SO(DA)^2: End-to-end Generation of Specialized Reconfigurable Architectures

Antonino Tumeo[1]
Pacific Northwest National Laboratory, Richland, WA, USA

Nicolas Bohm Agostini
Pacific Northwest National Laboratory, Atlanta, GA, USA
Northeastern University, Boston, MA, USA

Serena Curzel
Pacific Northwest National Laboratory, Richland, WA, USA
Politecnico di Milano, Italy

Ankur Limaye
Pacific Northwest National Laboratory, Richland, WA, USA

Cheng Tan[2]
Microsoft, Seattle, WA, USA

Vinay Amatya
Pacific Northwest National Laboratory, Richland, WA, USA

Marco Minutoli
Pacific Northwest National Laboratory, Richland, WA, USA

Vito Giovanni Castellana
Pacific Northwest National Laboratory, Richland, WA, USA

Ang Li
Pacific Northwest National Laboratory, Richland, WA, USA

Joseph Manzano
Pacific Northwest National Laboratory, Richland, WA, USA

Abstract

Modern data analysis applications are complex workflows composed of algorithms with diverse behaviors. They may include digital signal processing, data filtering, reduction, compression, graph algorithms, and machine learning. Their performance is highly dependent on the volume, the velocity, and the structure of the data. They are used in many different domains (from small, embedded devices, to large-scale, high-performance computing systems) but in all cases they need to provide answers with very low latency to enable real-time decision making and autonomy. Coarse-grained reconfigurable arrays (CGRAs), i.e., architectures composed of functional units able to perform complex operations interconnected through a network-on-chip and configure the datapath to map complex kernels, are a promising platform to accelerate these applications thanks to their adaptability. They provide higher flexibility than application-specific integrated circuits (ASICs) while offering increased energy efficiency and faster reconfiguration speed with respect to field-programmable gate arrays (FPGAs). However, designing and specializing CGRAs requires significant efforts. The inherent flexibility of these devices makes the application mapping process equally important to the hardware design generation. To obtain efficient systems, approaches that simultaneously considers software and hardware optimizations are necessary. In this paper, we discuss the Software Defined Architectures for Data Analytics (SO(DA)^2) toolchain, an end-to-end hardware/software codesign framework to generate custom reconfigurable architectures for data analytics applications. (SO(DA)^2) is composed of a high-level compiler (SODA-OPT) and a hardware generator (OpenCGRA) and can automatically explore and generate optimal CGRA designs starting from high-level programming frameworks. SO(DA)^2 considers partial dynamic reconfiguration as key element of the system design. We discuss the various elements of the framework and demonstrate the flow on the case study of a partial dynamic reconfigurable CGRA design for data streaming applications.

2012 ACM Subject Classification Computer systems organization → Reconfigurable computing

Keywords and phrases Reconfigurable architectures, data analytics

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.1

[1] Corresponding Author
[2] with PNNL when this work was performed

13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin; Article No. 1; pp. 1:1–1:15

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Category Invited Talk

Supplementary Material *Software (Source Code)*: `https://gitlab.pnnl.gov/sodalite/soda-opt`
Software (Source Code): `https://github.com/pnnl/OpenCGRA`

Acknowledgements The research described in this paper is part of the Data-Model Convergence (DMC) Initiative at Pacific Northwest National Laboratory. It was conducted under the Laboratory Directed Research and Development Program at PNNL, a multiprogram national laboratory operated by Battelle for the U.S. Department of Energy.

1 Introduction

Many emerging applications for several areas employ complex workflows that include a substantial data analysis component. For example, scientific instruments such as particle accelerators, electron microscopes, and protein sequencers capture large amounts of experimental data in bursts [2] that cannot be stored locally. The significant information needs to be filtered, prepared, and often compressed before being sent to a large-scale high-performance computing systems for further processing. The in-situ data analysis element should also be able to steer and control the instruments to perform the next set of experiments. Smart sensor networks for various environmental monitoring applications need power efficient ways to acquire and select relevant data that is then transmitted on potentially slow or unstable links. Autonomous systems need fast data processing to enable the low latency reasoning required to adapt and react to changes in the environment.

In all these situations, high volumes of multi-modal, heterogeneous, data, typically captured from a variety of sensors, are processed through a sequence of kernels that may expose significantly different, and often contrasting, requirements. These kernels include digital signal processing, graph algorithms, machine learning, and more. Furthermore, the volume of data streamed from the sensors and from one kernel to the others may be highly variable depending on the situation, leading to rapidly changing throughput of the processing pipelines.

The current trends in computer architecture highlight that only by leveraging domain specialization it is possible to reach the levels of hardware efficiency (power, performance, and area) required to process exponentially growing volumes and velocity of data.

Coarse-grained reconfigurable arrays (CGRAs), loosely defined as sets of functional units (FUs) and memories interconnected through a network-on-chip (NoC) that are dynamically configured to accelerate different computational patterns, represent a promising platform for these modern data analytics workflows [17–19, 22]. A compiler maps application kernels on a CGRA and determines how data will flow through the FUs and memories. Differently from a system composed of a multitude of fixed application-specific accelerators, CGRAs can modify their configuration to adapt to the requirements of different algorithms, still resulting power efficient while providing significant gains in area efficiency through resource reuse. Their refined communication networks enable efficient data transfer from one FU to the other, allowing the definition of complex data processing pipelines with high throughput. Additionally, they can potentially adapt to new algorithm and processing pipelines. CGRAs are also more power efficiency and faster to reconfigure than fine-grained configurable devices (such as field-programmable gate arrays – FPGAs).

