CCF 20.04 User Manual

Make Programming Simple Lab
Arizona State University

Contents

List of Figures

List of Tables

Chapter 1

Overview

CCF 20.04 (**C**GRA **C**ompilation **F**ramework **20.04**) is an end-to-end prototype that generates machine code for CGRAs (Coarse-Grained Reconfigurable Arrays) and simulate their performance. Through CCF, users can benchmark how CGRAs accelerate kernels designed for general-purpose applications. Currently, CCF 20.04 can accelerate over 70 loops from three benchmark suites, Parboil, MiBench, and Rodinia. It is named CCF 20.04 because this framework has been built and verified on Ubuntu 20.04.

1.1 CCF 20.04 Workflow

With LLVM 13.0[4] as the foundation, the implementation of the compiler for CCF includes numerous compiler analysis and transformation passes along with a customized code generation CGRA back-end. The user only needs to mark the performance-critical loops that they want to excute on the CGRA by using the annotation `#pragma CGRA`. Doing so, the CCF compiler will:

- Automatically extract the marked loop and maps it to the CGRA
- Generate code to communicate live data between the processor core and the CGRA
- Pre-load the live values into the CGRA's registers
- Generate the machine instructions to configure the PEs to execute the loop
- Generate the binary that will execute on CCF's simulator

The simulator is built by modifying the cycle-accurate computer architecture simulator gem5[5], and it models the CGRA as a separate core coupled to an ARM Cortex-like processor with the ARMv7a profile.

This open-source platform has been developed at Arizona State University and through CCF 20.04, we hope to accelerate the CGRA research by developing and making an accessible community-wide CGRA compilation infrastructure.

1.2 Organization of Source Code Directories

CCF 20.04 is the amalgamation of two open-source projects, LLVM and gem5, and a custom CGRA instruction generator. Due to the sheer size of CCF 20.04, it is important to understand the directory organization of this framework. The directory `llvm-project` contains the modified version of the LLVM compiler which serves as the front end of CCF 20.04. It is responisbile for identifying and extracting loops the user wants to accelerate on CGRAs. It also contains the passes manages the data communication between the CPU and the CGRA. These passes as referred to as `CondDDGGen`, `InvokeCGRA`, and `CGRAGen`, and their source code can be found in the `Transforms` directory.

Inside the directory `mappings`, there are many mapping algorithms implemented by our lab, including RAMP [2], FALCON+CRIMSON, and GraphMinor [1], and inside the directory `InstructionGenerator` lies the source code and binaries that generates the instructions for the CGRA. Together, they form the back-end of the framework. The directory `gem5` contains the simulator that models the execution of the CPU-CGRA platform via the system emulation mode. The directory `scripts` contains the CGRA library functions and a few other shell scripts to automate code generation.

The directory `benchmarks` contains examples of the code generation process using benchmarks from the benchmark suites Parboil, MiBench, and Rodinia.

Chapter 2

Code Generation and CPU-CGRA Simulation

To demonstrate the code generation process, we refer a custom benchmark that does binary to decimal conversion and power calculations.

2.1 Loop Annotation

Once the programmer has profiled the compute-intensive application and have identified a performance-critical loop, they can annote it with `#pragma CGRA`. The CCF 20.04 compiler will then generate the binaries to execute the application on the CPU-CGRA platform.

```
int fibonacci(int count){

 if(count == 0 || count == 1) return 0;
 int ret=1;
 int prev=1;
  #pragma CGRA
  for(int i=1; i<count-1; i++){
      int temp = ret+prev;
      prev = ret;
      ret = temp;
    }
    return ret;
}
```

FIGURE 2.1: Loop Annotation for CCF 20.04 Compiler

For example, as shown in Figure 2.1, we can annotate the loop computing powers of a number, and compile that loop for its execution on the CGRA.

