Rewriting History: Repurposing Domain-Specific CGRAs

Jackson Woodruff
University of Edinburgh
J.C.Woodruff@sms.ed.ac.uk

Alexander Brauckmann
University of Edinburgh
alexander.brauckmann@ed.ac.uk

Sam Ainsworth
University of Edinburgh
sam.ainsworth@ed.ac.uk

Thomas Kœhler
INRIA
thomas.koehler@thok.eu

Chris Cummins
Meta AI Research
cummins@meta.com

Michael F.P. O'Boyle
University of Edinburgh
mob@inf.ed.ac.uk

arXiv:2309.09112v1 [cs.PL] 16 Sep 2023

***Abstract*—Coarse-grained reconfigurable arrays (CGRAs) are domain-specific devices promising both the flexibility of FPGAs and the performance of ASICs. However, with restricted domains comes a danger: designing chips that cannot accelerate enough current and future software to justify the hardware cost.**

We introduce *FlexC*, the first flexible CGRA compiler, which allows CGRAs to be adapted to operations they do not natively support. FlexC uses dataflow rewriting, replacing unsupported regions of code with equivalent operations that are supported by the CGRA. We use equality saturation, a technique enabling efficient exploration of a large space of rewrite rules, to effectively search through the program-space for supported programs.

We applied FlexC to over 2,000 loop kernels, compiling to four different research CGRAs and 300 generated CGRAs and demonstrate a 2.2× increase in the number of loop kernels accelerated leading to 3× speedup compared to an Arm A5 CPU on kernels that would otherwise be unsupported by the accelerator.

I. Introduction

Specialized hardware has demonstrated truly significant performance gains over general-purpose processors [1], yet despite its potential [2], [3], it faces real challenges to wider adoption [4]. The fundamental reason is that programming such accelerators is difficult [5], often requiring modification of the underlying algorithms [4]. Users are often reluctant modify their algorithms [6] raising frequency-of-use [7], [8], [9] and cost [10] as concerns.

Heterogeneous Coarse-Grained Reconfigurable Architectures (CGRAs) [11] are a class of architectures that promise to solve this problem [7]. CGRAs can achieve near-ASIC level performance [12] and provide enough flexibility to run a wider class of code [7]. Heterogeneous CGRAs use processing elements specialized to various degrees [13]. While specialization makes hardware more efficient [14], [15], [16], hardware specialization also introduces limitations on the software [17], [18], [19].

Despite aiming at flexibility, heterogeneous CGRAs are hard to use beyond the scope they were designed for. They age poorly as software evolves [20] and falls out of the scope of the narrowly designed hardware: the *domain-restriction problem*.

This problem is highlighted by existing state-of-the-art CGRA compilers such as OpenCGRA [21] which frequently fail to generate code for the specialized hardware. If code contains even a single operation that is unsupported by a particular hardware, existing techniques simply cannot accelerate it, restricting CGRAs to an overly narrow software domain. This domain-restriction poses a significant challenge and is not well understood [22]. What we need is a new approach that *automatically transforms* user programs to fit heterogeneous CGRAs, expanding the domain of supported software without user effort.

We introduce *FlexC*, the first *flexible* CGRA compiler that addresses the domain-restriction problem. FlexC uses a set of rewrite rules that translate unsupported operations into supported ones. This compilation strategy requires a non-trivial application of rewrites in an attempt to find a valid transformation to an expression the CGRA supports, leading to a large search space. To explore this space efficiently, FlexC uses a powerful technique called equality saturation [23], [24]. CGRA compilation presents a number of unique challenges to equality saturation including, crucially, transformation encoding and cost modelling. Overcoming these challenges enables us to efficient explore large spaces.

In summary, we contribute:

- FlexC, the first *flexible* CGRA toolchain designed to support operationally-specialized CGRAs, increasing the number of loops that can run on a particular CGRA by a factor of 2.2×;
- a compiler designed to integrate domain-specific rewrite rules, and four sets of rewrite rules demonstrating the effective translation of code to run on CGRAs designed for different domains;
- the first large-scale benchmark suite for CGRA compilers, with more than 2,000 loops from five different projects[1];
- an evaluation of these tools, demonstrating the importance of non-linear exploration techniques like equality saturation in finding working compilation sequences for real-world heterogeneous CGRAs.

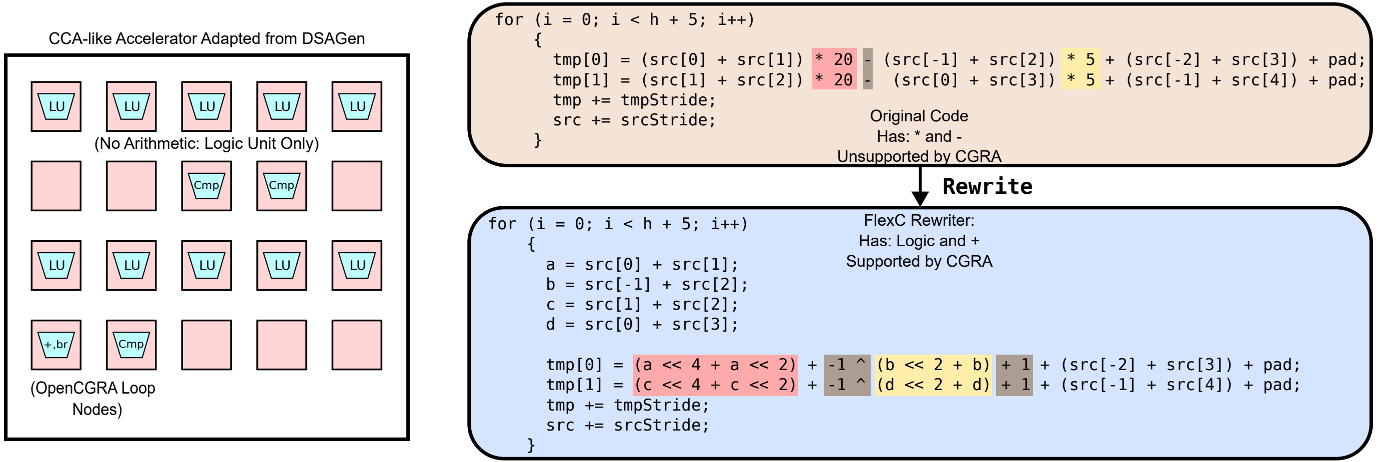


Fig. 1: An example from the FFMpeg [25] library, which is part of our benchmark suite. FlexC rewrites the loop to run within the context of the CCA-like accelerator adapted from DSAGen [26]. Equality saturation is critical in this example to enable the conversion of `a - b` into `a + -1 ^ b + 1`, as the rewriter must traverse the `a + (-b)` state, which is no better than `a - b`. This is an example of the cost-trap problem (Figure 2c).

II. Motivation

Heterogeneous CGRAs have the potential for better power efficiency and lower area utilization than their homogeneous counterparts [14], [27]. However, introducing this heterogeneity introduces significant compilation challenges.

A. The Software Domain-Restriction Problem

The cost of the specialized hardware has to be justified by enough use [7] and demand [10]. To illustrate this problem, consider the loop shown in figure 1 from the FFMpeg library. We wish to compile this code to the CCA-like accelerator adapted from DSAGen [26] shown on the left of the figure. Unfortunately, the loop contains multiplication and subtraction operators which are not supported by the CGRA. Currently, no compiler technique is able to generate code that is executable on this accelerator. Our approach is able to rewrite the program into the form shown in the bottom of the figure. It no longer uses subtraction or multiplication, but instead used additions and shifts which are supported. The loop can now be executed on the CCA.

B. Limits of existing compilers

To overcome the domain-restriction problem, we need compilers to rewrite software that uses operations not natively supported by the hardware, but existing approaches fail.

a) Standard compiler flows: Compiler frameworks, such as GCC, LLVM and MLIR, use canonicalization passes to transform IRs into a predictable and efficient form. Canonicalization is implemented with a set of simple fixed rewrite rules that are applied greedily. In heterogeneous CGRAs, canonicalizing, however, does not solve the domain-restriction problem, as the rewritten expressions may not be supported by the target hardware.

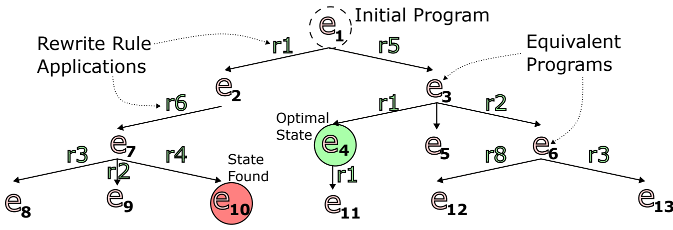
[1]To be released upon publication

b) The Limits of Greedy Dataflow Rewriting: For successful rewriting, we need to add new rewrites that allow translation to supported ones. However, we have to determine the order of, and whether to apply rewrites without searching a combinatorially large number of options. Greedy rewriting is an efficient approach but Figure 2 highlights three problems that can cause greedy rewriters to get stuck. In Figure 2(a), greedily applying the first available rules, $r1, r6, r3$ to expression e_1 leads to the resulting expression e_{10} which is less performant than the optimal expression e_4. In Figure 2(b), greedily applying the first available ruler leads to a cycle between e_1 to e_3, never reaching the solution e_4. Finally, in Figure 2(c), the greedy rewriter gets stuck in a local minimum, e_2 due to the cost of applying further local rewrites.