However, designing specialized systems based on CGRA devices and mapping software onto them are not trivial tasks. First, the entire toolchain needs to explore simultaneously hardware and software optimizations to effectively leverage the dynamic reconfiguration capabilities of the hardware substrate. Second, the actual process of designing and implementing the

hardware is complex and time consuming. A single general CGRA design may not even be sufficient to address critical use cases (e.g., edge devices, security systems) that may have very tight constraints and requirements, although only on a limited set of kernels. Thus, there is a demand for automated and integrated hardware/software codesign tools able to perform end-to-end optimization and generation of reconfigurable architectures. These tools also need to consider partial dynamic reconfiguration as critical element of the flow, especially with data streaming applications where multiple processing kernels and complex analysis pipelines may be active at the same time on inputs with highly variable characteristics.

To address the aforementioned issues, we developed Software Defined Architectures for Data Analytics – SO(DA)^2 framework [4], a modular compiler-based toolchain for the generation of custom reconfigurable architectures.

SO(DA)^2 integrates two open-source tools, SODA-OPT[1] and OpenCGRA[2], in a design flow that can automatically generate specialized CGRAs starting from a data analysis application written in a high-level software framework. SODA-OPT is a high-level optimizer that interfaces with data science programming frameworks, identifies sequences of kernels suitable for acceleration, and prepares them for offload onto the hardware. OpenCGRA is a framework for generating CGRAs, including automatic modeling, testing, evaluation, mapping, and a design space exploration (DSE) engine that allows to simultaneously optimize software and hardware parameters. This paper describes in detail the components of SO(DA)^2 and some of the key results obtained by generating and optimizing architectures with the framework itself. In particular, we demonstrate SO(DA)^2 capabilities by generating partial dynamic reconfigurable architectures for data streaming applications. The resulting specialized CGRAs designs can dynamically rebalance a pipeline of data-dependent processing kernels, maximizing the throughput (up to 2 times) and reducing the latency with respect to architectures where resources are statically partitioned among the kernels.

The paper proceeds as follows. Section 2 overviews the entire framework, discussing its key elements, including the high- and low-level compiler toolchain, the CGRA architecture templates, the design exploration engine, and the partial dynamic reconfiguration capabilities. Section 3 presents our case study. Section 4 discusses alternative generation frameworks for CGRAs. Section 5 presents possible future research and development opportunities. Finally, Section 6 concludes the paper.

2 Framework Overview

Figure 1 overviews the general concepts behind the SO(DA)^2 framework. Focus of the framework is to efficiently support data intensive applications, characterized by contrasting behaviors, by leveraging reconfigurable architectures. The framework interfaces with high-level programming frameworks through the multi-level intermediate representation (MLIR) [15] infrastructure. Our toolchain also supports Clang as a frontend, thus enabling mapping of conventional C applications onto a reconfigurable hardware substrate. One key part for data dependent workloads is the need to leverage data statistics and data-oriented optimization. A compilation flow provides opportunities to leverage dynamic information beside static analysis to enable dynamic adaptation.

The framework implements a design space exploration and synthesis (DSES) engine to perform mapping and generation of the configurations for the target architectures. The objective is identifying specific parallel patterns and explore trade-offs among multiple optim-

[1] `https://gitlab.pnnl.gov/sodalite/soda-opt`
[2] `https://github.com/pnnl/OpenCGRA`

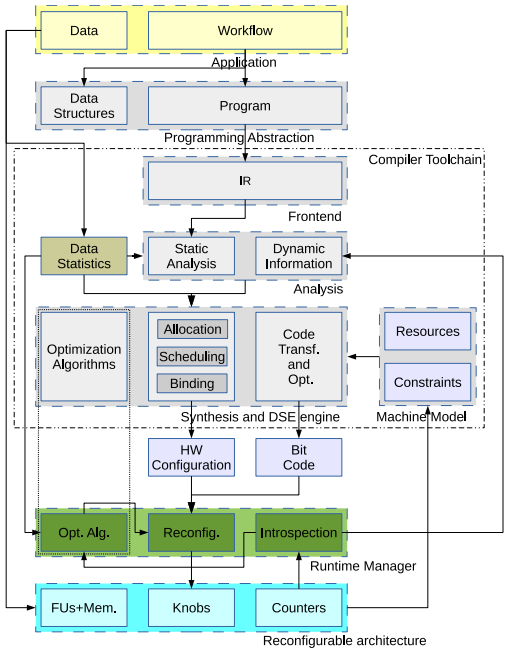


Figure 1 SODA high-level overview.

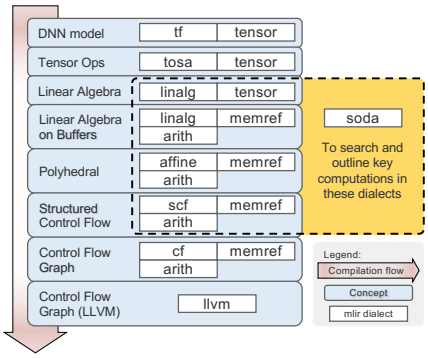


Figure 2 MLIR lowerings and dialects.

ization objectives (e.g., critical loop optimizations to exploit the spatial parallelism offered by CGRAs). The framework also considers the generation of a specialized reconfigurable architecture, leveraging a resource library with parametrized architectural templates that allows meeting specific design constraints.

Our framework considers partial dynamic reconfiguration as a key dimension for the compilation and hardware generation flow. Data dependent and data streaming applications are typically partitioned in kernels that execute for different phases of the applications and are mapped on a subset of resources. These can be statically partitioned or dynamically allocated depending on runtime metrics of the application. A runtime manager, interfacing with hardware knobs and monitoring hardware counters, allows triggering reconfiguration exploiting online optimization algorithms.

