An Architecture-Independent CGRA Compiler enabling OpenMP Applications

Takuya Kojima
Graduate School of Information Science and Technology
The University of Tokyo
Tokyo, Japan
tkojima@hal.ipc.i.u-tokyo.ac.jp

Boma Adhi, Carlos Cortes, Yiyu Tan, Kentaro Sano
Center for Computational Science (R-CCS)
RIKEN
Kobe, Japan
{boma.adhi,carlos.cortestorres,tan.yiyu,kentaro.sano}@riken.jp

***Abstract*—Coarse-Grained reconfigurable architecture (CGRA) is a promising platform for HPC systems in the post-Moore's era. A single-source programming model is essential for practical heterogeneous computing. However, we do not have a canonical programming model and a frontend compiler for it. Existing versatile CGRAs, in respect to their execution model, computational capability, and system structure, magnify the difficulty of orchestrating the compiler techniques. It consequently forces designers of the CGRAs to develop the compiler from scratch, working only for their architectures. Such an approach is outdated, given other successful accelerators like GPU and FPGAs. This paper presents a new CGRA compiler framework in order to reduce development efforts of CGRA applications. OpenMP annotated codes are fed into the proposed compiler, as recent OpenMP support device offloading to the accelerators. This property improves the reusability of the existing source code for HPC workloads. The design of the compiler is inspired by LLVM, which is the most famous compiler framework so that the frontend is built to be architecture-independent. In this work, we demonstrate that the proposed compiler can handle different types of CGRAs without changing the source codes. In addition, we discuss the effect of architecture-independent optimization algorithms. We also provide an open-source implementation of the compiler framework at** https://github.com/hal-lab-u-tokyo/CGRAOmp.

I. Introduction

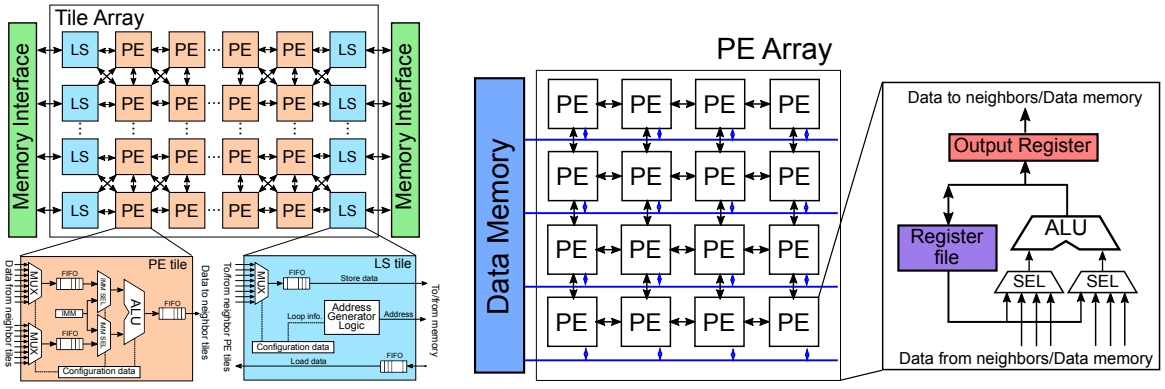
The performance of computers has been successfully improved over several decades. Most computer architects, however, are facing the challenge of keeping up with the performance scaling since the Moore's Law is ending. In order to overcome the challenge, they typically use accelerators instead of general-purpose processors for compute-intensive tasks [1]. GPUs are the most successful accelerators as they are employed for High Performance Computing (HPC) system today, including many TOP500 supercomputers [2]. Although a recent GPU offers tens to hundreds of TFLOPS performance, it suffers from significant power consumption in the order of a few hundred watts.

Another type of HPC system is emerging, which contains Field Programmable Gate Arrays (FPGAs) as computational resources [3]. FPGAs yield lower power dissipation than GPUs. FPGAs also achieve comparable or better performance than GPUs for some workloads so that the FPGAs demonstrate much higher energy-efficiency [4]. Yet FPGAs potentially have considerable timing-, area-, and power-overhead due to fine-grained, i.e., bit-level, reconfigurability. In addition, it poses a complicated and time-consuming compilation process to generate the configuration data. The Electronic Design Automation (EDA) tools can optimize the hardware design on the reconfigurable fabric to a limited extent and consequently need programmers to tune their source code manually in order to make it suitable for FPGAs.

Coarse-Grained Reconfigurable architecture (CGRA) is an alternative hardware platform to mitigate the above issues of FPGAs and promising for future HPC systems. It generally has an array of Processing Units (PE) and interconnection networks between PEs. Each PE is composed of a simple Arithmetic Logic Unit (ALU), multiplexers, and small local storage such as register file or FIFO buffer. The multiplexer selects an operand fed into the ALU from incoming data sent by neighbor PEs. The PE array can form a specialized pipeline for the target computation kernel by optimizing the mapped instruction into ALUs and the interconnection. In this respect, the reconfigurability of CGRAs is counted as a word-level, i.e., bit-width of ALU like 32-bit and 64-bit. It is much coarser than FPGAs. Therefore, CGRAs save the overheads for the reconfiguration and bring near-ASIC energy efficiency [5]. Furthermore, the coarser granularity alleviates the complexity of the compilation process, and thus the compiler affords to apply more aggressive optimization.