2.2 Make

Once the target loop has been annotated, the code can be compiled by modifying the existing `Makefile`. As shown in Figure 2.2, the user needs only to replace the target compiler (`gcc` in our case) with `cgracc` (CGRA Compiler Collection) (`cgra++` for `c++` applications). Afterwards, executing the command `make` will compile the correct binaries.

During compilation, the CCF compiler will inform the user whether the annotated loop will be executed on the CGRA. For example, if the loop contains system calls or if the loop can be vectorized by the compiler, then it will not be executed on the CGRA. If the CCF compiler deems the loop to be executable on the CGRA, then the user will find a new directory `CGRAExec` generated. `CGRAExec` contains information about the compiled loop and its binaries (e.g `prolog_ins.bin`, `kernel_ins.bin`, and `epilog_ins.bin`).

```
FILE1 = fib.c

all: fibonacci

CC = cgracc #gcc
ARMCC = arm-linux-gnueabi-gcc
LIB = -lm
fibonacci: ${FILE1} Makefile
        $(CC) -static -O3 ${FILE1} -o fibonacci

clean:
        rm -rf fibonacci output* *.ll CGRAExec m5out *.s
```

FIGURE 2.2: Required Modifications in the Makefile

Once the compilation process is terminated, you can see that the target executables (in this case, `basicmath_small`) is generated. Meaning, the user can now simulate the execution the application on the CPU-CGRA platform.

2.3 Parameterizing the CGRA

CCF 20.04 allows the user to specify different sizes of CGRAs, ranging from 2x2 to 16x16, and different sizes of local register files. Figure 2.3 shows the entirety of file `CGRA_config.csv` that sets the parameters for the CGRA and the mapping algorithm.

The parameters for the mapping algorithm will be further explored in Section 3.4. Table 2.1 describes the CPU-CGRA parameters in file `CGRA_config.csv`:

2.4 Simulating the CPU-CGRA Platform

CCF 20.04's simulation platform is built upon gem5, where it models the CGRA as a separate core coupled to an ARM Cortex-like processor. Instead of executing the script `se.py` for system emulation mode, it instead executes a custom script `se_cgra.py` which models the CPU-CGRA execution. For the user to simulate this heterogeneous execution

```
X,4
Y,4
R,4
IC,0
Cclock,0.7
CPUclock,2
Mem,8GB
MODE,0
ALGO,RAMP
MSA,10
MAPII,10
MAX_MAP,1000
MAX_II,50
LAMBDA,0.02
```

FIGURE 2.3: Example of `CGRA_config.csv`

`X`	Size of CGRA in X-dimension
`Y`	Size of CGRA in Y-dimension
`R`	Size of local register file
`IC`	//TODO
`Cclock`	//TODO
`CPUclock`	CPU Clock speed in GHz

TABLE 2.1: Modifiable CPU-CGRA Parameters

with ease, CCF 20.04 provides a script called `cgraexe`, which is installed into the user's install directory. This binary parses through `CGRA_config.csv`, configures simulation parameters, and models CPU-CGRA execution for the compiled binary. Figure 2.4 refers to its syntax.

```
cgraexe <name of executable> --prog-args <program arguments>
```

FIGURE 2.4: Syntax of `cgraexe`

Chapter 3

Inner-workings of the Compiler

In this chapter, we describe the intermediate steps of the compilation process

3.1 Extracting the Annotated Loop

The front-end of CCF 20.04's compiler (implemented by modifying clang 13.0) identifies and marks the annotated loop from C/C++ code. Then, it generates the source code's intermediate representation (IR) (e.g. `simplified.ll`). The compiler targets the highest optimization, (i.e. optimization level 3, including auto-vectorization enabled). In the IR file, the annotated loop contains metadata (e.g. `llvm.loop.CGRA.enable`) so that the compiler knows which loop will undergo analysis and transformations. The compiler then analyzes whether the loop will be accelerated on the CGRA. If the loop can, the compiler will generate the data flow graph (DFG) of that loop.