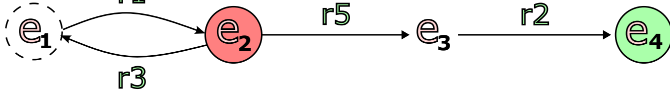
Figure 1 demonstrates these limitations in real code. To convert `a - b` into `a + -1 ^ b + 1`, the rewriter must traverse the `a + (- b)` state, which is no better than `a - b`. As there is no immediate improvement in cost, a greedy scheme would not apply such a rewrite. Equality saturation however, applies this rewrite leading to the transformed code executable on the accelerator.

C. Our approach

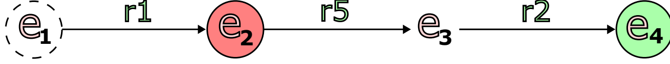
FlexC automatically adapts the software, replacing unsupported operations via dataflow rewriting. FlexC adaptively chooses between traditional greedy rewriting techniques and equality saturation [23], [28]. This overcomes challenges with traditional canonicalization and greedy techniques while retaining fast execution where possible. Equality saturation uses a graph data structure, called an e-graph, to record semantically equivalent programs while space-efficiently representing their different syntactic program variations. Rewrites are directly applied to this graph, rapidly growing the set of equivalent program variations. Rewrites are either be applied until satura-



(a) In the *exploration problem*, the expression in green is the optimal choice for this CGRA, but may never be reached in a greedy application of rewrite rules, which will reach the red state instead.



(b) In the *cycle problem*, A greedy rewriter may get stuck in a cycle due to cyclical groups of rules, preventing it from finding the optimal state.



(c) In the *cost-trap problem*, A greedy rewriter can get stuck in state e_2 as e_3 is a less valuable state.

Fig. 2: Applying rewrite rules with a greedy rewriter results in dead-ends that equality saturation avoids.

tion is reached and no rewrites can be applied any more, or until a pre-defined rewrite goal is reached [29].

Our results confirm that equality saturation enables FlexC to compile more software to the CGRA.

III. System Overview

FlexC is implemented in OpenCGRA [21], a CGRA compiler intended to target heterogeneous CGRAs. Given an input DFG, FlexC explores sequences of rewrites to eliminate the operations that are not supported by a specialized architecture from the DFG. After rewriting the DFG, FlexC uses OpenCGRA to target the hardware.

Figure 3 shows how FlexC compiles software for a CGRA. In a traditional CGRA compiler, a Data-Flow Graph (DFG) is used to generate a CGRA configuration. If the DFG does not match the target CGRA precisely, the code generation fails.

FlexC adds a rewrite system, using a set of rewrite rules dictated by the context and a cost function based on the target CGRA. After selecting the optimal graph according to the cost model (the most likely DFG to be compilable to the underlying CGRA), FlexC uses a traditional CGRA compiler to generate the final mapping.

FlexC can be applied in conjunction with any CGRA compiler — provided that appropriate rulesets using the right instructions can be supplied. We provide FlexC under a liberal license to allow this[2].

[2]Released upon publication

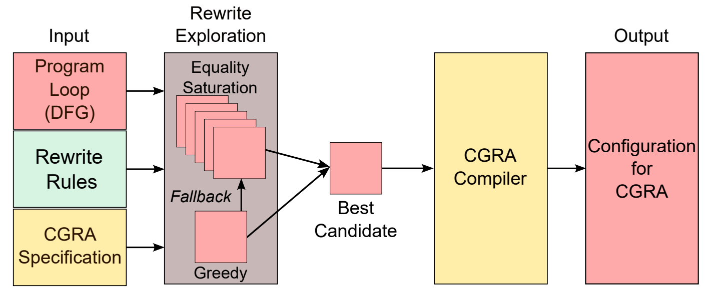


Fig. 3: FlexC system overview. A Data-Flow Graph (DFG), set of Rewrite Rules and a CGRA specification are input. FlexC first applies a hybrid-rewrite strategy and selects the most suitable candidate to pass to the CGRA compiler, which generates the configuration.

A. Graph Rewriting

FlexC translates programs to domain-specific CGRAs by generating a large set of equivalent code loops and finding a suitable match if one exists. This section formally defines our inputs: a dataflow graph representing a loop, a set of rewrite rules, and a CGRA specification and our rewriting strategy.

Definition 1: A *data flow graph* D is a finite set of nodes N corresponding to operations $op(n_1, ..., n_m)$, where op is an operation symbol and $n_i \in N$ are children operands.

D must be a *directed acyclic graph*, meaning that a function $id : N \to |N|$ should exist such that:

$$\forall n = op(n_1, ..., n_m) \in N.\ \forall i.\ id(n_i) < id(n)$$

While OpenCGRA uses Control Data Flow Graphs (CDFGs), and thus can handle branches and loops, we do not attempt to rewrite across control-flow boundaries. Instead, we break all control flow before rewriting, and restore control flow after rewriting.

Definition 2: A *rewrite rule* R is of the form $l \Rightarrow r$, where l and r are patterns. A *pattern* P is a tuple (N_P, O_P), where N_P is a data flow graph that may contain variable nodes on top of operation nodes, and $O_P \subseteq N_P$ is a list of output nodes.

R can be applied to D when l has a match (M_l, σ) in D, where M_l is a list of nodes from D matching O_l, and σ maps variables to matching nodes from D. To produce the list of nodes M_r that should replace the M_l nodes in D, the variables are substituted in r, written as $r[\sigma] = (N_r', M_r)$.

A rewrite rule must be *semantics-preserving*. This means that, $\forall (M_l, \sigma).\ M_l = M_r$ which depends on the element-wise application of a given semantic equality. The meaning of equality in this case depends on the rules provided. We will see in section IV that this may be true equality, fuzzy equality (e.g. with floating-point manipulation rules) or even weaker definitions of equality (e.g. with stochastic computing rules IV-B3).

Definition 3: In a CGRA, we have an array of processing elements, PEs (PE_i), each of which supports a particular set

of operations ($op(n_1, ..., n_m)$), $Supported_i$. We generate this set from the CGRA's specification.

Given a particular DFG D, with nodes N, there may be some subset of nodes $Unsupported(N)$ that have operations without hardware support *anywhere* on the CGRA. We wish to find a sequences of rewrite rules that we can apply to the DFG to produce D' with nodes N' such that $Unsupported(N') = \{\}$, as otherwise it will be impossible to schedule that particular code onto the CGRA. We thus define the set of operations a particular full CGRA can support as:

$$ops = \bigcup_i Supported_i$$

1) Rewriting Goal: The compiler takes a dataflow graph (DFG) D as input. Numerous existing techniques attempt to find a valid mapping [30], but in heterogeneous CGRAs, the operations in the DFG may not be in the supported set for *any* node.

The goal of a rewriting algorithm $A(D, Rs, ops)$ is to return D', obtained by rewriting D using the set of rules Rs, such that D' only uses operation symbols from ops.

We further define a cost function $C(D, ops)$ to minimize:

$$\sum_{op(...) \in N} 1 \text{ if } op \in ops \text{ else } 10^6$$

This incentivizes smaller programs by giving a cost of 1 to available operations, while giving a huge penalty to unavailable operations by giving them a cost of 10^6. Our CGRA specification and cost function aim to eliminate unavailable operations to successfully map the program onto the CGRA, without trying to precisely model the execution performance.

With the assumption that $|N| < 10^6$, rewriting successfully eliminates all unavailable operations if $C(D', ops) < 10^6$, and fails to do so if $C(D', ops) \geq 10^6$.

2) Greedy Rewriting: Listing 1 shows our greedy rewriter. Greedy rewriting is the most straightforward rewriting approach; it runs quickly but often gets stuck in local minima.

On each greedy iteration, we iterate over every rewrite rule to find matches (lines 6 to 8). If applying a rewrite for a given match leads to a cost reduction, we proceed with the rewritten program and forget about the previous program (lines 9 to 11). The local_minima variable keeps track of whether a fixed point was reached, which is the termination condition (line 3).

3) Equality Saturation: Listing 2 shows our algorithm for rewriting via equality saturation. Equality saturation [23] is a more sophisticated rewriting approach; it avoids getting stuck in local minima but can be slow to execute. We leverage both the state-of-the-art Rust `egg` library [24] and existing work extending equality saturation to graph rewriting [31].

First, we initialize an e-graph data structure that compactly represents a space of equivalent programs by sharing equivalent sub-terms as much as possible (line 2). Then, we run the explorative phase of equality saturation using our set of rewrite rules, iteratively exploring possible rewrites in a breadth-first manner and growing the e-graph (line 3).

Listing 1: Greedy rewriting algorithm

```
1  def greedy(d, rs, ops):
2    local_minima = false
3    while not local_minima:
4      local_minima = true
5
6      for r in rs:
7        matches = find_matches(d, r)
8        for m in matches:
9            d2 = apply_match(d, m)
10           if C(d2, ops) < C(d, ops):
11             d = d2
12             local_minima = false
13             break
14
15   return d
```

Listing 2: Equality saturation algorithm

```
1  def eqsat(d, rs, ops):
2    egraph = initialize_egraph(d)
3    egg_exploration(egraph, rs)
4    return egg_lp_extraction(egraph,
5                             cost_for_egg(ops))
6
7  def egg_exploration(eg, rs):
8      ...
9      while not saturation_or_timeout:
10         matches = []
11         for r in rs:
12             matches += find_matches(eg, r)
13         for m in matches:
14             apply_match(eg, m)
15     ...
```

As visible in line 9, the explorative phase terminates when all possible rewrites have been explored (a fixed point, called saturation, is reached), or when another stopping criteria is reached (e.g. a timeout). On each explorative iteration, all rewrite-rule matches are collected (line 12) and applied in a non-destructive way, adding new equalities into the e-graph (line 14).