The compilation flow for CGRAs is usually divided into two steps: 1) dataflow extraction from application codes and 2) mapping the dataflow into the PE array (Place-and-Route). The extracted dataflow is represented as Data Flow Graph (DFG) or Control Data Flow Graph (CDFG), and it is passed to the next step. CGRAs have a wide variety of properties regarding execution style, supported instructions, and capability of handling control flow, including conditionals and nested loop structure. Compiler designers definitely have to implement the backend mapping algorithm to suit their target CGRAs. Nevertheless, they are forced to develop the frontend to generate the DFGs as well because feasible DFGs are different depending on the target CGRAs even for the same application kernel. As a result, existing CGRA compilers have been developed based on their own programming model or are specialized for a certain CGRA, even though there should exist a lot of common tasks in the compilation flow. Moreover, it is impossible for us to compare different types of CGRAs directly and compare CGRAs with other accelerators such as GPUs.

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00112



(a) RIKEN-CGRA: a case of spatially configured CGRA

(b) A classical CGRA with time-multiplexing of configuration

Fig. 1: Example CGRAs with different reconfiguration styles

This paper proposes a new compiler framework independent of a specific CGRA and targeted for any CGRAs. The design of the compiler is inspired by a concept of LLVM [6], which is one of the most popular compiler infrastructures. It helps us make an appropriate design choice by a fair comparison between different styles of CGRAs. In addition, the proposed compiler uses OpenMP offloading, which is a standard heterogeneous programming model. Thanks to such a standard model, only limited efforts are required to reuse existing source codes for HPC workloads since many of the source codes are written in either C/C++ or Fortran, which is supported in OpenMP. Moreover, several HPC benchmarks are already implemented with OpenMP [7]. To best our knowledge, this work is a first attempt for CGRAs to use OpenMP for the programming model and Fortran as a frontend language in addtion to C language.

II. Background

A. Coarse-Grained Reconfigurable Architectures

An application runtime is usually dominated by loops so that accelerators executing the loops faster and more energy-efficiently than CPUs are indispensable. CGRAs are promissing accelerators which can exploit both data-level and instruction-level parallelisms in the loops.

CGRAs are categorized into two groups based on their reconfiguration policy: 1) Spatially configured CGRAs and 2) ones with time-multiplexing configuration. The former type of CGRAs forms a fully pipelined dataflow on the reconfigurable fabric and keeps the configuration during runtime. It has advantages in terms of computational throughput and reconfiguration overheads, whereas it often struggles with the exhaustion of hardware resources. If the target dataflow demands more resources than those the CGRA physically owns, the compiler partitions the dataflow prior to the mapping [23]–[25]. In the early era when CGRAs emerged, such a style was commonly adopted as in XPP [26], and many research efforts have been focussing on it again recently [27]–[30]. Figure 1(a) describes an example of the spatially configured CGRA called RIKEN CGRA [28]. The leftmost and the rightmost columns of tiles are Load Store (LS) units responsible for memory access. The LS tile contains an address generation logic to provide regular memory access. Each PE tile inside the array consists of an ALU and a few FIFO to synchronize operand data. When required operands arrive, an operation of the ALU automatically fires.

CGRAs in the latter category execute the loop kernel while changing the configuration repeatedly. They generally handle several iterations simultaneously in a software pipeline manner to improve the thoughtput. Figure 1(b) shows a classical CGRA with a time-multiplexing configuration, such as ADRES [31], which is a representative CGRA, and many research has adopted a similar architecture especially on the topic of mapping method such as [32]. Those CGRAs can share and switch the interconnection links dynamically, unlike the statically configured CGRAs. Even though such a reconfiguration is more power-hungry, it enhances the possibility to route the dependent PEs with limited physical routing resources. In addition, the flexible reconfiguration offers a variety of design options to support complicated control flow.

Partial predication [33] is the most common technique for CGRAs to handle conditional statements inside the loop, i.e., *if-else* part. It transforms the control dependencies into corresponding data dependencies, replacing the control flow with comparators and phi nodes to select values from the taken path. Although this technique is available for both classes of CGRAs, it brings about a large DFG, especially for nested conditionals. Therefore, some sophisticated techniques have been proposed to handle the conditionals statements efficiently [14], [34]–[36].

B. Existing frontend compilers for CGRAs

As explained in Section I, the CGRA compilers fulfill two tasks to transform the application code written in some high-level languages into the configuration of the CGRAs. The first task is to extract the compute-intensive kernel from the code and generate a DFG compatible with the target CGRA. It is regarded as a frontend of the compilation flow. Then, the extracted DFG is mapped to the PE array, the configuration file corresponding to the mapping result is generated at the next stage. This process strongly depends on the target CGRA and is considered as a backend.