3.2 Generating DDG and Communicating Live Data

The LLVM Pass `CondDDGGen` generates the DDG of the loop, which can be visualized using the dot tool [3]. In the DDG, the nodes show the operations to be performed and the arcs show the data dependencies. Figure 3.2 shows the DDG for the target loop in the custom benchmark.

We assume that each operation has a latency of 1 cycle. For future works, we plan on varying the latency of operations. The red arc shows a loop-carried dependence with an arc weight equal to the dependence distance. For example, there is a loop-carried dependence between operations `1` and `4`, forming a cycle with a path delay of 2 cycles. The loop-carried dependencies limits minimizing the total cycles required to finish executing a single loop iteration on the CGRA. For more information, refer to the CGRA Iterative Modulo Scheduling (IMS) [6] literature for determining recurrence-constrained II).

The gray faced operations represent the constants or live-in values. The yellow arc indicates the live arcs, that is, a load from a live-in value or a store to live-out variable (e.g. `gVar1`→`5`, or `3`→`gVar2`). The memory accesses to the live variables occur typically

```
.preheader:                                       ; preds = %.preheader.preheader, %95
  %84 = phi i32 [ %97, %95 ], [ 0, %.preheader.preheader ]
  %85 = phi i32 [ %96, %95 ], [ 0, %.preheader.preheader ]
  %86 = phi i32 [ %87, %95 ], [ 0, %.preheader.preheader ]
  %87 = add nuw nsw i32 %86, 1
  %88 = icmp eq i32 %86, 0
  br i1 %88, label %89, label %91

89:                                               ; preds = %.preheader
  %90 = call i32 @strtol(i8* nocapture nonnull %80, i8** null, i32 10) #11
  br label %95

91:                                               ; preds = %.preheader
  %92 = icmp eq i32 %87, 2
  br i1 %92, label %93, label %.loopexit.loopexit

93:                                               ; preds = %91
  %94 = call i32 @strtol(i8* nocapture nonnull %80, i8** null, i32 10) #11
  br label %95

95:                                               ; preds = %93, %89
  %96 = phi i32 [ %85, %89 ], [ %94, %93 ]
  %97 = phi i32 [ %90, %89 ], [ %84, %93 ]
  %98 = call i8* @fgets(i8* nonnull %80, i32 256, %struct._IO_FILE* %48)
  %99 = icmp eq i8* %98, null
  br i1 %99, label %.loopexit.loopexit, label %.preheader, !llvm.loop !15

.loopexit.loopexit:                               ; preds = %95, %91
  %.ph = phi i32 [ %96, %95 ], [ %85, %91 ]
  %.ph6 = phi i32 [ %97, %95 ], [ %84, %91 ]
  br label %.loopexit
```

FIGURE 3.1: Intermediate Representation (IR) showing the loop region.

just once, since the compiler manages them in the CGRA registers. The alignment of the memory access is indicated by the weight of the arc (typically 4, for 32-bit system).

To communicate the necessary variables or live data, the compiler inserts instructions to manage the data automatically through global variables. This compilation strategy avoids copies of data by adopting a shared memory model. This is possible because in our model, the CGRA is tightly coupled with the CPU core at the interface of the L2 cache or by sharing a scratchpad with the processor.

Finally, the library call pertaining to the loop execution on CGRA is inserted and the IR corresponding to the loop body is purged through the LLVM passes `InvokeCGRA` and `CGRAGen`. This modified IR `CGRAGen.ll` is then taken to the machine code generation for the CPU.

Figure 3.3 shows the new IR. We can see that a library call `accelerateOnCGRA` is automatically inserted. The argument is the loop number, and along with the help of other CGRA library functions, the compiler will generate the binaries to execute the application on the CPU-CGRA platform.

3.3 Understanding the DDG

In order to create the `.dot` file, the compiler must create many different `.txt` files containing information regarding the DDG and live variables (e.g. `loop_node.txt`, `loop_edge.txt`, `livein_node.txt`, `livein_edge.txt`, etc.). The syntax of these files can be seen in Figure 3.4.