2.1 High-Level Compiler Frontend

Our infrastructure can accept input descriptions from high-level ML and domain-specific frameworks describing data analysis workflows, translated by the frontend into a high-level intermediate representation (IR). The frontend performs hardware/software partitioning and architecture-independent optimizations on the high-level IR; subsequently, it generates a low-level IR (LLVM IR) for hardware generation. SODA-OPT is the high-level compiler frontend of the SO(DA)^2 framework. Its role is to perform compiler passes to isolate and optimize code on the input program, preparing it for hardware acceleration on several different backends. To implement these functionalities, SODA-OPT leverages and extends the MLIR framework.

MLIR allows building reusable, extensible, and modular compiler infrastructure by defining *dialects*, i.e., self-contained IRs that respect MLIR's meta-IR syntax. Each dialect is designed to capture a specific abstraction, and multiple dialects can coexist in the same MLIR IR. The process to progressively refine the IR and transition between dialects is called lowering. Figure 2 shows the progressive lowering across several different MLIR dialects. The following MLIR dialects are routinely used by many tools, including SODA-OPT: `linalg` contains linear algebra operations on tensors or memory buffers, `affine` supports polyhedral transformations, `scf` provides structured control flow operations such as for and while loops, `cf` has lower-level control flow operations such as branches and switches, and the `llvm` dialect represents LLVM IR operations in the MLIR IR. Several high-level programming frameworks for various domains such as machine learning (TensorFlow, ONNX-MLIR, TORCH-MLIR), scientific computing (NPCOMP), and general-purpose languages (e.g., the FLANG frontend for Fortran) started leveraging MLIR to implement their own specific dialects, optimizations passes, and lowering methods to translate their programs into existing MLIR dialects.

SODA-OPT introduces the `soda` dialect to partition input applications into an orchestrating host program and custom hardware accelerators. SODA-OPT analysis and transformation passes ingest MLIR inputs from high-level frameworks, identify key code regions, and outline them into separate MLIR modules. Code regions that are selected for hardware acceleration can undergo a high-level optimization pipeline with progressive lowerings through different MLIR dialects (`linalg` → `affine` → `scf` → `cf` → `llvm`), or they can directly be translated into an LLVM IR without high-level optimizations.

As previously highlighted, the framework also supports inputs in C through the Clang LLVM fronted. In such a case, the code is partitioned into kernels through functions that can be mapped in various way on the underlying reconfigurable substrate. The Clang frontend also lowers to LLVM IR and, in such a case, optimizations obviously happen at the LLVM level.

2.2 CGRA Architecture Template

The SO(DA)^2 generic CGRA template is depicted in Fig. 3. It consists of modular tiles, a NoC, and a set of scratchpad (SPM) data buffers. A tile contains an FU, a configuration memory, a set of registers, and a crossbar switch; the template allows any subset of tiles to connect to the SPM banks. All the components in the template architecture are highly modular and parameterizable. For example, the flow can customize the size of the SPM, the tile count, the interconnect topology (changing the number of ports of the crossbar switch), the number of registers, and the control memory size. The type of FU is also customizable: an FU could support multiple operations, in parallel, as a sequence or as a complex pattern as shown in Fig. 3c. Fig. 3d-g show how the generic parameterizable architecture can be

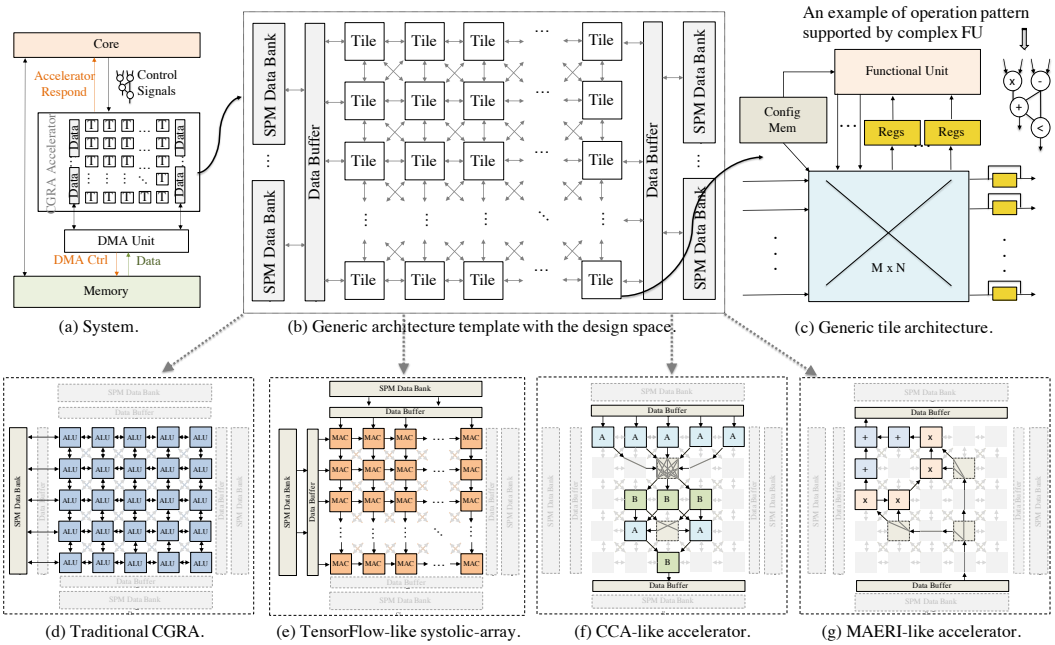


Figure 3 Generic architecture template – The generic parameterizable template provides a design space to be explored and eventually generates an optimized accelerator for the given workloads. (d)-(g) show how the template is customized into different state-of-the-art spatial reconfigurable accelerators.

customized into different state-of-the-art spatial reconfigurable accelerators, including a TensorFlow-like systolic array (e), architectures (f) similar to the configurable compute array (CCA) [6], and designs (g) like the Multiply-Accumulate Engine with Reconfigurable Interconnect (MAERI) [14].