Finally, we extract the best program from the e-graph according to our cost function using Linear Programming (line 4).

4) Hybrid Rewriting: FlexC uses hybrid rewriting (listing 3), which takes the best from both strategies. In hybrid rewriting, we first apply a fast greedy rewriter. If the greedy rewriter does not find a suitable candidate, FlexC falls back to the more

Listing 3: Hybrid rewriting algorithm

```
1  def hybrid(d, rs, ops):
2      d2 = greedy(d, rs, ops)
3      if cost(d2, ops) < 10^6: return d2
4      else:
   eqsat(d, rs, ops)
```

expensive, but more likely to succeed, equality saturation.

IV. Rewrite Rules

FlexC is a platform that can target any domain-specific CGRA, and can integrate domain-specific rules to work alongside traditional rules. Equality-saturation enables this flexibility by using the same rule exploration algorithms regardless of changes in the ruleset. We explore several different rulesets: some rules are always correct, while other rulesets may only be useful in certain domains, such as the stochastic-computing rewrite rules (section IV-B3).

In a traditional application of rewrite rules, compilers look to perform strength reduction [32], by replacing more complex rules with simpler rules — this is typically achieved by canonicalizing towards the simplest method of representing an expression. In a traditional compiler, a rule is typically formatted as:

$$\text{Complex Operation} \rightarrow \text{Simpler Operation}$$

A typical rewrite system produces a series of independent rewrites,

$$e_1 \underset{\text{rule}1}{\Longrightarrow} \cdots \underset{\text{rule}N-1}{\Longrightarrow} e_N$$

to produce the best suited expression. The rules are written in such a way that they chain together, as they are in existing compilers. We stop rewriting when no more rules are applicable.

However, when compiling for a CGRA, replacing simpler operations with *more complex* ones can be beneficial if they enable an entire region of code to be run on faster, fixed-function hardware.

As a result, for some sequence of rules

$$e_1 \underset{\text{rule}1}{\Longrightarrow} \cdots \underset{\text{rule}i-1}{\Longrightarrow} e_i \underset{\text{rule}i}{\Longrightarrow} \cdots \underset{\text{rule}N-1}{\Longrightarrow} e_N$$

some intermediate e_i may be the best choice of expression, and further, rule application can occur bidirectionally. Rather than strength reduction, which implies a linear sequence of operations that become strictly simpler, the process for compiling for a CGRA is instead *rewrite exploration*.

A. Core Integer Rules

We use a set of strength-reduction and canonicalization rules representative of those in a typical compiler. An example is:

$$x * -1 \Rightarrow -x$$

On the left-hand side of this rule, we require a multiplication operation, and on the right-hand side, we require a negation operation. For most compilers, the right-hand side is (almost) always the better choice, so most rewriters only apply these rules forwards

FlexC applies this rule in both directions, as some CGRAs may have multiplication-supporting PEs and other CGRAs may have negation-supporting PEs. We refer to this universally applicable ruleset as the *integer ruleset*. Some examples are shown in Table I.

```
  x - y   <=>   x + (-y)
 x >> y   <=>   x / (1 << y)
x and y   <=>   not ((not x) or (not y))
```

TABLE I: Some example rewrite rules that can be used to change the operations an expression requires.

```
   x * y   <=>   x / (1.0 / y)
-1.0 * x   <=>   -x
```

TABLE II: Rewrite rules enabled by reducing requirements on floating-point equality.

```
x AND y   =>   x * y
 x OR y   =>   (x + y) > 0
x XOR y   =>   x != y
```

TABLE III: Rewrite rules under the assumption that binary logical operators are boolean operators.

```
x * y   =>   x AND y
```

TABLE IV: Example rewrite rule for stochastic computing

B. Domain-Specific Rules

The core integer ruleset represents a baseline of rules widely applicable to all CGRAs. However, the constrained nature of heterogeneous CGRAs, which may feature highly specialized operations, means that domain-specific relaxations of correctness typically result in better targetability. FlexC can use custom rewrite rules as input, tailored for a given accelerator.

1) Floating-Point Rules: Floating-point rewrite rules are rarely bit-for-bit correct. Compilers typically use flags to allow for different levels of correctness guarantees, enabling floating-point transformations only when the programmer is willing to forgo accuracy.

When compiling floating-point operations to CGRAs, FlexC uses these rules by default (they can be turned off). This enables more rewrites at the cost of losing bit-correctness. An example of rules enabled by this assumption are shown in Table II.

2) Boolean Logical Operations: Logical operations such as `AND` (&) and `OR` (—) take two different meanings: they specify bitwise operations on entire words at a time, and they specify boolean operators (where any non-zero result is `true`). With a compiler flag provided by a programmer to indicate these are equivalent, we can add more rewrite rules.

For example, as boolean operations, `AND` can be rewritten using multiplication nodes, increasing the space of programs that a CGRA without logical operator support can be used for. We supply a set of rewrite operations that assume logical operations are equivalent to boolean operations. Some examples of rewrite rules in this set are shown in Table III.

3) Stochastic Computing: Stochastic computing is a computing paradigm aimed at achieving better energy efficiency than traditional computing by trading off accuracy [33]. In particular, stochastic computing allows multipliers to be replaced by logical `and` operators, and `add` operations to be replaced by muxes [34], in contexts where the absolute result is not needed, and this allows the use of simpler accelerators. Table IV shows an example ruleset.

Domain	Project	Samples
Compression	Bzip2 [35]	13
Multimedia	FFmpeg [25]	1852
	FreeImage [36]	223
Scientific Computing	DarkNet [37]	77
	LivermoreC [38]	26
Sum		2188

TABLE V: Quantities of unique loops in benchmark suite.

V. Experimental Setup

We implement FlexC above OpenCGRA, which is written in C++. We use the `egg` Rust library [24] to implement our rewriters. For equality saturation, we use an iteration limit of 10 with a node limit of 100,000 to prevent the e-graphs from growing too large. FlexC is integrated into the LLVM framework and is invoked using the `opt` tool using LLVM IR as input.

FlexC relies on OpenCGRA to find the loop to accelerate. OpenCGRA looks for the first loop in each provided function. We implement the architecture specification in JSON, adding a mapping from each PE to the sets of operations it will be able to support.

A. Benchmarks

We have collected a benchmark suite of 2188 real-world open-source code loops composed of projects in the multimedia, compression and simulation domains shown in table V. Typically, CGRA compilers are evaluated on benchmark suites of a few tens of loops which do not capture the wide spectrum of loops that programmers write, and are easy to hand optimize [39]. Our benchmark suite captures a wide range of loops, without the overheads of running whole programs [40]. These loops allow us to demonstrate FlexC works on a wide range of architectures and programs.

We extract loops suitable for CGRA scheduling from the projects shown in table V. Each extracted is the innermost loop, has no internal branches or function calls, and contains at least one array access. These properties ensure our benchmarks compile to many different CGRAs using a variety of compilation techniques.

We build a custom Clang-based tool that identifies loop structures and detects required type definitions. Clang is run using the build-system rules for each project. Each loop is placed into a function skeleton so it can be compiled.

B. Alternative approaches

We compare FlexC against three alternative approaches: OpenCGRA [21], the LLVM [41] Rewriter and our own Greedy Rewriter. OpenCGRA is the default scheme that simply maps operations to function units without any rewriting, with LLVM's rewriter disabled to enable a comparison to it. The LLVM rewriter employs the rewrite rules within the LLVM compiler infrastructure, which are intended to cannonicalize the program for typical CGRA architecture. The greedy rewriter is FlexC without equality saturation fallback.

C. Existing Domain-Specific Accelerators

We evaluate domain-specific architectures from three prior works. We consider one domain-specific CGRA work (REVAMP [14]), one more general domain-specific *accelerator* work (DSAGen [26]), and one stochastic computing CGRA (SC-CGRA [42]).

1) DSAGen: DSAGen [26] is a framework for generating domain-specific architectures. These architectures share many properties with CGRAs in that they expose architectural details to the compiler and present coarse-grained reconfigurable blocks. We make minor modifications[3] to the architectures shown in Figure 4(b) and 4(c) in [26] so they can be represented within OpenCGRA.

2) REVAMP: REVAMP [14], a framework for generating domain-specific CGRAs provides an example of a CGRA for heterogeneous compute optimization, with nodes for addition, subtraction, multiplication and some logic operations implemented within a 6x6 CGRA.

3) SC-CGRA: SC-CGRA [42] is a stochastic-computing-based CGRA. Typical exact multipliers are replaced with approximate multipliers, and similarly for adders within a 4x4 CGRA. We implement this in OpenCGRA, providing approximate adders/multiplers instead of exact ones[4].

VI. Results

We evaluate FlexC against traditional heterogeneous-CGRA compilers, improving the number of benchmarks that can be compiled to heterogeneous CGRAs by 2.2×, and demonstrate that despite making the computation more complex, the rewrite rules do not introduce slow-downs, showing geomean speedups of 3×.

A. Existing Domain-Specific Accelerators: Compilation Rate

We apply FlexC to four accelerators presented in section V-C, comparing to three other rewriting strategies. Figure 4 shows that FlexC increases the number of loops that these CGRAs can support by a factor of 2.2×. Figure 5 gives details split by benchmark suite for each accelerator.