The backend, especially for the mapping method, is a

TABLE I: Comparison of the existing CGRA compilers from the viewpoint of the frontend characteristics

Reference	Kernel code	Assumed execution model		Freedom of PE instructions	DFG transformation	Targets
		Spatially configured	Time multiplexed			
PipeRench Compiler [8]	C-like DSL	√	×	Fixed	-	PipeRench
Musketeer [9]	C	×	√	Fixed	-	Renesas STP
XPP-VC [10]	C	√	×	Fixed	-	PACT XPP
SambaFlow [11]	PyTorch, etc	√	×	Fixed	-	SambaNova
DRESC [12]	C	×	√	Fixed	-	ADRES
CCF [13]	Annotated C	×	√	LLVM-IR	-	ADRES-like
Das *et al.* [14]	C	×	√[1]	-	Implemented	IPA
Kim *et al.* [15]	OpenCL	×	√	Fixed	-	SRP[16]
CGRA-ME [17]	C	√	√	A subset of LLVM IR	-	Any [2]
OpenCGRA [18]	Python DSL C/C++	×	√	Any LLVM-IR	-	Any [2]
Bonzini *et al.* [19]	C	√	×	Composition	Clustering	EGRA
DSAGEN [20]	Annotated C	√	×	Composition	Diverse	Any of SA [3]
CHiPReP [21]	Annotated C	√	△[4]	LLVM-IR [5]	Clustering	HiPReP
DFGenTool [22]	C	×	√	LLVM-IR	-	-
This work	OpenMP	√	√	Any	Customizable	Independent

[1] The execution model is specialized to handle control flow and is different from the ADRES-like model.
[2] Any CGRA, as far as the framework can describe
[3] Spatial Accelerator: a superset of spatially configured CGRA
[4] The architecture can map more than one instruction into a single PE with time-multiplexing while it differs from ADRES-like CGRAs.
[5] Instruction replacement to utilize compound operators by a subgraph matching after DFG generation

hot topic of CGRAs since the quality of mapping has a considerable impact on the computation throughput and energy efficiency. Thus, new methods are being investigated every year [37]–[40]. In contrast to the backend, a programming methodology for CGRAs has attracted much less attention. Thus, there exists no canonical programming model and the frontend compiler, even though such a modern parallel programming model as OpenMP, OpenACC, and CUDA should be utilized for the CGRAs, as claimed in [5]. Nonetheless, the frontend is essential in the compilation flow for any CGRAs. As a result, the compiler designer for each CGRA develops a frontend tightly coupled with the backend and confines it into its own compilation toolchain.

Table I summarizes the characteristics of the existing CGRA compilers. Each of them has one or more of the scalability issues as follows:

- A dedicated programming model
- Constant behavior due to targeting a specific architecture
- Less flexibility to exploit any custom instructions
- No consideration for customizing middle-end optimization

Most compilers use ANSI C as the programming launguage. Some of them are based on their original syntax for kernel annotation. SambaFlow for SmabaNova [11] is basically targeting machine learning applications. Therefore, it supports the de facto standard frameworks for machine learning, such as TensorFlow and PyTorch. OpenCL is a programming model for heterogeneous computing and has been introduced to SRP[16]. Yet it seems to have a limitation to applying other CGRAs.

DFGenTool [22] provides only frontend and is similar to this work. However, generated DFGs are constant to the application code even though various CGRAs need different forms of DFGs. The compilers in the uppermost four rows in Table I are software tools for the commercial products of CGRAs. Therefore, it is no wonder that they support only their CGRAs. Nevertheless, most others also focus on a specific type of CGRA. It means the behavior of the frontend is always definite. DSAGEN [20] is a remarkable framework for design space exploration of *Spatial Accelerator*, which is a superset of spatially configured CGRAs. It changes the generated DFG depending on the target accelerator and verifies whether the application code is compatible. On the other hand, it cannot be utilized for the time-multiplexed CGRAs.

Most compilers consider only primary operators like arithmetic binary operators, logical ones, and comparators since the instruction set architecture of their targeting CGRA has only those instructions. Some of CGRAs have a PE containing a cluster of ALUs like EGRA [19] or supporting compound instruction like multiplied-accumulate (ACC) [21]. In this case, their compilers try to find a sub-graph with pattern matching techniques to aggregate several primary instructions into a single compound one. Such a graph transformation is integrated into the compilation flow. However, this approach cannot leverage rich mathematical functions such as $\sin$ and $\exp$.

Another type of graph transformation in advance of the mapping is also proper. For example, a high fanout node is eliminated by inserting a recomputation node like [14], [32]. The graph transformations are regarded as a middle-end process of the compilation, whereas there is no framework allowing a plugin to customize the transformations. It requires much effort to implement a novel algorithm for middle-end optimization.

III. OpenMP compiler for CGRAs

To address the scalability issues of the existing CGRA compilers, this study proposes a flexible compilation framework based on the OpenMP programming model. The offloading capability had been supported in OpenMP 4.0, especially for supporting GPUs. In addition, OpenMP supports C/C++ and Fortran. Such standardized model and multi-language support encourages the reuse of existing codes and fair comparison between different accelerators.