2.3 Loop Transformations

Figure 4 details the rest of the SO(DA)^2 CGRA generation flow, which includes the compiler optimizations, the DSE process that defines an architecture, and the actual modeling, testing and evaluation of the resulting design.

Loop-level transformations are applied to expose appropriate parallelism that fits a specific CGRA architecture. In SO(DA)^2, we can apply `affine` transformations at the MLIR level within SODA-OPT, or use LLVM loop transformations on the lowered LLVM IR when inputs come from the Clang fronted.

Nested loops are flattened into a single loop to facilitate subsequent mapping and to avoid the overhead of multiple invocations of the innermost loop. Loop blocking (also known as loop tiling) constrains the size of the required data for each invocation of a kernel running on the CGRA and facilitates overlapping computation and communication with the help of double buffering. An appropriate loop blocking factor should be determined based on the memory bandwidth and the data buffer size of the accelerator. Loop unrolling significantly affects instruction-level parallelism. When the target CGRA has sufficient hardware resources (e.g., tiles, crossbars, etc.), a larger loop unrolling factor can be used; a smaller loop unrolling factor requires instead loop pipelining to recover parallelism between iterations.

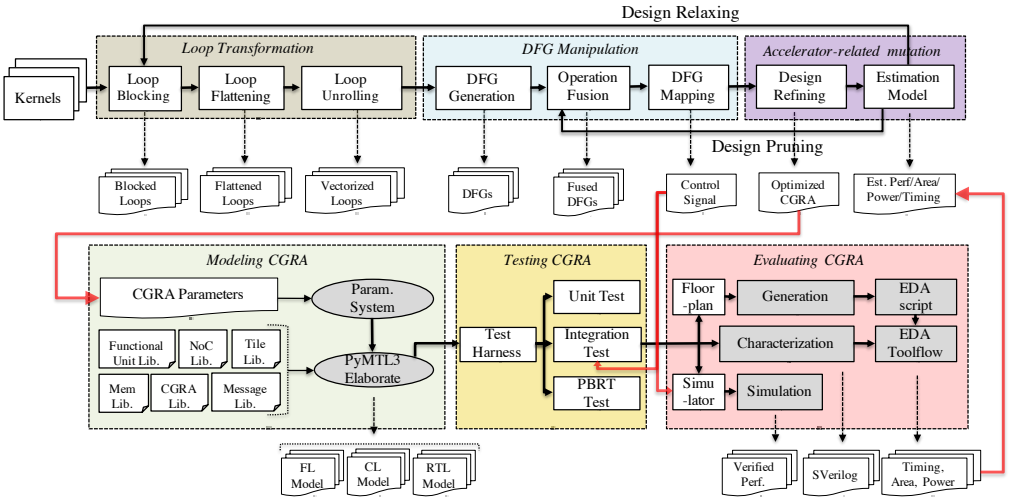


Figure 4 The rest of the flow from the generation of the kernels to the OpenCGRA generator. OpenCGRA is powered by PyMTL3 [10], PyOCN [23], Mflowgen [1], and Commercial ASIC tools.

.

2.4 Design Space Exploration

The LLVM IR produced by the frontend is optimized for execution on a CGRA architecture through a series of compiler passes performing DSE of the software (e.g., loop blocking and unrolling factors) and the hardware (e.g., amount and type of available resources) parameters [25]. The CGRA architecture itself is refined during DSE starting from the pre-designed template. In this phase, the framework extracts the data flow graph (DFG) of each kernel, fuses common arithmetic operations, and maps the resulting DFGs onto the CGRA.

We use a simulated annealing algorithm to perform DSE along all the previously listed dimensions/parameters, starting from the simplest design choice (a single tile supporting all types of operations). The heuristic then searches for the design points that meet a customizable objective (e.g., performance, area-efficiency, power-efficiency, etc.).

The DSE is supported by estimation models for the performance, power, area, and timing of the resulting designs. The overall execution time is obtained after mapping the DFG, while the other metrics are computed by leveraging analytical regression models. Specifically, the models are built by synthesizing the basic components of the architecture template and collecting the corresponding statistics (e.g., area, power, and timing). The operating frequency of the target design is dominated by the component with the longest critical path.

As DSE is time-sensitive, a fast DFG mapping algorithm is needed. Our framework implements a heuristic mapping algorithm inspired by [11], where the objective is to statically schedule operations to reach the lowest possible initiation interval (II). The algorithm incrementally increases the II value until it finds a valid mapping between the DFG and the available hardware resources. Data dependency between operations is represented as data communication between FUs and routed using Dijkstra's algorithm.

2.5 CGRA Generation

SO(DA)^2 generates the target CGRA design through a generator built on top of PyMTL3 [10], following the configuration found by the DSE. The CGRA Generator [26] enables automatic modeling, testing, and evaluation of the target optimized CGRA.