1) DSAGen: Figure 4 shows that using FlexC increases the number of loops that can be supported on the CCA and Maeri architectures by a factor of 2.2× and 1.6× respectively. Maeri does particularly well on LivermoreC (figure 5), especially once equality saturation is used, because of the workload's heavy use of floating-point operations, though it is less suited to Bzip2 than CCA because of Maeri's lack of boolean arithmetic.

[3] OpenCGRA requires more routing to be present between compute elements, so the architectures we use are more flexible than in DSAGen. OpenCGRA also does not support architectural features like distribution trees, which we have omitted. We further add the nodes required by OpenCGRA to support loop pipelining (an `add` and an integer compare).

[4] The authors discuss different accuracies of adder/multiplier, but do not state the number of each used, so we use a simple assignment of one multiplier and one adder per node. We also omit node-fusing, as we use OpenCGRA to target this accelerator. The operators other than the multipliers and adders are not specified completely. For this evaluation, we assume each node has logical operations, and arithmetic operations simpler than multiplication. To enable OpenCGRA to compile some things on it's own, we add one exact adder, which is required for induction variables in almost all loops.

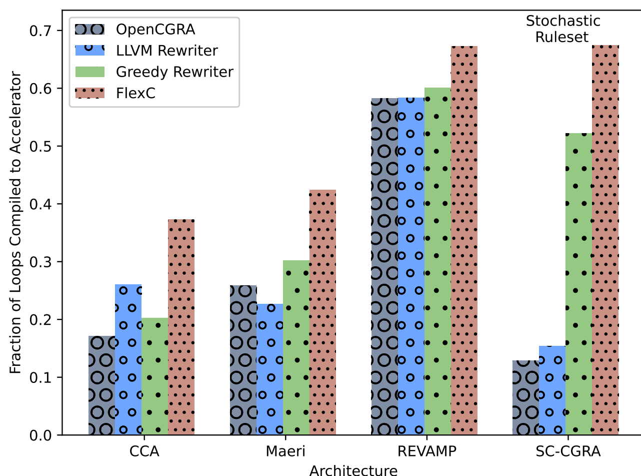


Fig. 4: We consider four different architectures, adapted from DSAGen [26] (CCA and Maeri), REVAMP [14] and SC-CGRA [42]. All architectures use the integer and floating-point rulesets, and SC-CGRA uses the stochastic ruleset. The architectures are specialized to different degrees: the more specialized architectures, CCA and Maeri, benefit from FlexC more than the more generic architecture from REVAMP.

LLVM performs well on the CCA architecture as it has a more comprehensive set of rewrite rules than have been implemented in FlexC, and on the CCA architecture, the canonicalization rules it uses are appropriately targeted. Nevertheless, FlexC outperforms it due to more comprehensive exploration of the rewrite space.

This case study, on non-CGRA architectures, reveals the generality of FlexC: while we do not claim that this comprehensively demonstrates that our rewriter can compile to different architectures (as we still rely on the OpenCGRA backend), it does demonstrate that FlexC may be applicable to more diverse computation models than CGRAs.

2) REVAMP: We implement REVAMP in the OpenCGRA framework and compile each of our benchmarks to it (fig. 4). FlexC increases the number of loops that can be supported on this CGRA by 15%, consistently across different workloads (fig. 5).

This increase is small because REVAMP's example already supports almost all the required operations for non-floating point code. We will see in other examples that FlexC becomes more important as the domain becomes more restricted.

3) SC-CGRA: Figure 4 shows FlexC increases the number of loops that can be supported by a factor of 5.2×.

This case study demonstrates FlexC is not only relevant within heterogeneous fabrics: if a homogeneous CGRA lacks operations that compilers typically assume to be available, FlexC's methods may still be necessary to generate working code. Bzip2 in particular (fig. 5) more than doubles the amount of targeted code once FlexC's equality saturation is used, compared to greedy-only, because otherwise it gets stuck in local minima and fails to explore the space enough to find a match.

B. Compilation Rate: Architectures Specialized for Loops

We demonstrate that the rewriting technique used by FlexC is applicable to many different specialized CGRAs within a varied design space.

Using 300 randomly selected loops in our benchmark suite, we first build a heterogeneous CGRA designed for that loop in particular. We run FlexC across the other loops in the benchmark suite and measure which loops can and cannot be compiled. Figure 6 shows what fraction of loops can be compiled, making a distinction between loops that are in the same suite (so are often more likely to share the same class of operation) and loops from different domains.

FlexC improves the applicability of the accelerators, both within the domain they were designed for by a factor of 2.3×, and between domains by a factor of 2.9×, demonstrating the applicability of FlexC to many different types of heterogeneous CGRA. In some cases, a typical accelerator for a loop in one benchmark will actually do better on the other workloads (e,g, for freeimage and ffmpeg). This is because freeimage and ffmpeg are highly diverse, and so an accelerator designed for one loop is less likely to match others in the same diverse benchmark.

Figure 7 shows FlexC supports CGRAs across a wide range of specializations, from very specialized CGRAs with only a few operators to complex heterogeneous CGRAs. For architectures with fewer operations, equality saturation is more important, as there are fewer paths to a valid rewrite.

1) Speedups: This section demonstrates that rewriting code in ways that at first-glance are inefficient can result in speedup by enabling accelerator utilization. Compiling to CGRA implementations typically improves performance *and* reduces power usage. We consider speedup in this evaluation. In line with other CGRA work, we consider speedup in the case that loops are executing large numbers of iterations, so one-time overheads like offloading costs for loosely coupled accelerators are ignored.

We compare two systems with similar specifications. For a CGRA system, we take architectural parameters for ADRES [43], a 6x6 CGRA which we clock at 200 MHz. We use the initialization interval to obtain performance estimates for the CGRA. To obtain a realistic CPU baseline, we execute the loops on an Arm A5 running at 500 MHz using an Analog Devices SC-589EZKit development board [44] and methodology for generating inputs from Exebench [45]. Speedups are shown in figure 8, showing a geomean performance improvement of 3×, demonstrating that FlexC's rewrite rules are not only effective in enabling targeting of CGRAs, but also in achieving speedup on them.

C. Existing Domain-Specific Accelerators: End-to-End Evaluation

We demonstrate that FlexC also performs well on well-known and computationally important kernels. To do this, we

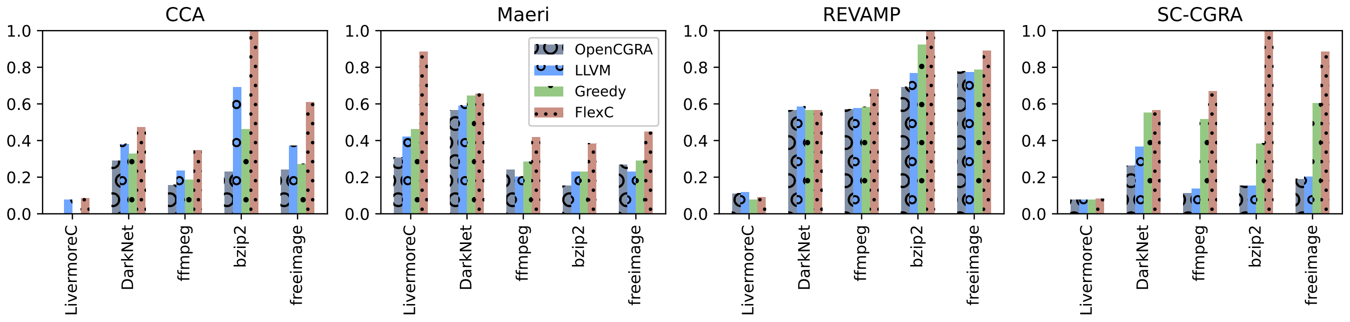


Fig. 5: Results for each accelerator pairing by benchmark suite. Equality saturation often dramatically improves coverage for particular workload-accelerator combinations (e.g. bzip2 on CCA and SC-CGRA, and LivermoreC on Maeri), where otherwise the accelerator would appear entirely unsuitable. In these cases, the accelerator has the right class of operator for the tasks required (logical operators for bzip2 and floating-point operators for LivermoreC) but the code still requires transformation to fit the individual available operations.

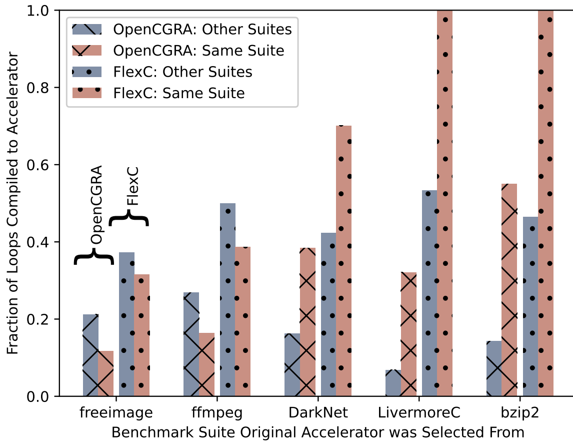


Fig. 6: Using accelerators designed for individual loops in each benchmark suite, how much code in the same suite (red) and other suites (blue) can be compiled to these accelerators. FlexC increase the compilation rate by a factor of 2.3× in the same suite and 2.9× between suites.