Code 1 is a snippet from *jacobi-1d-imper* based on Poly-

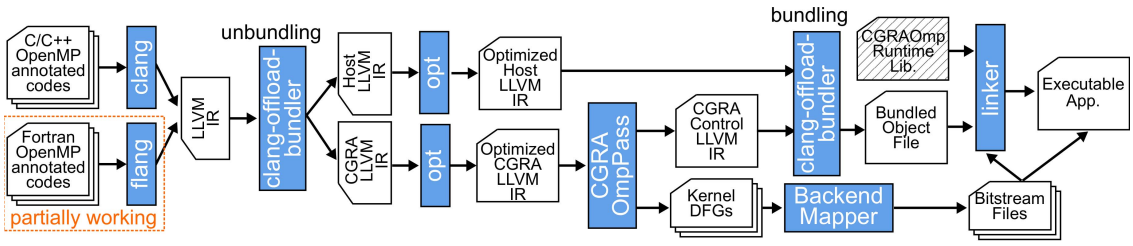


Fig. 2: Overview of the compilation flow for CGRA OpenMP

Code 1: Code snippet for OpenMP offloading

```
1 #pragma omp target data map(to:A[0:N]) map(from:B[0:N])
2 {
3     #pragma omp target parallel for private(i)
4     for (i = 1; i < _PB_N - 1; i++)
5         B[i] = 0.33333f * (A[i-1] + A[i] + A[i + 1]);
6 }
```

Bench/ACC benchmark suite [41]. For the OpenMP offloading, target directive specifies the kernel executed on accelerators. The map clause gives the compiler a hint to manage data transfer between the host and the CGRA. For example, a map type to copys data to the CGRA at the beginning of execution but does not write the data back to the host at the end of the computation.

A. Overview of the compilation

The proposed framework is implemented as an extension of LLVM [6]. LLVM consists of serval subprojects including C/C++ frontend Clang, Fortran frontend, and OpenMP. Therefore, the framework can be built with minimal effort.

Figure 2 describes an overview of the proposed compilation flow. It is automatically processed by a compiler driver we have developed. The frontends compile application codes written either in C or Fortran into LLVM Intermediate Representation (IR). The LLVM-IR has both codes executed by the host CPU and ones to be offloaded. The LLVM toolchain contains *clang-offload-bundler*, which is a utility tool for heterogeneous single-source programming and separates code objects for the host CPU and accelerators. Thereby, the compiler driver extracts the kernels executed on a CGRA from the bundled codes.

LLVM provides valuable and powerful analysis and optimization functionalities as *Passes*. The extracted codes of the CGRA kernel are optimized by the built-in passes beforehand CGRA-centric process. For example, codes optimization available regardless of the target hardware, such as dead code elimination, constant folding, and redundancy elimination, are applied.

The well-optimized LLVM-IR is fed into *CGRAOmpPass*. The behavior of the pass is changed depending on the target CGRA model, as explained later. The pass verifies whether a kernel code can be executed, extracts the verified kernels as DFGs, and replaces the kernel codes with appropriate runtime functions for reconfiguration, data transfer, and synchronization. However, the runtime part is currently under developing. The exported DFGs are fed into any backend mapper for the target CGRA. Finally, the configuration file for each kernel is created. It is statically liked into the execution binary or is loaded dynamically at runtime.

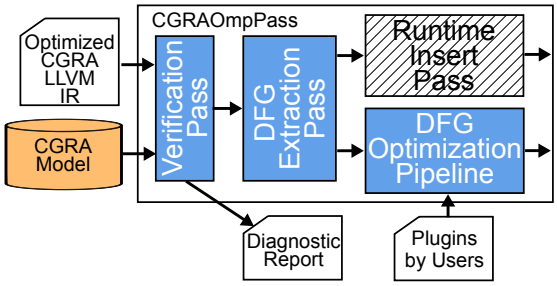


Fig. 3: Sequence of the CGRAOmpPass

The remaining LLVM-IR after the replacement contains only the control part so that it is merged into the host LLVM-IR by *clang-offload-bundler* again. The framework is designed to provide a template for runtime libraries. Therefore, the CGRA designers have only to implement a library derived from the template to suit their systems. In this way, it is expected to be used even for some standalone CGRAs, which have no host processor.

B. Details of CGRAOmpPass

The *CGRAOmpPass* is composed of several internal passes, as shown in Fig. 3. A CGRA model described in the JSON file is loaded and a model instance is created when the pass is invocated. It tells characterization of the target CGRA to the internal passes. Each internal pass is abstracted, and actually executed pass is selected according to the specified model with the object oriented design similar to LLVM. Thus, it required little effort to extend these passes to support a new type of CGRA model. As mentioned previously, part of the compilation related to the runtime function is left for future work.

1) Code Verification

VerificationPass verifies whether the candidates of CGRA kernels are accepted for the target CGRA. The verification categories are as follows:

- All of the needed instructions supported
- Containing only allowed conditional parts
- Loop nested structure canonicalized
- Having only allowed inter-loop dependencies
- Memory access pattern

Some CGRA models do not have any constraints on certain items. In this case, these items are skipped. For example, the

time-multiplexed CGRAs usually does not have limitations about memory access pattern, whereas as for the spatially configured one, the available memory access patterns are sometimes restricted.

If a kernel does not satisfy any of these constraints, it is ignored for the offloaded kernel to CGRAs. In case of the code violations, the pass tells users the reason as a diagnostic report. It is helpful for design space exploration to analyze which capabilities are essential in a target application region.