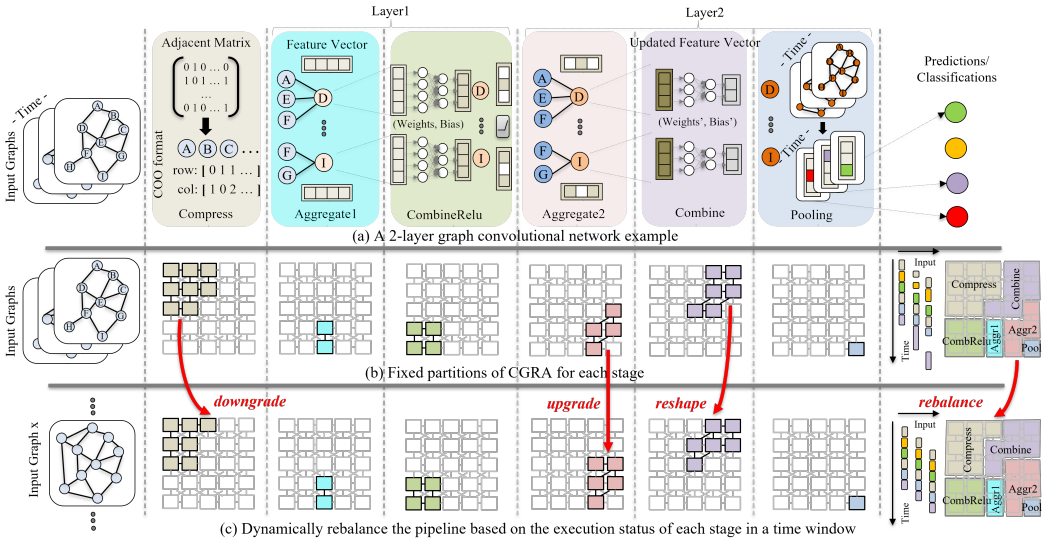


Figure 5 Example of dynamic rebalancing – (a) A 2-layer GCN inference includes 6 kernels. (b) Fixed partition for each stage targeting high throughput. (c) Dynamically reconfigure the CGRA based on the execution status of each kernel, which rebalances the pipeline and improves the overall throughput.

We support unit tests for all basic CGRA components, integration tests for the entire CGRA design, and property-based random testing (PBRT). PBRT automatically shrinks the design with minimal counterexamples that can trigger a bug, which helps users locate a design issue and eases the debugging procedure.

CGRA simulation and Verilog generation are powered by the PyMTL3 infrastructure (simulation and translation passes). The control signals generated from the mapping algorithm can serve as the input for the simulation. Moreover, with the help of a set of logic synthesis scripts, the generated synthesizable Verilog can be used to perform characterization in terms of power, area, and timing.

2.6 Partial Dynamic Reconfiguration

Data analytics applications often deals with highly variable volumes of data, arriving at variable velocities and sometimes organized in malleable data structures (e.g., graphs) with varying degrees of sparsity. This may lead to high variability in execution time of the application kernels. Current approaches for reconfigurable architectures either configure and execute one kernel at a time or statically partition resources among multiple kernels. In both cases, the latency of the application can vary from one execution of the *pipeline* of kernels to the other, limiting the overall application throughput.

As previously highlighted, the SO(DA)^2 approach considers reconfiguration as a key component of the generation flow. To address these types of applications, we developed the DynPaC [21] and the DRIPS [20] approaches. Both the designs include novel hardware and software mechanisms that enable partial dynamic reconfiguration to rebalance the execution of data-intensive, data-dependent kernels at runtime.

In both the designs, the compilation framework identifies the application kernels, outlines them, and generates potential configurations with different assignments of operations to intercommunicating tiles. These regularly shaped mappings are assigned to available tiles at

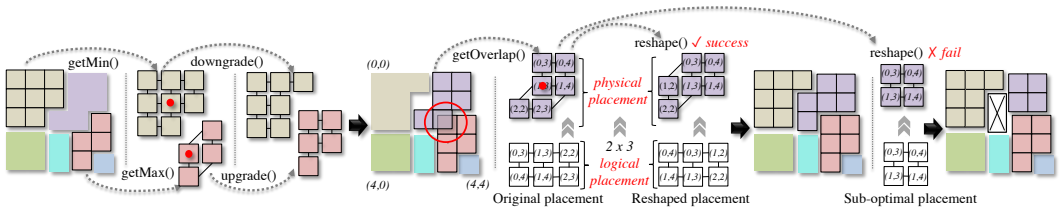


Figure 6 The *upgrade()*, *downgrade()*, and *reshape()* operations form the logical placement by enabling appropriate interconnection between neighboring tiles.

runtime. The designs leverage a king mesh interconnect topology. In this topology, each tile is interconnected with all the neighboring ones: such a rich interconnect allows remapping the kernel configurations generated by the compiler even in irregular shapes while maintaining the same communication patterns.

At runtime, the reconfiguration controller enables dynamic rebalancing by keeping track of the execution of kernels and the status of allocated resources (Figure 5). The controller identifies the fastest and slowest kernels by checking the execution delays. As shown in Figure 6, the slowest kernel is subject to the *upgrade* operation, which selects a configuration with a higher number of resources. The fastest kernel, instead, is *downgraded*, selecting a configuration with a lower number of resources. If the selected configurations do not fit in the available tiles, the controller performs *reshaping*, which adjusts the resources assigned to the kernels to fit the new layout, while respecting the communication dependencies.

