# CCF: A CGRA Compilation Framework

Shail Dave, Aviral Shrivastava
Compiler Microarchitecture Lab, Arizona State University, Tempe, AZ
Email: {shail.dave, aviral.shrivastava}@asu.edu

## I. Introduction

*Coarse-grained reconfigurable array* (CGRA) can efficiently accelerate even non-parallel loops [1]. Although scores of techniques have been developed in the past decade to map loops on CGRA PEs, several challenges in enabling acceleration of general-purpose applications on CGRAs remained unresolved, in particular, the automatic code generation for the CGRA accelerator coupled with modern processor cores.

In this demonstration, we showcase CCF – **C**GRA **c**ompiler **f**ramework. We show that given performance-critical loops annotated in embedded applications, how CCF extracts the loop, constructs the *data dependency graph* (DDG), maps it onto CGRA architecture, off-loads necessary configuration instructions for CGRA PEs, and automatically communicates data between the CPU and CGRA.

## II. CGRA Compiler Framework

The compute-intensive applications can be profiled to determine the top non-vectorizable performance-critical loops. These loops are annotated with a *pragma* directive and are extracted later by our clang based CGRA front-end. CCF collects all the source files and it emits an optimized *intermediate representation* (IR), which is obtained through LLVM just before *SelectionDAG* phase. CCF is implemented in LLVM 4.0 [2] and includes a set of transformation and analysis passes. One such important pass performs the liveness analysis on the IR for the loop variables, based on the use-definition chain. It ensures the automatic communication of live-in/live-out variables of the loop, between the CPU and CGRA.

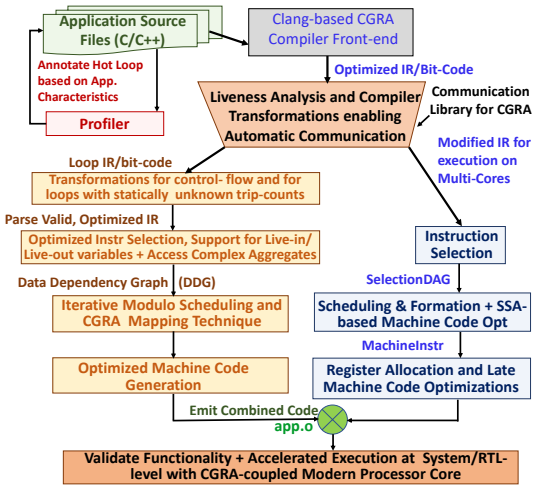


Fig. 1. A High-Level Overview of CCF

These live variables are communicated as shared data between the CPU and the CGRA, or alternatively, can be transferred to/from the CGRA memory via *memcpy*-style functions.

CGRAs typically handle the branch divergence through partial predication; loop operations from both the if- and else-path executes and then correct output is selected. Therefore, in the presence of the (complex) control flow, CCF ensures that a store and/or an update to the live-out variable occurs correctly. It parses the IR for each loop and generates a DDG, as shown in Fig. 1. DDG is a directed graph; nodes represent the operations to be executed by PEs and edges represent data dependencies among the operations. We also ensure that a valid DDG is generated in case of loops accessing sub-words/pointers. Then, an iterative modulo schedule is generated for DDG and operations are mapped on PEs in a software pipelined manner. Once the mapping is achieved, the machine instructions for CGRA PEs are generated based on the CGRA microarchitecture. If CGRA PEs manage the live operands into CGRA registers, the machine instructions to preload them and/or write-back are also generated.

Finally, the part of the IR corresponding to the loop body is purged along with an insertion of a function call corresponding to the loop execution on the CGRA. This modified IR is then taken to the machine code generation for the CPU.

In compiling the applications, we use optimization level 3 and also consider complex loops including intertwined loops and loops with dynamic trip-counts. The generated binary is evaluated on the popular cycle-accurate simulator gem5 [3] in system emulation mode; we modeled CGRA as a separate core coupled to ARM Cortex-like processor core with ARMv7a profile. In this presentation, we demonstrate the capabilities and the execution flow of our framework. Further details can be accessed at http://aviral.lab.asu.edu/cgra/.

## Acknowledgment

This work was partially supported by funding from the NSF grants CCF 1723476, 1055094 (CAREER), and CNS 1525855. We gratefully acknowledge the contributions of Mahdi Hamzeh, Mahesh Balasubramanian, Dipal Saluja and Shrihari Rajendran Radhika.

## References

[1] T. Vander Aa, P. Raghavan, S. Mahlke, B. De Sutter, A. Shrivastava, and F. Hannig, "Embedded tutorialcompilation techniques for cgras: Exploring all parallelization approaches," in *CODES+ ISSS*, 2010.
[2] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.
[3] N. Binkert *et al.*, "The gem5 simulator," 2011.