Here, `vopc` refers to the virtual opcode of an operation in the DDG. Table 3.6 summarizes

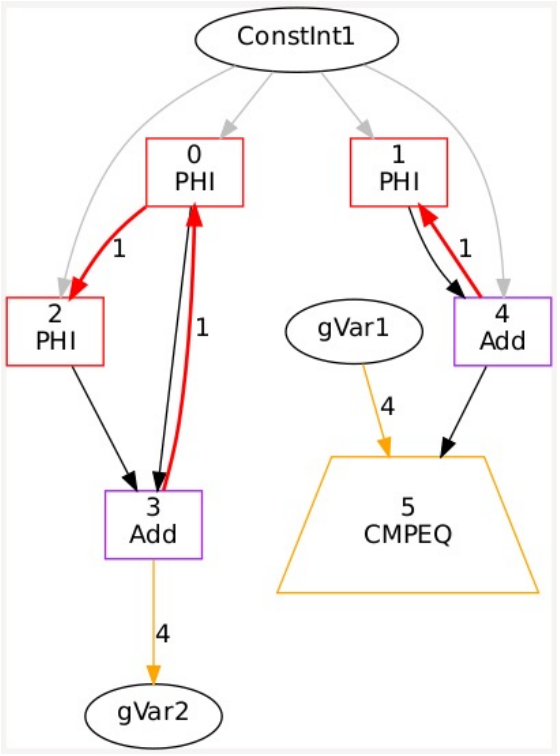

FIGURE 3.2: Data-Flow Graph (DFG) of the annotated loop.

```
; Function Attrs: nofree nounwind
declare dso_local noundef i32 @fprintf(%struct._IO_FILE* nocapture noundef, i8* nocapture noundef readonly, ...) local_unnamed_addr #2

; Function Attrs: nounwind
define dso_local void @checkTotalLoops() local_unnamed_addr #0 {
  %1 = alloca [40 x i8], align 1
  %2 = tail call i32 @puts(i8* nonnull dereferenceable(1) getelementptr inbounds ([16 x i8], [16 x i8]* @str.36, i32 0, i32 0))
  %3 = getelementptr inbounds [40 x i8], [40 x i8]* %1, i32 0, i32 0
  call void @llvm.lifetime.start.p0i8(i64 40, i8* nonnull %3) #11
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* nonnull align 1 dereferenceable(40) %3, i8* nonnull align 1 dereferenceable(40) getelementptr inbounds ([40 x i8], [40 x i8]* @__const.che
ckTotalLoops.myfile, i32 0, i32 0), i32 40, i1 false)
  %4 = call %struct._IO_FILE* @fopen(i8* nonnull %3, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @.str.7, i32 0, i32 0))
  %5 = tail call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %4, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.26, i32 0, i32 0), i32* nonnull @tota
lLoops) #11
  %6 = tail call i32 @fclose(%struct._IO_FILE* %4)
  call void @llvm.lifetime.end.p0i8(i64 40, i8* nonnull %3) #11
  ret void
}

; Function Attrs: nounwind
define dso_local i8* @runOnCGRA() local_unnamed_addr #0 {
  %1 = tail call i32 @puts(i8* nonnull dereferenceable(1) getelementptr inbounds ([12 x i8], [12 x i8]* @str.37, i32 0, i32 0))
  tail call void asm sideeffect "mov r11,$0", "r"(i32 15) #11, !srcloc !19
  ret i8* null
}

; Function Attrs: nounwind
define dso_local void @accelerateOnCGRA(i32 %0) local_unnamed_addr #0 {
  %2 = tail call i32 @puts(i8* nonnull dereferenceable(1) getelementptr inbounds ([19 x i8], [19 x i8]* @str.38, i32 0, i32 0))
  %3 = tail call i32 @configureCGRA(i32 %0)
  %4 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([35 x i8], [35 x i8]* @.str.30, i32 0, i32 0), i32 %0)
  %5 = tail call i32 @puts(i8* nonnull dereferenceable(1) getelementptr inbounds ([12 x i8], [12 x i8]* @str.37, i32 0, i32 0)) #11
  tail call void asm sideeffect "mov r11,$0", "r"(i32 15) #11, !srcloc !19
```