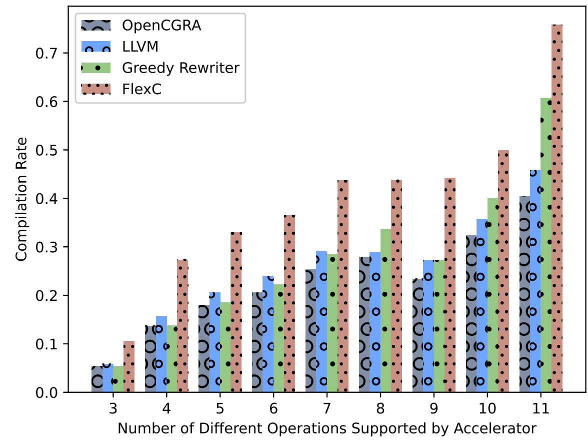


Fig. 7: How the number of different operations in a CGRA influences compilation rate. FlexC performs consistently across many levels of generality, from very specialized accelerators with few operators to much more generic accelerators with many different operators. Equality saturation is most important for more specialized architectures.

take the OpenCGRA benchmark suite [21], along with the LivermoreC benchmark suite previously explored. We use the same setup as in section VI-B1.

The results are shown in Figure 9. Compared to running on an Arm Cortex-A5, FlexC achieves a speedup on 2.0× across all applications. This compares to the LLVM rewriter, which is only able to extract 1.5× performance increase across all applications.

[5]We omitted ADPCM Encoder/Decoder as OpenCGRA is unable to compile it to any accelerators due to size, and Conv, FFT, MVT and Relu due to presence of divide operations that cannot be eliminated, as none of our case-study accelerators support divide operations.

D. Using Different Rulesets

FlexC provides a generic rewriting framework that can be applied to many different rulesets. These rulesets may be flagged by the programmer as valid for particular loops, or valid for a particular program.

We inspect four different rulesets here (covered in more detail in Section IV-A), an integer ruleset, derived of rules that may always be applied, a floating-point ruleset, derived of rules that may be applied under assumptions such as `-ffast-math`, a logical operations-as-binary operations ruleset that can be used to provide greater flexibility of rewrites involving logical operators and a stochastic computing ruleset that enables typical stochastic computing transformations. These secondary rulesets

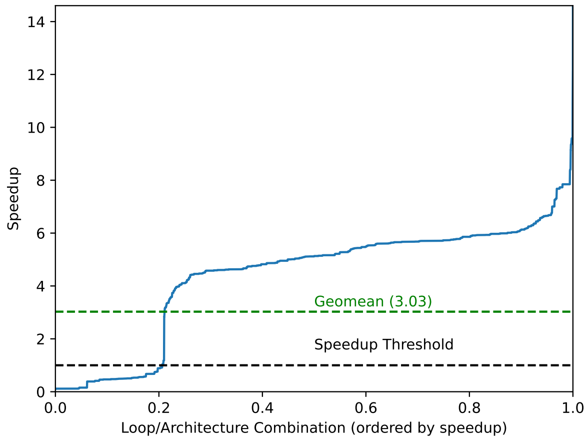


Fig. 8: Speedup achieved by rewriting applications to run on a low-power CGRA vs running on a comparable low-power CPU. We achieve 3× geomean over running on a CPU.

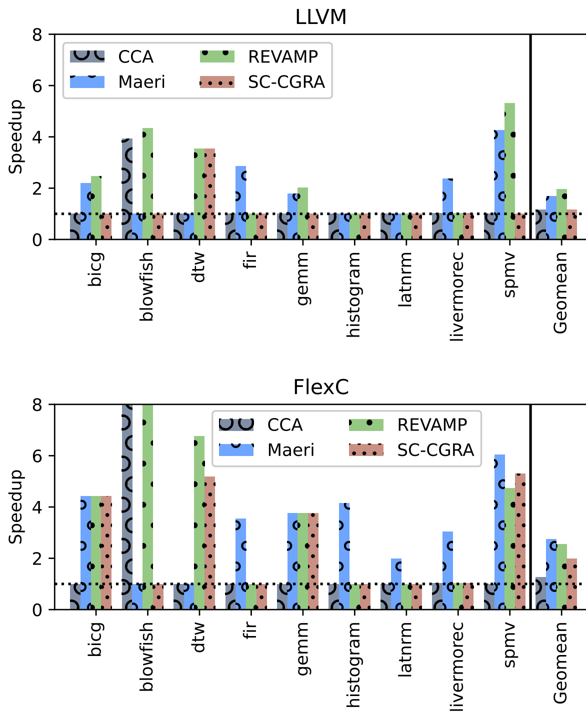


Fig. 9: Speedups using the OpenCGRA benchmark suite and the Livermore C benchmark suite, comparing various CGRA architectures to an Arm Cortex-A5. Benchmarks that were unsupported by any architecture/compiler pairs have been omitted[5]. The top figure shows the speedup achieved using the LLVM rewriter to target each CGRA, and the bottom figure shows speedup via FlexC.

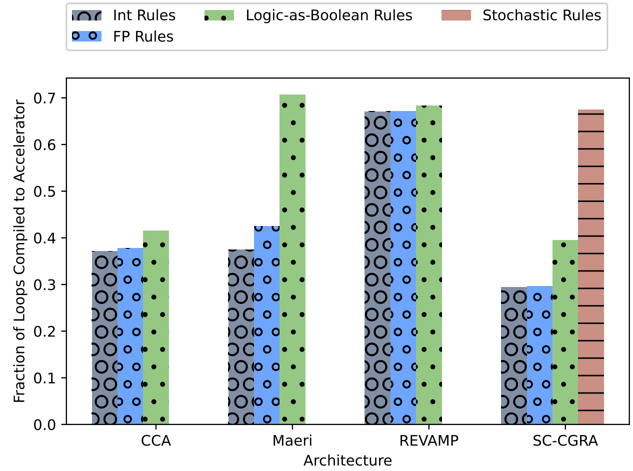


Fig. 10: Comparing how different sets of rewrite rules improve the code coverage of an accelerator. All rulesets are run with the int ruleset. The stochastic computing rules are only applied to SC-CGRA as they require specialized hardware support not available in other accelerators.

can be activated by the programmer using a flag. Figure 10 shows how these different rulesets provide different compilation performance. Rulesets are run in combination with the int ruleset as it contains many *enabling* rewrites for the specialized rewrites in the other rulesets. We can see that determining which rulesets are useful is architecture-specific. For example, Maeri benefits a lot from the logic-as-boolean ruleset, as it does not have logic operators, while CCA benefits from the stochastic rules as it does not have multipliers.

E. Most Frequently Applied Rewrite Rules

Part of the power of FlexC is that the rewrite rules that need to be applied vary by architecture. By using equality saturation, FlexC is able to use one standard set of rules for all architectures and apply the relevant rules in each case. Table VI shows the most frequently applied rules for the CCA and Maeri architectures (when compiled using the integer and floating point rulesets): two architectures with nearly disjoint sets of operators.

F. Compile Time

A challenge with Equality Saturation is in keeping the search-space manageably sized, as e-graphs can grow rapidly, causing excessive compile times and resource usage [29], [46]. We avoid these issues in FlexC by limiting the number of explorative iterations, still finding good solutions in many cases.

Figure 11 shows the time taken by FlexC to rewrite and schedule the DFG. We use a cutoff time of 300 s to avoid exploring the rewrite space fruitlessly — we can see that the rate of successful compilations drops off rapidly after 60 s, followed first by a large number of early terminations without successful scheduling (most likely due to reaching saturation,

	CCA
1	`x * 2 => x << 1`
2	`x * 4 => x << 2`
3	`x * 1 => x`
4	`-x (Floating Point) => x + 2^32 (Int)`
	Maeri
1	`x * 1 => x`
2	`x << 1 => x * 2`
3	`x << y => mul(x, load(csel(y > 32, 33, y)))`
4	`x - y => x + -y`
	REVAMP
1	`x * 1 => x`
2	`-x (Floating Point) => x + 2^32 (Int)`
3	`x  / 2 => x >> 1`
4	`x / 8 => x >> 3`
	SC-CGRA
1	`x * y => x & y`
2	`x * y => ISC(x, y)`
3	`x * 1 => x`
4	`-x (Floating Point) => x + 2^32 (Int)`

TABLE VI: The most commonly applied rules for each architecture. We omit LLVM-specific rewrites for SC-CGRA. As the CCA and Maeri provide nearly disjoint operators, they are examples of the need for rewrites to apply bidirectionally.

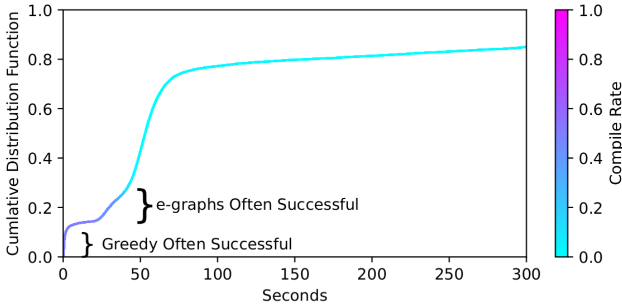


Fig. 11: Time to schedule code on a CGRA using FlexC and OpenCGRA. We cut-off rewriting at 300 s to avoid excessive exploration. After 60 s, the compilation rate is very low, so FlexC is not missing many compilations at longer timeouts.

iteration or node limit), then by stagnation in progress for infeasibly large search spaces. These compile times are fast enough that more exhaustive CGRA schedulers will be able to incorporate this strategy within the existing order of magnitude of compilation time. For example Beidas and Anderson report ILP compile times with a geomean of 60 s [47].

We can also see the effect of using a greedy rewriter as a preliminary step here. In 10% of cases, FlexC is able to rely on the greedy rewriter and find a compiling loop rapidly. We can further see that when FlexC uses equality saturation, it is more successful early on in the exploration.