2) DFG extraction

Each verified kernel is given to *DFGExtractionPass*, and a corresponding DFG is generated. Each DFG is finally exported in DOT format, which is a widely used graph description language and makes it easy to connect other back-end tools as in [13], [17]. Some attributes of the nodes and the edges are customizable by the command line option. An internal representation of the DFG is derived from LLVM built-in classes. Therefore, several utilities for graphs implemented in LLVM, such as breadth-first search, are also available. In order to avoid losing generality, edges attributed to loop-carried dependencies have the distance between the dependent iterations as additional information.

3) DFG Optimization

Users can apply a sequence of optimizations for each DFG before exporting. Currently, a tree height reduction algorithm [42] is included as a built-in optimization. It tries to make the graph more balanced as far as possible, exploiting commutativity and associativity of operators. The tree height reduction is important optimization and is commonly used for VLSI design and high-level synthesis of hardware description language since it can reduce the critical path length and expose more instruction-level parallelism [43]. The floating point operations are non-associative generally because of rounding error. Thus, the transformation works for floating-point operation only at the expense of precision with options such as -fast-math.

Moreover, the pass accepts user implemented plugins and applys them into the DFGs with the command line option. Therefore, it is easy to add another graph transformation introduced in Section II-B and also target-specific graph optimization.

IV. Model description

As the previous section explains, the CGRA model for the compilation is defined as a JSON file. It should include enough information for the code verification. This work considers two categories: 1) *decoupled* and 2) *simple time-multiplexed*. The first one is a subclass of the spatially configured CGRA. For CGRAs in this category, the memory access part and computational part must be separated as in [20]. CGRAs in the second class are the ADRES-like CGRA.

Code 2 gives an example of the model description, which is a compatible model with RIKEN CGRA [44]. The field with "address_generator" describes what types of memory access are allowed for the memory access. This example means the CGRA can handle affine access with up to three loop induction variables, i.e., accessed address must be expressed by $C_0X_0 + C_1X_1 + C_2X_2$. It is a dedicated field for the *decoupled* class. The analyzed memory access information is also exported together with the DFGs and is generally packed into the configuration data.

Code 2: An example of model description

```
{
    "category": "decoupled",
    "address_generator": {
        "control": "affine",
        "max_nested_level": 3
    },
    "conditional" : {
        "allowed": false
    },
    "inter-loop-dependency": {
        "allowed": false
    },
    "generic_instructions": [
        "add", "sub", "mul", "udiv",
            ↪ "sdiv", "and", "or",
            ↪ "xor", "fadd", "fsub",
            ↪ "fmul", "fdiv" ],
    "custom_instructions": ["fexp",
        ↪ "fsin", "fcos", "fpow"],
    "instruction_map": [
        {"inst": "xor", "rhs": {
            ↪ "ConstantInt" : -1},
            ↪ "map": "not"},
        {"inst": "xor", "map": "xor"}
    ]
}
```

Code 3: Application code exploiting a custom instruction

```
CGRAOMP_CUSTOM_INST float FMA(
        float x, float y, float z) {
    return x * y + z;
}
void kernel(...)
....
    #pragma omp target parallel for private(i)
    for (i = 1; i < N - 1; i++) {
        Z[i] = FMA(A, X[i], Y[i])
    }
...
```

The following two fields are for the capabilities of handling the control dependencies. An application code with conditional statements or inter-loop dependencies cannot be compiled for the CGRA expressed by this example. This setting changes the behavior of the verification and the form of DFG to be created.

The remaining fields are used to describe the instruction set architecture of the target CGRA. The "generic_instruction" indicates which the primary operator defined in LLVM-IR are available for the CGRA. The "custom_instruction" enumerates custom instructions which the PE can execute and cannot be expressed in LLVM-IR. In the application source code, the same name function with an attribute CGRAOMP_CUSTOM_INST must be declared as shown in Code 3. If the CGRA model does not support fused multiply-add FMA instruction like Code 2, the compiler automatically performs inlining as far as the function can be inlined.

The last field makes a conversion table from LLVM-IR to the CGRA ISA. It remaps an LLVM instruction with additional

TABLE II: Comparison of the generated DFGs

Benchmark	Decoupled CGRA			Simple-time-multiplexed		
	N_V	N_E	N_L	N_V	N_E	N_L
convolution-2d	27 (17,9,1)	26	2	45	54 (52, 2)	1
convolution-3d	32 (20,11,1)	31	3	58	69 (67, 2)	1
jacobi-1d-imper	7 (3,3,1)	6	1	13	16 (14, 2)	1
jacobi-2d-imper	11 (5,5,1)	10	2	14	17 (15, 2)	1
seidel-2d	Violate			32	41 (37, 4)	1
fft-radix2	18 (10,4,4)	20	1	31	49 (46, 3)	1
fft-radix3	40 (28,6,6)	50	1	69	96 (93, 3)	1
fft-radix4	50 (34,8,8)	64	1	76	126 (123, 3)	1
fft-radix5	64 (44,10,10)	78	1	96	156 (153, 3)	1

N_V: the number of nodes, N_E: the number of edges, and N_L: loop nested level extracted as the DFGs

conditions to the CGRA instruction. In this example, xor instructions whose right-hand side of the operands is "-1" are treated as not instruction for the CGRA, whereas the other xor instructions are still mapped to xor ones as for the CGRA. If there is no entry for a generic instruction supported in the CGRA, remapping does not occur. Thereby, the compiler provides the flexibility to express the instructions of CGRAs.

