of about 10% and hardly limit scalability due to resource restrictions. The control terminal is used by an engineer to supervise the execution on the test rig. Large amounts of data need to be intermediately stored, until they can be send to the host, thus putting pressure on the data memories in the CGRA. This leads to almost infeasible big RFs, even

Figure 19 Decrease in clock time period if RF entries are reduced from 256 to 64—CMem size is kept at 8k.



though output buffers are used to reduce that pressure. The result is a significant drop of the clock frequency (up to 30% loss for an increase from 64 to 256 RF entries). CGRA concepts should consider how that large amount of configurations and data can be handled efficiently instead of just increasing the memories.

Our future work will concentrate on a novel architecture that is able to cope with the presented challenges. The focus will lie on the optimization of the CGRA core (pipelined operations, fusing operators like ADD and SUB, allow interleaved operations and additional memories to reduce load on RFs) and scheduling (a novel and more fine grained modeling of the CGRAs to aim for better utilization). Additionally, methodologies to explore the design space of heterogeneous PE operator sets and irregular interconnects will be implemented.

Acknowledgements This research was funded by the German Federal Ministry for Education and Research with the funding ID 01 IS 15020 and supported by iXtronics.

Funding Open Access funding enabled and organized by Projekt DEAL. This research was funded by the German Federal Ministry for Education and Research with the funding ID 01 IS 15020.

Availability of Data and Material Please contact the authors for data requests.

Code Availability The implemented framework is 100% custom Java code and not publicly available. Please contact the authors for code requests.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

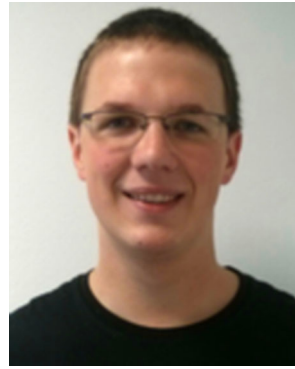
References

- Chin, S.A., Sakamoto, N., Rui, A., Zhao, J., Kim, J.H., Hara-Azumi, Y., Anderson, J. (2017). Cgra-me: a unified framework for cgra modelling and exploration. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)* (pp. 184–189).
- Cong, J., Huang, H., Ma, C., Xiao, B., Zhou, P. (2014). A fully pipelined and dynamically composable architecture of CGRA. In *IEEE international symposium on field-programmable custom computing machines* (pp. 9–16).
- dSpace: DS5203—FPGA programmable per application (2018).
- Fathoni, M.F., & Wuryandari, A.I. (2015). Comparison between Euler, Heun, Runge-Kutta and Adams-Bashforth-Moulton integration methods in the particle dynamic simulation. In *International conference on interactive digital media* (pp. 1–7).
- Fricke, F., Werner, A., Shahin, K., Huebner, M. (2018). CGRA tool flow for fast run-time reconfiguration. In N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos, P.C. Diniz (Eds.) *Applied reconfigurable computing. Architectures, tools, and applications* (pp. 661–672). Springer Int. Publishing.
- Fricke, F., Werner, A., Shahin, K., Werner, F., Hübner, M. (2019). Automatic tool-flow for mapping applications to an application-specific cgra architecture. In *2019 IEEE international parallel and distributed processing symposium workshops (IPDPSW)* (pp. 147–154).
- Ho, C.H., Govindaraju, V., Nowatzki, T., Marzec, Z., Agarwal, P., Frericks, C., Cofell, R., Benson, J., Sankaralingam, K. (2015). Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation. *Energy (mJ)*, 5(10), 15.
- iXtronics: mechatronics, tools & technologies. <http://www.ixtronics.com>.
- Karunaratne, M., Mohite, A.K., Mitra, T., Peh, L. (2017). Hycube: a cgra with reconfigurable single-cycle multi-hop interconnect. In *2017 54th ACM/EDAC/IEEE design automation conference (DAC)* (pp. 1–6).
- Lee, D., Jo, M., Han, K., Choi, K. (2009). FloRA: coarse-grained reconfigurable architecture with floating-point operation capability. In *International conference on field-programmable technology* (pp. 376–379). <https://doi.org/10.1109/FPT.2009.5377609>.
- Liu, L., Zhu, J., Li, Z., Lu, Y., Deng, Y., Han, J., Yin, S., Wei, S. (2019). A survey of coarse-grained reconfigurable architecture and design: taxonomy, challenges, and applications. *ACM Computing Survey*, 52(6). <https://doi.org/10.1145/3357375>.
- Podobas, A., Sano, K., Matsuoka, S. A survey on coarse-grained reconfigurable architectures from a performance perspective preprint (2020).
- Ruschke, T., Jung, L., Wolf, D., Hochberger, C. (2016). Scheduler for inhomogeneous and irregular CGRAs with support for complex control flow. In *IEEE international parallel and distributed processing symposium workshops* (pp. 198–207).
- Ruschke, T., Jung, L., Hochberger, C. (2017). A near optimal integrated solution for resource constrained scheduling, binding and routing on CGRAs. In *IEEE international parallel and distributed processing symposium workshops* (pp. 213–218).
- Wolf, D., Ruschke, T., Hochberger, C. (2018). Amidar project: lessons learned in 15 years of researching adaptive processors. In *International symposium on reconfigurable communication-centric systems-on-Chip*.
- Wolf, D., Ruschke, T., Hochberger, C., Engel, A., Koch, A. (2019). Ultrasynth: integration of a CGRA into a control engineering environment. In C. Hochberger, B. Nelson, A. Koch, R.F. Woods, P.C. Diniz (Eds.) *Applied reconfigurable computing—15th international symposium, ARC 2019, Darmstadt, Germany, April 9–11, 2019, Proceedings, Lecture Notes in Computer Science*, (Vol. 11444 pp. 247–261). Springer. https://doi.org/10.1007/978-3-030-17227-5_18.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Dennis Leander Wolf has been a research associate at TU Darmstadt from 2015 to 2021. His main research interest is reconfigurable computing, especially Coarse Grained Reconfigurable Architectures. Wolf is going to hand in his PhD thesis in 2021 that focuses on two aspects: research on CGRA microarchitectures and design space exploration of individual CGRA compositions for a specific application domain.



Tajas Ruschke has been a research associate at Technische Universität Darmstadt until 2020. His research interests include adaptive processors, hardware accelerators and Coarse Grained Reconfigurable Architectures (CGRAs). Ruschke is currently finishing his dissertation on scheduling algorithms for CGRAs.



Andreas Engel has been a postdoctoral researcher at Technische Universität Darmstadt until 2018. His research interests include microcontroller- and FPGA-based wireless sensor networks, energy-management concepts for long-term WSN operation, and Coarse Grained Reconfigurable Architecture-based hardware accelerators. Engel received a PhD in engineering from TU Darmstadt.



Christian Hochberger received his diploma and PhD in computer science in 1992 and 1998. After joining University of Rostock as a senior engineer (similar to lecturer), in 2003 he was appointed associate Professor at TU Dresden in the computer science department. 2012 he changed to TU Darmstadt as a full professor in the EE department. He has worked with FPGAs since 1988 and reconfigurable systems are still his main research focus.

Particularly, CGRAs form a major part of his research.