VII. Related Work

A. Existing CGRAs

Research on CGGRs has been extensive [11], [30]. Older CGRAs [43], [48], [49] tend to provide a homogeneous grid of PEs with a programmable interconnect. However, the design-space of CGRAs is immense, including heterogeneous on-chip networks [50], [51], [52], [53], [54], decouplings of memory and compute [55], [26], unifying memories [56], and various techniques to specialize PEs [57], [58], [59], [60], [28], [27], [61], [62]. Toolchains to enable development [63], [64], [65], [66], [67], [68], [69], [70] and aid design [14], [71], [72] mean that is is relatively easy to design and build a domain-specific CGRA.

1) Domain-Specific CGRAs: Domain-specific CGRAs exist for neural networks [73], [74], [75], [76], scientific kernels [77], [78], [62], [16], [79], [80], approximate computing [81], [42], stencil computations [82], HPC [52], multimedia [83], [84], [85] and streaming applications [12].

2) Industrial CGRAs: Xilinx's ACAP provides a CGRA-like model of computation [86] and uses an MLIR-based toolchain [87]. Samsung have designed the SLP-URP [88] for low-power medical use-cases. Smaller companies such as Wave Computing [89] and SambaNova Systems [74] and Recore Systems [90] also involved in CGRA-design. Large-scale research projects of producing real hardware [91], [92] also use CGRAs.

B. Compiling for CGRAs

Numerous authors address compiling branches [93], [94], [95], [96], [97], [98], nested loops [99], [96], [100], scheduling of large loops [101], [102], [103], [104], [105] and irregular memory accesses [98]. Compilation time is relevant in many fields [106], [107] and has been addressed both with faster algorithms [106] and hardware acceleration [108], [109].

CGRA scheduling can be done with binary decision diagrams [47], the polyhedral model [52], [110], SAT solvers [111], [112] and ILP models [113], [114], [115], [116]. Heuristic approaches can use information from failed placements [117], [118], rewrite rules to simplify routing [119], sharing information between placement and routing phases [120] and integration of hardware features within the compiler model [121], [122]. Machine learning can automate many approaches [123], [124], [125], [105], [126].

DSLs can be used to simplify the compilation processes, enabling greater parallelism [127], [128], [129], [130], [131], [132] but do not totally eliminate compiler challenges [133]. API interfaces can eliminate compiler challenges, but also eliminate the flexibility of CGRAs [134], [135].

C. Compiling for Hardware Accelerators

Compiling for API programmable accelerators has been explored using equality saturation [136] and program synthesis [17]. Equality saturation has also been used to optimize tensor programs for tensor accelerators [137]. Similarly, externalizing rewrite rules to enable programs to run efficiently on hardware accelerators [138]. While these approaches could target CGRAs behind API interfaces, they do not support compiling to CGRAs directly — and so lose flexiblity. Idiomatic compilation [139] has been used to target closely-related spatial accelerators [140].

D. Equality Saturation

Equality saturation [23], [24] has been used for a range of tasks, including: optimization and translation validation of Java bytecode and LLVM programs [141], improving accuracy of floating point expressions [142], synthesizing structured CAD models [143], optimizing linear algebra expressions [144], tensor graph superoptimization [31], vectorization for digital signal processors [145], optimizing integer multiplication on FPGAs [146], hardware datapath optimization [147].

A proposed DSLs to Accelerators (D2A) methodology [137], [136] uses equality saturation for optimization and hardware mapping of DSLs. This paper aligns with the *flexible matching* idea from the D2A methodology, but considers mapping arbitrary C code to CGRAs to address the CGRA domain-restriction problem, and evaluates the difference between equality saturation and greedy rewriting.

While equality saturation is a powerful rewriting technique that addresses limitations of greedy rewriting, scaling to long rewrite sequences is limited as the e-graph grows quickly. The Pulsing Caviar mechanism [46] was evaluated on arithmetic expressions to balance exploration and exploitation, and compared to greedy rewriting for this purpose. Sketch-guiding [29] is another recently proposed semi-automated technique to improve scaling of equality saturation.

VIII. Conclusion

We introduce FlexC, a compiler for domain-specific CGRAs that addresses the domain-restriction problem: where CGRAs that have been designed for a particular domain are hard to apply to software outside that domain. FlexC uses equality-saturation to rewrite software from different domains so it can run on hardware not designed for it. FlexC increases the number loops that can be supported by a factor of $2.2\times$ over existing CGRA compilers and enables acceleration of loops leading to a geomean speedup of $2.1\times$.

FlexC demonstrates the potential that rewriting software to match novel hardware has: the techniques developed here are applicable to other kinds of accelerators with programmable networks. We present the first study that characterizes how different decisions surrounding heterogeneity effect the fraction of code supported by an accelerator, showing that the more specialized an accelerator is, the more important FlexC is. FlexC opens up new development possibilities by promising that even if software requirements change in a heartbeat, accelerators with a large sunk-cost can still be applied.

References

[1] A. Fuchs and D. Wentzlaff, "The accelerator wall: Limits of chip specialization," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2 2019.

[2] M. B. Taylor, "Is dark silicon useful?," ACM Press, 6 2012.

[3] "International roadmap for devices and systems 2020 update: More moore," 2020.

[4] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, pp. 48–57, 6 2020.

[5] H. Corporaal and E. de Bruin, "What is a CGRA?," *ACM/SIGDA e-newsletter*, 2023.

[6] M. Silberstein, "Accelerators in data centers: the systems perspective," *ACM SIGARCH*, 2017.

[7] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, pp. 40–50, 2017.

[8] E. Brunvand, D. Kline, and A. K. Jones, "Dark silicon considered harmful: A case for truly green computing," *International Green and Sustainable Computing Conference*, 2018.

[9] U. Gupta, M. Elgamal, G. Hills, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "ACT: Designing sustainable computer systems with an architectural carbon modeling tool," *ISCA*, 2022.

[10] M. Khazraee, I. Magaki, L. Vega Gutierrez, and M. Taylor, "ASIC clouds: Specializing the datacenter," *IEEE Micro*, pp. 1–1, 2017.

[11] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design," *ACM Computing Surveys*, vol. 52, pp. 1–39, 10 2019.

[12] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "Chordmap: Automated mapping of streaming applications onto CGRA," *Computer-aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, 2022.

[13] E. Aliagha and D. Gohringer, "Energy efficient design of coarse-grained reconfigurable architectures: Insights, trends and challenges," *FPT*, 2022.

[14] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "REVAMP: A systematic framework for heterogeneous CGRA realization," *ASPLOS*, 2022.

[15] B. Van Essen, R. Panda, A. Wood, C. Ebeling, and S. Hauck, "Energy-efficient specialization of functional units in a coarse-grained reconfigurable array," *FPGA*, 2011.

[16] Z. Ebrahimi and A. Kumar, "BioCare: An energy-efficient CGRA for bio-signfal processing at the edge," *ISCASS*, 2021.

[17] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O'Boyle, "Bind the gap: Compiling real software to hardware FFT accelerators," *PLDI*, 2022.

[18] S. Hooker, "The hardware lottery," *Communications of the ACM*, 2021.

[19] J. Woodruff and M. F. P. O'Boyle, "New regular expressions on old accelerators," *DAC 2021*, 2021.

[20] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," *Symposium on Low Power Electronics and Design*, 2013.

[21] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pp. 381–388, IEEE, 2020.

[22] M. Wijvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," *SAMOS*, 2016.

[23] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 264–276, 2009.

[24] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *POPL*, 2021.

[25] "Ffmpeg." Available from git://git.ffmpeg.org/ffmpeg.git.

[26] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: synthesizing programmable spatial accelerators," *ISCA*, 2020.

[27] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: An ultra-low-power, energy-minimal CGRA-generation framework and architecture," *ISCA*, 2021.

[28] M. Willsey, V. T. Lee, A. Cheung, R. Bodik, and L. Ceze, "Iterative search for reconfigurable accelerator blocks with a compiler in the loop," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, pp. 407–418, 3 2019.

[29] T. Koehler, P. Trinder, and M. Steuwer, "Sketch guided equality-saturation," *CoRR*, 2022.

[30] K. J. M. Martin, "Twenty years of automated methods for mapping applications on CGRA," *IDDD International Parallel and Dsitributed Processing Symposium Workshops (IPDPSW)*, 2022.

[31] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality saturation for tensor graph superoptimization," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 255–268, 2021.

[32] K. D. Cooper, L. T. Simpson, and C. A. Vick, "Operator strength reduction," *ACM Trans. Program. Lang. Syst.*, vol. 23, p. 603–625, sep 2001.

[33] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *Transactions on Embedded Computing Systems*, vol. 12, no. 2, 2013.