3 Case Study

We evaluate the SO(DA)^2 approach by generating a CGRA design supporting partial dynamic reconfiguration. We selected two applications with data-dependent execution time: a 2-layer graph convolutional network (GCN) and a lower-upper (LU) decomposition on sparse matrices. GCNs are an emerging class of machine learning models that operate on graphs. We run inference on a pre-trained model implemented in PyTorch Geometric that predicts protein function on the ENZYMES data set (600 graphs with 2 to 126 nodes). Our streaming GCN is composed of 5 kernels (two *aggregate* operators, *Combine*, *CombRelu*, and *Pooling*). LU decomposition is a key part of solvers for systems of linear equations, a critical element of scientific simulation workflows. Our benchmark implements a streaming LU decomposition composed of 6 kernels. Table 1 describes applications, datasets, and kernels.

Table 1 Representative data-dependent applications – Each kernel of an application runs on a CGRAs with different numbers of tiles (4x4, 4x8, 6x8) and unrolling factors (1, 2, and 4). The optimal speedup (OpSp) is obtained in each case with a different regular shaped partition (OpPa); #opt represents the number of LLVM instructions in the loop body.

Application	Dataset	Kernel	4x4 CGRA, U. F. = 1			4x8 CGRA, U. F. = 2			6x8 CGRA, U. F. = 4		
			#opt	OpSp	OpPa	#opt	OpSp	OpPa	#opt	OpSp	OpPa
2-layer Graph Convolutional Network (*GCN*)	ENZYME 600 graphs 450 for training 150 for inference	*Aggregate* (×2)	27	6.8	2×4	54	13.5	2×7	99	19.8	5×5
		Combine	26	6.5	2×3	52	13	3×5	95	23.8	5×5
		CombRelu	30	7.5	3×3	60	15	3×6	111	18.5	4×5
		Pooling	16	4	2×2	32	8	2×4	55	13.6	3×5
Synthesized Lower–Upper (*LU*) Decomposition kernels	150 matrices (within the size of 100×100) selected from the University of Florida sparse matrix collection	*Init*	7	1.8	1×2	11	4	1×3	19	4.8	2×3
		Decompose	87	12.4	3×4	167	20.9	5×5	327	23.4	6×6
		Solver0	31	7.8	3×3	63	12.6	4×4	121	17.3	4×5
		Solver1	33	8.3	3×3	67	13.4	4×4	129	18.4	4×5
		Invert	65	13	4×4	127	15.9	5×5	251	19.3	6×6
		Determinant	20	3.3	2×2	39	3.9	2×2	71	3.9	2×2

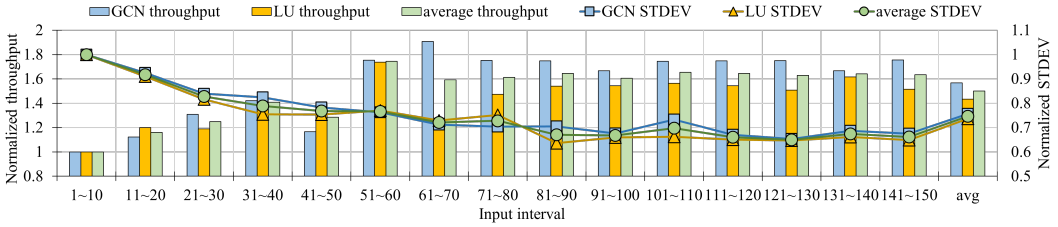


Figure 7 Normalized throughput and normalized standard deviation of different applications running on our reconfigurable designs over the baseline – The time window has a size of 10 rounds and the SPM memory is 32KB.

The table also shows that the optimal speedup, given a fixed unrolling factor, is achieved by mapping the kernel on a subset of the tiles available on the CGRA design, rather than by using the entire design. The reason is that loop-carried dependencies and the increase in routing complexity do not provide a reduction in execution time by simply adding more tiles. This indicates that sharing tiles among multiple kernels leads to better hardware utilization and improved overall throughput compared with the sequential invocation of the kernels on the CGRA, i.e., when each kernel is allocated the entire CGRA and the CGRA itself is reconfigured as kernels are progressively executed.

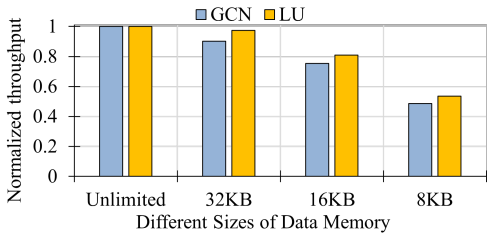
3.1 Effects of Dynamic Rebalancing

Both GCN and LU are sensitive to variations in input data. Thus, the ability of the hardware to dynamically adapt and redistribute resource during execution could provide significant benefits. We compare a partial dynamic reconfigurable design to a statically partitioned CGRA, keeping the number of tiles and the size of the SPM the same. We statically partition resources among all kernels proportionally to their overall average execution times. The same partition scheme is adopted as the initial configuration for the dynamically reconfigurable design. Fig. 7 shows the normalized throughput and standard deviation of the two applications running on the partial dynamic reconfigurable design. Dynamic adjustments are triggered after the time windows has passed. We set the time-window to 10 executions of the whole pipeline, which we found as a good intermediate point between the ability to. follow the input trends and maintaining stable throughput. A smaller time window allows quickly adapting to input variations, but if it becomes too small and bursts of data present similarities, reconfiguration may be triggered by noise. In the plot, throughput and standard deviation are calculated on the average *per time window* (i.e., 10 input samples) and normalized over the baseline (i.e., the statically partitioned solution).