V. Experiments

This section demonstrates that the proposed compiler can alter its behavior of the code verification and DFG generation based on the target CGRA model without changing the source code. Moreover, we analyze the effect of the pre-optimization before the mapping process.

A. Setup

The proposed compiler was implemented as an extension of LLVM/Clang 12.0.1. Flang version 20211221 [45] was employed for Fortran frontend. In order to evaluate the compiler, we selected five benchmarks: *convolution-2d, convolution-3d, jacobi-1d-imper, jacobi-2d-imper, and seidel-2d* from PolyBench/ACC [41]. This benchmark suite has several flavors of implementations, such as OpenMP and OpenACC. We used the OpenMP version and just modified the directive for offloading. In addition, we prepared Stockham Fast Fourier Transform with radix-2, 3, 4, and 5 as a representative scientific application.

Considering the case where the CGRA supports the FMA instruction like Code 3, these FFT codes are written to use as many FMA instructions as possible. Please note that the function call in the loop kernel is inlined when the CGRA does not support the dedicated instruction. This evaluation targets two CGRA models: 1) the same model as Code 2 and 2) simple-time-multiplexed allowing the loop-carried dependencies without any conditional statement support. Thus, the second class of CGRAs cannot control the nested loop directly. In this case, only the innermost loop is extracted as a kernel.

B. Verification results and generated DFGs

We compare the DFGs for both models. Table II describes the statistics regarding the generated DFGs. Despite the same source code, each has a different structure, as emphasized through the paper. As for the decoupled CGRA, size for computational nodes, memory load, and memory store are respectively shown inside the parenthesis in addition to the total number of the nodes. The table also explains how many edges due to loop-carried dependencies are included in DFGs for the simple-time-multiplexed CGRA. The left side is data dependencies within an iteration, while the right one is attributed to the loop-carried dependencies.

TABLE III: DFG size reduction by enabling custom instruction for FFT kernels

Radix	Normalized sizes of	
	nodes	edges
2	0.90	0.90
3	0.80	0.92
4	0.78	0.91
5	0.82	0.85

In addition to the C implementation of *jacobi-1d-imper*, we manually translated it to Fortran with OpenMP pragmas. Then, we validated that our compiler can extract the same DFGs based on the Fortran code. As explained in Fig. 2, the code was firstly compiled to LLVM-IR with Flang, while the remaining process is the same as the case of C codes. This evaluation result suggests Fortran is a new option for CGRA programming. Unfortunately, Flang does not fully support the syntaxes of target directive yet. That is why our compiler driver cannot automate the compilation flow completely for Fortran.

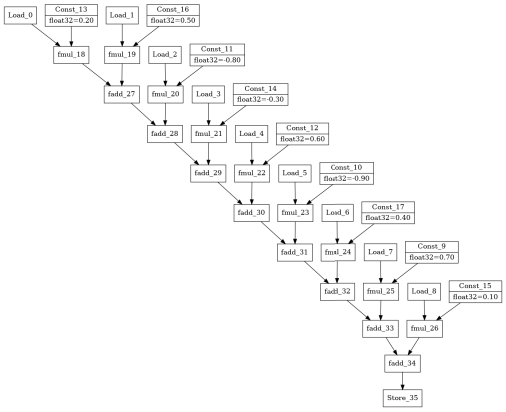
At runtime, the decoupled CGRA does not have to calculate the memory address on the computational resources, e.g., ALU in the PEs. Instead, the configuration for the dedicated memory access controller, such as the LS unit of RIKEN CGRA, is generated at compile time. Thus, the DFGs are smaller than that for the time-multiplexed one. Furthermore, it can execute the nested loops, as far as they are perfectly nested and the memory access patterns are compatible with the controller. However, the studied model does not permit any inter-loop dependencies so that it loses an opportunity to accelerate *seidel-2d*.

On the contrary, the simple-time-multiplexed CGRA accepts such a dependency, and thus *seidel-2d* is available. Thanks to the proposed compiler, we can analyze and discuss the tradeoffs between different architectures without changing the benchmarks. It helps to justify design choices in the early design stages. For example, if we target applications which has more complex memory access, conditional statements, and imperfect loop nest than the studied ones, another type of CGRA should be considered.

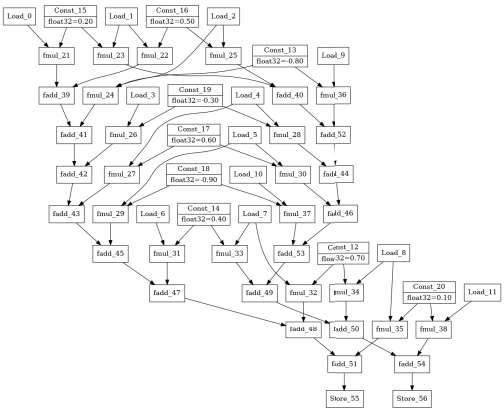
We also evaluate the DFG sizes when enabling the custom instructions. For a case study, FMA instruction is added to the "custom_instructions" field in the model description, and then FFT codes are compiled with the modified model. Table III shows how much the DFGs size is reduced. The custom instruction contributes to around 10-20% of the size reduction in nodes and edges. Practically, these effects have to be assessed considering hardware overhead to support such an instruction.