[34] T. J. Baker and J. P. Hayes, "The hypergeometric distribution as a more accurate model for stochastic computing," *DATE*, 2020.
[35] "bzip2." Available from git://sourceware.org/git/bzip2.git.
[36] "Freeimage." Available from http://downloads/sourceforge.net/freeimage/FreeImage3180.zip.
[37] "Darknet." Available from git://github.com/pjreddie/darknet.
[38] "Livermore loops." Available from https://netlib.org/benchmark/livermorec.
[39] "Processor benchmark limitations,"
[40] "An introduction to CPU performance benchmarks and how this applies to the home market," November 2021.
[41] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, pp. 75–86, 2004.
[42] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, "Compilable neural code generation with compiler feedback," *CoRR*, 2022.
[43] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*, pp. 61–70. Springer Berlin Heidelberg, 2003.
[44] "Analog devices SHARC+ dual-core DSP with Arm Cortex-A5: ADSP-SC582/SC583/SC584/SC589/ADSP21583/21584/21587," 2018. Available at https://www.analog.com/media/en/technical-documentation/data-sheets/ADSP-SC582_583_584_587_589_ADSP-21583_584_587.pdf.
[45] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. F. O'Boyle, "Exebench: An ML-scale dataset of executable C functions," *International Symposium on Machine Programming*, 2022.
[46] S. Kourta, A. A. Namani, F. Benbouzid-Si Tayeb, K. Hazelwood, C. Cummins, H. Leather, and R. Baghdadi, "Caviar: an e-graph based trs for automatic code optimization," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pp. 54–64, 2022.
[47] R. Beidas and J. H. Anderson, "CGRA mapping using zero-supporessed binary decision diagrams," *ASP-DAC*, 2022.
[48] J. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," IEEE Comput. Soc, 1997.
[49] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *Micro*, 2012.
[50] L. Zhou, L. Hengzhu, and J. Zhang, "Loop acceleration by cluster-based CGRA," *IEICE*, pp. 1–8, 2013.
[51] D. Wijerathne, Z. Li, T. K. Bandara, and T. Mitra, "PANORAMA: divide-and-conquer approach for mapping complex loop kernels on CGRA," *DAC*, 2022.
[52] K. T. Madhu, S. Das, N. S, S. K. Nandy, and R. Narayan, "Compiling HPC kernels for the REDEFINE CGRA," *HPCC*, 2015.
[53] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube," in *the 54th Annual Design Automation Conference 2017*, ACM Press, 6 2017.
[54] B. Adhi, C. Cortes, Y. Tan, T. Kojima, A. Podobas, and K. Sano, "The cost of flexibility: Embedded versus discrete routers in CGRAs for HPC," *International Conference on Cluster Computing*, 2022.
[55] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "CASCADE: High throughput data streaming via decoupled access-execute CGRA," *Transactions on Embedded Computing Systems*, vol. 18, no. 5s, 2019.
[56] L. Dai, Y. Wang, C. Liu, F. Li, H. Li, and X. Li, "Reexamining CGRA memory sub-system for higher memory utilization and performance," *ICCD*, 2022.
[57] Z. Zhao, W. Sheng, W. He, Z. Mao, and Z. Li, "A static-placement, dynamic-issue framework for cgra loop accelerator," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 3 2017.
[58] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: A coarse grained reconfigurable architectural template," *VLSI*, 2011.
[59] Y. Park, H. Park, and S. Mahlke, "CGRA express: Accelerationg execution using dynamic operation fusion," *CASES*, 2009.
[60] J. Melchert, K. Feng, C. Donovick, R. Daly, C. Barrett, M. Horowitz, P. Hanrahan, and P. Raina, "Automated design space exploration of CGRA processing element architectures using frequent subgraph analysis," *CoRR*, 2021.
[61] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne, "Selective flexibility: Creating domain-specific reconfigurable arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 681–694, 5 2013.
[62] W. Byun, M. Je, and J.-H. Kim, "An energy-efficient domain-specific reconfigurable array processor with heterogeneous pes for wearable brain-computer interface socs," *Transactions on Circuits and Systems*, 2022.
[63] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 7 2017.
[64] C. Tan, N. B. Agostini, J. Zhang, M. Minutoli, V. G. Castellana, C. Xie, T. Geng, A. Li, K. Barker, and A. Tumeo, "Opencgra: Democratizing coarse-grained reconfigurable arrays," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 7 2021.
[65] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Aurora: Automated refinement of coarse-grained reconfigurable accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2 2021.
[66] C. Liu, H.-C. Ng, and H. K.-H. So, "Quickdough: A rapid FPGA loop accelerator design framework using soft CGRA overlay," *FPT*, 2015.
[67] A. Podobas, K. Sano, and S. Matsuoka, "A template-based framework for exploring coarse-grained reconfigurable architectures," *ASAP*, 2020.
[68] L. Silva, M. Canesche, R. Ferreira, and J. A. Nacif, "Hpcgra - an orthogonal designed cgra generator for high performance spatial accelerators," in *XXI Simpósio em Sistemas Computacionais de Alto Desempenho*, Sociedade Brasileira de Computação, 10 2020.
[69] Y. Guo and G. Luo, "Pillars: An integrated CGRA design framework," *WOSET*, 2020.
[70] D. Wijerathne, Z. Li, M. Karunaratne, L.-S. Peh, and T. Mitra, "Morpher: An open-source integrated compilation and simulation framework for CGRA," *Workshop on Open-Source EDA Technology*, 2022.
[71] S. Döbrich and C. Hochberger, "Exploring online synthesis for CGRAs with specialized operator sets," *International Journal of Reconfigurable Computing*, 2011.
[72] J. Weng, A. Jian, J. Wang, L. Wang, Y. Wang, and T. Nowatzki, "UNIT: Unifying tensorized instruction compilation," *CGO*, 2021.
[73] T. Geng, C. Wu, C. Tan, B. Fang, A. Li, and M. Herbordt, "CQNN: a CGRA-based QNN framework," *HPEC*, 2020.
[74] "Accelerated computing with a reconfigurable dataflow architecture," 2021.
[75] H. Anwar, S. M. A. H. Jafri, S. Dytckov, M. Daneshtalab, M. Ebrahimi, and A. Hemani, "Exploring spiking neural network on coarse-grain reconfigurable architectures," *Mes*, 2014.
[76] J. Lee and J. Lee, "NP-CGRA: Extending CGRAs for efficient processing of light-weight deep neural networks," *DATE*, 2021.
[77] G. Charitopoulos, I. Papaefstathiou, and D. N. Pnevmatikatos, "Creating customized CGRAs for scientific applications," *Electronics*, vol. 10, 2021.
[78] B. W. Denkinger, M. Quiros-Peon, M. Konijnenburg, D. Atienza, and F. Catthoor, "VWR2A: A very-wide-register reconfigurable-array architecture for low-power embedded devices," *DAC*, 2022.
[79] R. Prasad, S. Das, K. J. M. Martin, and P. Coussy, "Floating point CGRA based ultra-low power DSP accelerator," vol. 93, pp. 1159–1171, 2021.
[80] L. Liu, G. Pen, and S. Wei, "Dynamic reconfigurable chips for massive MIMO detection," *Massive MIMO Detection Algorithm and VLSI Architecture*, pp. 229–306, 2019.
[81] O. Akbari, M. Kamal, A. Kusha-Afzali, M. Pedram, and M. Shafique, "PX-CGRA: Polymorphic approximate coarse-grained reconfigurable architecture," *DATE*, 2018.
[82] J. J. Tithi, F. Petrini, H. Rong, A. Valentin, and C. Ebeling, "Mapping stencils on coarse-grained reconfigurable spatial architecture," *CoRR*, 2021.
[83] Y.-H. E. Yang and V. K. Prasanna, "High-performance and compact architecture for regular expression matching on FPGA," *Transactions on Computers*, 2012.
[84] B. Mei, B. De Sutter, T. Vander Aa, M. Wouters, A. Kanstein, and S. Dupont, "Implementation of a coarse-grained reconfigurable media processor for avc decoder," *Journal of Signal Processing Systems*, vol. 51, pp. 225–243, 6 2008.