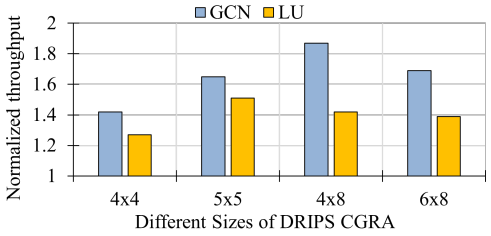
The overhead of the pipeline rebalancing process is also included in this evaluation. Dynamic reconfiguration for modified kernels terminates in less than 1000 cycles and does not stop execution. Moving the new control signals from the memory to the tiles to reconfigure them for a new or modified kernel only takes dozen of nanoseconds with a typical direct memory access (DMA) unit. Hence, rebalancing overhead is negligible with respect to the execution time of the entire pipeline of kernels (e.g. 30k to 50k cycles for the GCN).

3.2 Architectural Exploration

As previously illustrated, the SO(DA)^2 flow integrates an automated hardware generator that allows to implement and evaluate designs with different hardware parameters. We demonstrate this capability of our toolchain by exploring SPM sizes, scalability, and provide the evaluation in terms of timing, area, and power consumption.



(a) Throughput of different applications with various sizes of SPM, normalized over the throughput on a design with unlimited SPM size.



(b) Throughput of different applications running with DRIPS partial dynamic reconfiguration with various numbers of tiles, normalized over a statically partitioned design with the same size (scalability evaluation).

Figure 8 Exploration of hardware parameters: size of the SPM and number of tiles.

Figure 8a compares the throughput of the benchmark applications running on partially reconfigurable designs with different SPM sizes. Data memory in CGRA represents a critical resource, and its dimension are highly dependent from both the software and the hardware optimization processes. We show the throughput of the different options normalized over the throughput of a design with an SPM of unlimited size. We can see that if the size of the SPM decreases by 4 times, the speedup reduces only about 2 times. This happens because with less available memory the loop tiling and unrolling decisions taken during DSE will lead to smaller kernels that have access to more resources than before.

Leveraging the automated generation flow, we can also evaluate the scalability of the architecture (Figure 8b) in terms of number of tiles. We observe that the speedup decreases on a 6×8 design: this happens because kernels do not expose enough parallelism to increase the unrolling factor effectively, due to loop-carried dependencies, and are harder to schedule on larger designs. Therefore, larger CGRAs fabrics are better utilized to accelerate large applications composed of many kernels, or multiple small applications concurrently.

Finally, we evaluate the timing, area, and power consumption of a 5×5 CGRA design using the Verilog code generated by SO(DA)[2]. We use Synopsys Design Compiler, Cadence Innovus, and Synopsys PrimeTime PX with FreePDK45 to synthesize, place, route, and estimate the power consumption of the design. We use CACTI[3] to estimate the area and power of the 32KB SPM. The entire chip area is 2.07mm^2 and the operating frequency is 800MHz @ 45nm with an average power consumption of 564.8mW. The controller for partial dynamic reconfiguration only takes 16.34% of the entire area.

4 Related Work

CGRAs have emerged as promising accelerators for data analysis thanks to their ability to quickly adapt to different computational patterns while providing efficiency similar to application-specific integrated circuits. Several research frameworks were designed to facilitate the development of domain specific CGRAs.

KressArray Xplorer [9] explores the architectural design space (array size, function sets, routing channels, etc) of the KressArray architecture, composed of reconfigurable data path units. [5] provides more options (e.g., functional unit and topology) for DSE and is able

[3] `https://github.com/HewlettPackard/cacti`

to generate synthesizable Verilog. Kim et al. propose a CGRA DSE flow optimized for digital signal processing applications [12], which efficiently rearranges processing elements (PEs) by reducing the array size, and identifies interconnection topologies that minimize area and power. DSAGEN [27] explores the design space of configurable spatial accelerators starting from a generic design and trying to refine it towards an optimized version. All these approaches only perform DSE of architectural parameters, without considering software-level optimizations (e.g., loop tiling, loop unrolling, operation fusion, etc.).

RADISH [28] iteratively searches and evaluates opportunities for combining PEs. The spatial [13] compiler applies several optimizations (including loop optimizations) to efficiently map applications onto FPGAs or onto the Plasticine [19] CGRA design. However, these works do not provide an end-to-end framework, including compilation infrastructure, CGRA generation (modeling, testing, and evaluation), and integrated DSE. Furthermore, none of the aforementioned works addresses the challenges of streaming data analytics applications.

5 Development and Research Opportunities

While SO(DA)^2 infrastructure has reached a level of maturity to allow the release of its modular components in open-source, there are several opportunities to extend them for new research.

We have demonstrated that some of our designs can scale to a relatively large number of tiles [20, 21], allowing to instantiate multiple application kernels at the same time. We have also shown approaches to scale our designs to multiple nodes composed of a core and a tightly coupled CGRA [24]. However, there are further aspects to explore regarding the scalability of designs. These include the evaluation of the impact of more advanced technology nodes, larger die areas (such as those of current leading- edge accelerators, up to wafer-scale), and chiplet-based approaches.

Given the ability to generate relatively small and efficient designs, we also expect that our CGRA designs could be applicable for inclusion on logic dies of 3D-stacked memory devices, which may be only manufacturable at conservative technology nodes.

From the architectural point of view we aim at evaluating impacts and tradeoffs of adding more dynamic aspects to our statically scheduled design, leveraging the dataflow paradigm. The modular infrastructure provided by our toolchain also allows integration of new types of tiles, including solutions with new numeric formats (new standards, or custom) and highly specialized tiles generated through our state-of-the art high-level synthesis tools [7, 16].