C. Discussion on the optimization effects

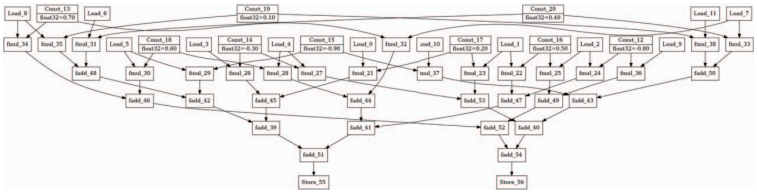
Nextly, we discuss necessary optimization, focussing on the decoupled CGRAs. Loop unrolling is a possible option in order to exploit the reconfigurable fabric of the spatially configured CGRAs. LLVM has already implemented the loop unrolling optimization. Thus, we tested the loop unrolling, giving the compiler a hint of unrolling count. Nonetheless, we ob-



(a) DFG without loop unrolling



(b) Unrolled DFG with unrolling factor 2



(c) DFG after applying tree height reduction to the DFG (b)

Fig. 4: Generated DFGs with different optimization settings

served built-in optimization sequence such as "-O2" in LLVM brings the unrolled loop kernels while collapsing the perfectly nested loop structure during the optimization. We thoroughly surveyed an appropriate sequence of the optimization, and the following sequence for *opt* tool brings expected results in our case: –indvars -loop-unroll -simplifycfg -loop-simplify -loop-instsimplify -loop-rotate -mem2reg -constmerge -simplifycfg -indvars -polly-canonicalize. Figure 4(a) and Figure 4(b) show the obtained DFGs with and without loop unrolling for *convolution-2d*. They are built for the decoupled CGRAs. As explained in Section IV, the loop control is handled by the memory access unit and the configuration for the loop control is omitted in those figures.

The kernel is convolution for 2D images with a 3x3 kernel and stride 1. Six-pixel data of the images between two consecutive iterations are identical. Hence, the number of memory load instructions in the unrolled DFG is reduced to twelve thanks to the pre-optimization.

In addition to the pre-optimization for LLVM-IR, we evaluate the tree height reduction algorithm, which is a built-in optimization for DFGs in our compiler, as explained in Section III. The optimized DFG is illustrated in Figure 4(c). The critical path length is reduced to 6 from 10, and the tree structure is well-balanced. It strongly depends on the target architecture which of DFGs is better. In addition, it is also different what kind of graph transformation is needed. The proposed compiler provides an easy way to evaluate such an optimization since the optimization sequence can be easily changed, and it offers the interface for users to implement their algorithm as plugins.

VI. Conclusion

In this paper, we have proposed a compiler for CGRAs based on the OpenMP programming model. It encourages us to reuse existing application codes written in C or Fortran. Furthermore, the proposed compiler is designed to be independent of a specific architecture, considering there exist different CGRAs in terms of their reconfiguration policy, capabilities of handling control flow, and supported instructions. Thanks to the scalable features, it enables us to compare different CGRA models without changing application source code. In this work, simple two CGRA models has been used. For future work, we plan to add futher CGRA models and develop the OpenMP runtime for CGRAs. In addition, we should discuss other syntaxes of OpenMP needed for CGRAs.

Acknowledgement

This work is partially supported by the New Energy and Industrial Technology Development Organization (NEDO) Project JPNP16007 and Japan Society for the Promotion of Science (JSPS) KAKENHI Grant 19J21493.

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[2] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Belloum, and R. V. Van Nieuwpoort, "The landscape of exascale research: A data-driven literature analysis," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–43, 2020.

[3] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of FPGAs for heterogeneous platforms in HPC," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, 2015.

[4] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," in *2018 IEEE*

26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2018, pp. 93–96.

[5] 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: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[7] P. Jamieson, A. Sanaullah, and M. Herbordt, "Benchmarking heterogeneous hpc systems including reconfigurable fabrics: Community aspirations for ideal comparisons," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–6.

[8] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen, "Piperench implementation of the instruction path coprocessor," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 147–158.

[9] R. E. Corporation. Dynamically Reconfigurable Processor (DRP) Technology Development — Renesas. [Online]. Available: https://www.renesas.com/us/en/application/key-technology/artificial-intelligence/voice-face-recognition/drp-development

[10] J. M. Cardoso and M. Weinhardt, "XPP-VC: AC compiler with temporal partitioning for the PACT-XPP architecture," in *International Conference on Field Programmable Logic and Applications*. Springer, 2002, pp. 864–874.

[11] S. Systems. Accelerated Computing with a Reconfigurable Dataflow Architecture. [Online]. Available: https://sambanova.ai/wp-content/uploads/2021/06/SambaNova_RDA_Whitepaper_English.pdf

[12] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEEE Proceedings-Computers and Digital Techniques*, vol. 150, no. 5, pp. 255–261, 2003.

[13] S. Dave and A. Shrivastava, "CCF: A cgra compilation framework," 2017. [Online]. Available: https://github.com/MPSLab-ASU/ccf

[14] S. Das, K. J. Martin, D. Rossi, P. Coussy, and L. Benini, "An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1095–1108, 2018.