[85] H. K. Nguyen and X.-T. Tran, "Design and implementation of a coarse-grained dynamically reconfigurable multimedia accelerator," *ACM Transactions on Parallel Computing*, vol. 9, no. 3, 2022.
[86] Xilinx, "AI engine: Meeting the compute demands of next-generation applications," Accessed 2022. Available at https://www.xilinx.com/products/technology/ai-engine.html.
[87] Xilinx, "MLIR-based AIEngine toolchain," Accessesed 2022.
[88] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications," in *2012 International Conference on Field-Programmable Technology (FPT)*, IEEE, 12 2012.
[89] C. Nicol, "A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing," techreport, Wave Computing, 2017.
[90] P. M. Heysters, G. K. Rauwerda, and L. T. Smit, "A flexible, low power, high performance DSP IP core for programmable systems-on-chip," *Design and Reuse*, 2005.
[91] S. Pal, S. Feng, D.-h. Park, G. Kim, A. Amarnath, C. Yang, X. He, J. Beaumont, K. May, Y. Xiong, K. Kaszyk, J. M. Morton, J. Sun, M. F. P. O'Boyle, M. I. Cole, C. Charkrabarti, D. T. Blaauw, H. Kim, T. N. Mudge, and R. G. Drewlinski, "Transmuter: Briding the efficiency gap using memory and dataflow reconfiguration," *PACT*, 2020.
[92] M. D. Gomony, F. d. Putter, A. Gebregiorgis, G. Paulin, L. Mei, V. Jain, S. Hamdioui, V. Sanchez, T. Grosser, M. Geilen, M. Verhelst, F. Zenke, F. Gurkaynak, C. de Bruin, S. Stuijk, S. Davidson, S. De, M. Ghogho, A. Jimborean, S. Eissa, L. Benini, D. Soudris, R. Bishnoi, S. Ainsworth, F. Corradi, O. Karrakchou, T. Guneysu, and H. Corporaal, "CONVOLVE: Smart and seamless design of smart edge processors," *CoRR*, 2023.
[93] B. Yuan, J. Zhu, X. Man, Z. Ma, and S. Yin, "Dynamic-II pipeline: Compiling loops with irregular branches on static-scheduling CGRA," *TCAD*, 2021.
[94] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4D-CGRA: Introducing branch dimension to spatio-temporal applicaiton mapping on CGRAs," *ICCAD*, 2019.
[95] M. Balasubramanian, S. Dave, A. Shrivastava, and R. Jeyapaul, "LASER: A hardware/software approach to accelerator complicatd loops on CGRAs," *DATE*, 2018.
[96] A. Wood, *Offset Pipelining for Coarse Grain Reconfigurable Arrays*. phdthesis, 2017.
[97] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 10 2021.
[98] G. Gobieski, S. Ghosh, T. Nowatzki, T. C. Mowry, N. Beckmann, and B. Lucia, "Riptide: A programmable, energy-minimal dataflow compiler and architecture," *Micro*, 2022.
[99] M. Karunaratne, C. Tan, A. Kulkarni, T. Mitra, and L.-S. Pen, "DNestMap: Mapping deeply-nested loops on ultra-low power CGRAs," *DAC*, 2018.
[100] T. Ruschke, J. Jung, Lukas, d. Wolf, and C. Hochberger, "Scheduler for inhomogeneous and irregular CGRAs with support for complex control flow," *PDS*, 2016.
[101] T. Kojima, A. Ohwada, and H. Amano, "Mapping-aware kernel partitioning method for CGRAs assested by deep learning," *Parallel and Distributed Systems*, vol. 33, May 2022.
[102] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: Using epimorphism to map applications on CGRAs," *DAC*, 2012.
[103] B. Egger, H. Lee, D. Kang, M. S. Moghaddam, Y. Cho, Y. Lee, S. Kim, S. Ha, and K. Choi, "A space- and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2 2017.
[104] A. Ohwada, T. Kojima, and H. Amano, "An efficient compiation of coarse-grained reconfigurable architectures utilizing pre-optimized sub-graph mappings," *PDP*, 2022.
[105] M. Kou, J. Zeng, B. Han, F. Xu, J. Gu, and H. Yao, "GEML: GNN-based efficient mapping method for large loop applications on CGRA," *DAC*, 2022.
[106] J. Lee and T. E. Carlson, "Ultra-fast cgra scheduling to enable run time, programmable cgras," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 12 2021.
[107] R. Wirsch and C. Hochberger, "Towards transparent dynamic binary translation from RISC-V to a CGRA," *ARCS*, 2021.
[108] M. Vieira, M. Caneshe, L. Braganca, J. Campos, and M. Silva, "RESHAPE: A run-time dataflow hardware-based mapping for CGRA overlays," *ISCAS*, 2021.
[109] M. Canesche, W. Carvalho, L. Reis, M. Oliveira, S. Magalhaes, P. Jamieson, J. M. Nacif, and R. Ferreira, "You only traverse twice: A YOTT placement, routing, and timing approach for CGRAs," *ACM Transactions on Embedded Computing Systems*, vol. 20, pp. 1–25, 2021.
[110] D. Liu, S. Yin, L. Liu, and S. Wei, "Polyhedral model based mapping optimization of loop nests for CGRAs," *DAC*, 2013.
[111] Y. Miyasaka, M. Fujita, A. Mishchenko, and J. Wawrzynek, "SAT-based mapping of data-flow graphs onto coarse-grained reconfigurable arrays," *IFIP Advances in Information and Communication Technology*, 2021.
[112] T. Nowatzki, M. Tarm-Sartin, L. D. Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *PLDO*, 2013.
[113] M. J. P. Walker and J. H. Anderson, "Generic connectivity-based CGRA mapping via integer linear programming," *FCCM*, 2019.
[114] S. Chaudhuri and A. Hetzel, "SAT-based compilation to a non-von neumann processor," *ICCAD*, 2017.
[115] J. W. Yoon, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek, "SPKM: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," *ASP-DAC*, 2008.
[116] S. Mu, Y. Zeng, and B. Wang, "Routability-enhanced scheduling for application mapping on CGRAs," *Access*, 2021.
[117] M. Balasubramanian and A. Shrivastava, "Pathseeker: A fast mapping algorithm for cgras," *DATE*, 2022.
[118] Z. Zhao, H. Sadok, J. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," 2020.
[119] W. Kim, Y. Choi, and J. Kim, "CGRA compilation boost up for acceleration of graphics," 2014.
[120] S. Dave, M. Balasubramanian, and A. Shrivastava, "RAMP: Resource-aware mapping for CGRAs," *DAC*, 2018.
[121] M. Kou, J. Gu, S. Wei, H. Yao, and S. Yin, "TAEM: Fast transfer-aware effective loop mapping for heterogeneous resources on CGRA," *DAC*, 2020.
[122] C. Yang, L. Liu, K. Luo, S. Yin, and S. Wei, "CIACP: A correlation- and iteration- aware cache partitioning mechanism to improve performance of multiple coarse-grained reconfigurable arrays," *Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 29–43, 2017.
[123] D. Liu, S. Yin, G. Luo, J. Shang, L. Liu, S. Wei, Y. Feng, and Z. Shangbo, "Data-flow graph mapping optimization for CGRA with deep reinforcement learning," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, December 2019.
[124] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciox: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning," *Micro*, 2020.
[125] A. X. M. Chang, P. Khopkar, B. Romanous, A. Chaurasia, P. Estep, S. Windh, D. Vanesko, S. D. B. Mohideen, and E. Culurciello, "Reinforcement learning approach for mapping applications to dataflow-based coarse-grained reconfigurable array," *CoRR*, 2022.
[126] Y. Zhuang, Z. Zhang, and D. Liu, "Towards high-quality CGRA mapping with graph neural networks and reinforcement learning," 2022.
[127] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine," in *the 44th Annual International Symposium*, ACM Press, 6 2017.
[128] D. Koeplinger, C. Kozyrakis, K. Olukotun, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, and A. Pedram, "Spatial: a language and compiler for application accelerators," in *the 39th ACM SIGPLAN Conference*, ACM Press, 6 2018.
[129] R. Keryell, A. Gozillon, G. Harnisch, H. Kwon, R. Chakaravarthy, and R. Wittig, "SYCL for Xilinx Versal ACAP AIE CGRA," *IWOCL SYCLCon*, 2021.
[130] Q. Liu, D. Huff, J. Setter, M. Strange, K. Feng, K. Sreedhar, Z. Wang, K. Zhang, M. Horowitz, P. Raina, and F. Kjolstad, "Compiling halide programs to push-memory accelerators," *CoRR*, 2021.
[131] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan, T. Hofstee, M. Horowitz, D. Huff, F. Kjolstad, T. Kong, Q. Liu, M. Mann, J. Melchert, A. Nayak, A. Niemetz, G. Nyengele, P. Raina, S. Richardson, R. Setaluri, J. Setter, K. Sreedhar, M. Strange, J. Thomas, C. Torng, L. Truong, N. Tsiskaridze, and K. Zhang, "Creating an agile hardware design flow," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 7 2020.
[132] Y. Zhang, N. Zhang, M. Vilim, M. Shahbaz, and K. Olukotun, "SARA: Scaling a reconfigurable dataflow accelerator," *ISCA*, 2021.
[133] M. Feldman, *Software-Defined Hardware Without Sacrificing Performance*. phdthesis, 2021.

[134] J. D. Lopes and J. T. de Sousa, “Fast fourier transform on the versat CGRA,” *Silicon Errors Logic-System Effects*, pp. 174–187, 2016.

[135] Z. E. Rakossy, D. Stengele, A. Acosta-Aponte, S. Chafekar, P. Bientinesi, and A. Chattopadhyay, “Scalable and efficient linear algebra kernel mapping for low energy consumption on the layers CGRA,” *International Symposium on Applied Reconfigurable Computing*, 2015.

[136] B.-Y. Huang, S. Lyubomirsky, Y. Li, M. He, T. Tambe, G. H. Smith, A. Gaonkar, V. Canumalla, G.-Y. Wei, A. Gupta, Z. Tatlock, and S. Malik, “Specialized accelerators and compiler flows: Replacing accelerator apis with a formal software/hardware interface,” 2022.

[137] G. H. Smith, A. Liu, S. Lyubomirsky, S. Davidson, J. McMahan, M. Taylor, L. Ceze, and Z. Tatlock, “Pure tensor program rewriting via access patterns (representation pearl),” in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pp. 21–31, 2021.

[138] Y. Ikarashi, G. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, “Exocompilation for productive programming of hardware accelerators,” *PLDI*, 2022.

[139] P. Ginsbach, B. Collie, and M. F. P. O’Boyle, “Automatically harnessing sparse acceleration,” ACM, 2 2020.

[140] J. Weng, S. Liu, D. Kupsh, and T. Nowatzki, “Unifying spatial accelerator compilation with idiomatic and modular transformations,” *Micro*, 2022.

[141] M. B. Stepp, *Equality saturation: engineering challenges and applications*. University of California, San Diego, 2011.

[142] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, “Automatically improving accuracy for floating point expressions,” *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 1–11, 2015.

[143] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock, “Synthesizing structured cad models with equality saturation and inverse transformations,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 31–44, 2020.

[144] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu, “Spores: sum-product optimization via relational equality saturation for large scale linear algebra,” *arXiv preprint arXiv:2002.07951*, 2020.

[145] A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, “Vectorization for digital signal processors via equality saturation,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 874–886, 2021.

[146] E. Ustun, I. San, J. Yin, C. Yu, and Z. Zhang, “Impress: Large integer multiplication expression rewriting for fpga hls,” in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1–10, IEEE, 2022.

[147] S. Coward, G. A. Constantinides, and T. Drane, “Automatic datapath optimization using e-graphs,” *arXiv preprint arXiv:2204.11478*, 2022.