The integration with modular, interoperable, compiler-based tools allows simultaneous exploration of software and hardware parameters. Our DSE engine mainly exploits simulated annealing, but as the space to explore grows, we plan to explore more effective heuristic search algorithms, including bioinspired heuristics such as evolutionary algorithms [3] and ant colony optimization [8], and reinforcement learning.

Finally, while our designs already support runtime partial dynamic reconfiguration, there are opportunities for monitoring other metrics beside performance (e.g., energy and real-time deadlines), and integrate different online adaptation approaches.

6 Conclusion

This paper discusses SO(DA)^2, an end-to-end framework for the generation and customization of reconfigurable architectures for data analytics.

The ability to quickly perform data analysis, including data filtering, data classification, and data reduction, are critical for many application areas (scientific computing, internet of things, finance, security, cybersecurity, and more). Additionally, efficient ways to perform data analysis are needed to enable low latency reasoning and autonomous decision processes. CGRAs, which exploit coarse grained FUs interconnected with a fast NoC, provide efficiency as well as adaptability to complex data-dependent computational patterns. SO(DA)^2 is a fully open-source toolchain composed of a compiler infrastructure that interfaces with high-level productive data science frameworks (SODA-Opt) and a CGRA generator (OpenCGRA), providing users with the capability to quickly go from algorithmic description to hardware implementation. The combination of the tools allows performing DSE and building specialized CGRAs for the applications of interests. We further show how our toolchain considers partial dynamic reconfiguration as a key part of the hardware/software optimization process, demonstrating its applicability to perform runtime rebalancing of complex pipelines of data streaming kernels.

References

1 Mflowgen. URL: `https://github.com/cornell-brg/mflowgen`.

2 E. Bethel and eds. Report of the doe workshop on management, analysis, and visualization of experimental and observational data – the convergence of data and computing. Technical report, Lawrence Berkeley National Laboratory, 2016.

3 Marco Branca, Lorenzo Camerini, Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Evolutionary algorithms for the mapping of pipelined applications onto heterogeneous embedded systems. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 1435–1442. ACM, 2009.

4 Vito Giovanni Castellana, Marco Minutoli, Antonino Tumeo, Marco Lattuada, Pietro Fezzardi, and Fabrizio Ferrandi. Software defined architectures for data analytics. In Toshiyuki Shibuya, editor, *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*, pages 711–718. ACM, 2019.

5 Anupam Chattopadhyay, Xiaolin Chen, Harold Ishebabi, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. High-level modelling and exploration of coarse-grained re-configurable architectures. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1334–1339, 2008.

6 Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *37th international symposium on microarchitecture (MICRO-37'04)*, pages 30–40. IEEE, 2004.

7 Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *DAC' 21: 58th ACM/IEEE Design Automation Conference*, pages 1327–1330, 2021.

8 Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 29(6):911–924, 2010.

9 Reiner Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. KressArray Xplorer: A new CAD environment to optimize reconfigurable datapath array architectures. In *Proceedings 2000. Design Automation Conference.(IEEE Cat. No. 00CH37106)*, pages 163–168. IEEE, 2000.

10 Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. PyMTL3: a Python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 40(4):58–66, 2020.

11 Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

12 Yoonjin Kim, Rabi N Mahapatra, and Kiyoung Choi. Design space exploration for efficient resource utilization in coarse-grained reconfigurable architecture. *IEEE transactions on very large scale integration (VLSI) systems*, 18(10):1471–1482, 2009.

13 David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

14 Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

15 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

16 Marco Minutoli, Vito Giovanni Castellana, Cheng Tan, Joseph B. Manzano, Vinay Amatya, Antonino Tumeo, David Brooks, and Gu-Yeon Wei. SODA: a new synthesis infrastructure for agile hardware design of machine learning accelerators. In *ICCAD '20: IEEE/ACM International Conference On Computer Aided Design*, pages 98:1–98:7, 2020.

17 Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 370–380, 2009.

18 Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

19 Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE, 2017.

20 Cheng Tan, Nicolas Bohm Agostini, Tong Geng, Chenghao Xie, Jiajia Li, Ang Li, Kevin Barker, and Antonino Tumeo. DRIPS: Dynamic Rebalancing of Pipelined Streaming Applications on CGRAs. In *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

21 Cheng Tan, Tong Geng, Chenhao Xie, Nicolas Bohm Agostini, Jiajia Li, Ang Li, Kevin J. Barker, and Antonino Tumeo. Dynpac: Coarse-grained, dynamic, and partially reconfigurable array for streaming applications. In *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021*, pages 33–40. IEEE, 2021.

22 Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–587. IEEE, 2018.

23 Cheng Tan, Yanghui Ou, Shunning Jiang, Peitian Pan, Christopher Torng, Shady Agwa, and Christopher Batten. Pyocn: A unified framework for modeling, testing, and evaluating on-chip networks. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 437–445. IEEE, 2019.

24 Cheng Tan, Chenhao Xie, Tong Geng, Andres Marquez, Antonino Tumeo, Kevin J Barker, and Ang Li. Arena: Asynchronous reconfigurable accelerator ring to enable data-centric parallel computing. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

25 Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, et al. AURORA: Automated Refinement of Coarse-Grained Reconfigurable Accelerators. In *The 2021 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2021.

26 Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. OpenCGRA: An open-source unified framework for modeling, testing, and evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388. IEEE, 2020.

27 Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

28 Max Willsey, Vincent T Lee, Alvin Cheung, Rastislav Bodík, and Luis Ceze. Iterative search for reconfigurable accelerator blocks with a compiler in the loop. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(3):407–418, 2018.