[15] H.-S. Kim, M. Ahn, J. A. Stratton, and W. H. Wen-mei, "Design evaluation of opencl compiler framework for coarse-grained reconfigurable arrays," in *2012 International Conference on Field-Programmable Technology*. IEEE, 2012, pp. 313–320.

[16] M. Konijnenburg, Y. Cho, M. Ashouei, T. Gemmeke, C. Kim, J. Hulzink, J. Stuyt, M. Jung, J. Huisken, S. Ryu *et al.*, "Reliable and Energy-Efficient 1MHz 0.4V Dynamically Reconfigurable SoC for ExG Applications in 40nm LP CMOS ," in *Proc. of ISSCC*, Feb. 2013, p. 24.6.

[17] 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, 2017, pp. 184–189.

[18] 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)*. IEEE, 2020, pp. 381–388.

[19] P. Bonzini, G. Ansaloni, and L. Pozzi, "Compiling custom instructions onto expression-grained reconfigurable architectures," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008, pp. 51–60.

[20] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 268–281.

[21] M. Weinhardt, M. Messelka, and P. Käsgen, "CHiPReP—A Compiler for the HiPReP High-Performance Reconfigurable Processor," *Electronics*, vol. 10, no. 21, p. 2590, 2021.

[22] M. Mukherjee, A. Fell, and A. Guha, "DFGenTool: A dataflow graph generation tool for coarse grain reconfigurable architectures," in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*. IEEE, 2017, pp. 67–72.

[23] W. Sheng, W. He, J. Jiang, and Z. Mao, "Pareto optimal temporal partition methodology for reconfigurable architectures based on multi-objective genetic algorithm," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pp. 425–430.

[24] G. Ansaloni, K. Tanimura, L. Pozzi, and N. Dutt, "Integrated kernel partitioning and scheduling for coarse-grained reconfigurable arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 12, pp. 1803–1816, 2012.

[25] T. Kojima, A. Ohwada, and H. Amano, "Mapping-Aware Kernel Partitioning Method for CGRAs Assisted by Deep Learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 5, pp. 1213–1230, 2021.

[26] M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, and M. Glesner, "The XPP architecture and its co-simulation within the simulink environment," in *International Conference on Field Programmable Logic and Applications*. Springer, 2004, pp. 761–770.

[27] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[28] A. Podobas, K. Sano, and S. Matsuoka, "A template-based framework for exploring coarse-grained reconfigurable architectures," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 1–8.

[29] Y. Zhang, A. Rucker, M. Vilim, R. Prabhakar, W. Hwang, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 615–628.

[30] M. Vieira, M. Canesche, L. Bragan¥ cca, J. Campos, M. Silva, R. Ferreira, and J. A. Nacif, "RESHAPE: A Run-time Dataflow Hardware-based Mapping for CGRA Overlays," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[31] B. Mei, F.-J. Veredas, and B. Masschelein, "Mapping an H. 264/AVC decoder onto the ADRES reconfigurable architecture," in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 622–625.

[32] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: Using epimorphism to map applications on CGRAs," in *Proceedings of the 49th Annual Design Automation Conference*. IEEE, 2012, pp. 1280–1287.

[33] K. Han, J. Ahn, and K. Choi, "Power-efficient predication techniques for acceleration of control flow execution on cgra," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 2, pp. 1–25, 2013.

[34] M. Balasubramanian, S. Dave, A. Shrivastava, and R. Jeyapaul, "LASER: A hardware/software approach to accelerate complicated loops on CGRAs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 1069–1074.

[35] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4D-CGRA: Introducing Branch Dimension to Spatio-Temporal Application Mapping on CGRAs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[36] M. Karunaratne, C. Tan, A. Kulkarni, T. Mitra, and L.-S. Peh, "DNestMap: Mapping Deeply-Nested Loops on Ultra-Low Power CGRAs," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[37] M. Canesche, M. Menezes, W. Carvalho, F. Torres, P. Jamieson, J. A. Nacif, and R. Ferreira, "TRAVERSAL: A Fast and Adaptive Graph-based Placement and Routing for CGRAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[38] M. Balasubramanian and A. Shrivastava, "CRIMSON: Compute-Intensive Loop Acceleration by Randomized Iterative Modulo Scheduling and Optimized Mapping on CGRAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3300–3310, 2020.

[39] M. Kou, J. Gu, S. Wei, H. Yao, and S. Yin, "TAEM: fast transfer-aware effective loop mapping for heterogeneous resources on CGRA," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[40] S. Mu, Y. Zeng, and B. Wang, "Routability-enhanced scheduling for application mapping on cgras," *IEEE Access*, vol. 9, pp. 92 358–92 366, 2021.

[41] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*. Ieee, 2012, pp. 1–10.

[42] K. E. Coons, W. Hunt, B. A. Maher, D. Burger, and K. S. McKinley, *Optimal huffman tree-height reduction for instruction-level parallelism*. Computer Science Department, University of Texas at Austin, 2008.

[43] D. L. Kuck, *Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.

[44] A. Podobas, K. Sano, and S. Matsuoka, "A template-based framework for exploring coarse-grained reconfigurable architectures," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 1–8.

[45] P. Osmialowski, "How the Flang frontend works: Introduction to the interior of the open-source fortran frontend for LLVM," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, 2017, pp. 1–14.