FIGURE 3.3: Modified IR with CGRA execution commands.

```
Format of an entry in node files:
<node #> <vopc> <node name> <mem alignment>
Format of an entry in edge files:
<from node #> <to node #> <arc weight> <edge type> <op order>
```

FIGURE 3.4: Syntax for DDG `.txt` files

the various opcodes and their corresponding operations. It also shows the corresponding LLVM instruction opcodes. Please note that the `vopc` is **not** the actual opcode embedded in the CGRA machine instructions. The true CGRA opcodes are describe later along with a discussion on the CGRA's microarchitecture. It is important to note that many

LLVM opcodes are realized using these `vopcs` (e.g. `Trunc` or `ZExt` is translated to `andop`, or `GetElementPtr` is realized using `add`, etc.).

Memory alignment is for memory access operations, and 0, otherwise. Node name is often the same as the node number, except for constants and live values.

For edge files, arc weight denotes the dependence distance for a loop-carried dependency. Edge types are `TRU` (true dependency for most arcs, including loop-carried dependencies), `LRE` for load operations (an arc between operations with vopc ld_add and `ld_data`, `SRE` for store operations (an arc between operations with `vopc st_add` and `st_data`, `LIE` for data dependency for live-in operations, and `PRE` for data dependency from predicated execution. Typically, operation with `vopc cond_select` has an input arc of type `PRE`.

`<Op Order>` denotes the operand order in case an operation has more than one operand.

Opcode	`vopc`	LLVM Opcode
0	add	Add
1	sub	Sub
2	mult	Mul
3	div	SDiv
4	shiftl	Shl
5	shiftr	Ashr
6	andop	And
7	orop	Or
8	xorop	XOR
9	cmpSGT	ICMP_SGT
10	cmpEQ	ICMP_EQ
11	cmpNE	ICMP_NE
12	cmpSLT	ICMP_SLT
13	cmpSLEQ	ICMP_SLE
14	cmpSGEQ	ICMP_SGE
15	cmpUGT	ICMP_UGT
16	cmpULT	ICMP_ULT
17	cmpULEQ	ICMP_ULE
18	cmpUGEQ	ICMP_UGE
19	ld_add	Load
20	ld_data	
21	st_add	Store
22	st_data	
23	ld_add_cond	reserved
24	ld_data_cond	reserved
25	loopctrl	special function
26	cond_select	Select
27	route	special function
28	llvm_route	reserved
29	select	PHI
30	constant	ConstantIntVal
31	rem	SRem
32	sext	SExt
33	shiftr_logical	LShr
34	rest	default

TABLE 3.1: Translation of `vopc` to LLVM IR Opcode

MODE	//TODO
ALGO	The name of the algorithm the user wants to use
MSA	//TODO
MAPII	//TODO
MAX_MAP	//TODO
MAX_II	//TODO
LAMBDA	An exploration parameter in FALCON+CRIMSON

TABLE 3.2: Configurable Mapping Parameters

3.4 Mapping the DDG onto the CGRA

The mapping algorithm maps the DDG onto the CGRA, generates the corresponding prologue, kernel, and epilogue. CCF 20.04 has implementations of various mapping algorithms including RAMP, FALCON+CRIMSON, and GraphMinor.

Add Figure of Mapping

Figure 3.5 shows a portion of the prologue, the entirety of the kernel, and a portion of the epilogue of the mapping. The vertical axis is the time scale, and the horizontal expansion shows the spatial execution on PEs. Here, at every time instance, we can see the execution for 16 different PEs. This is because the current compilation target is set to a 4x4 CGRA where each PE has 4 local registers.

We can see that each spot of the PE is represented by an operation. For example, in the prologue, operation `0` is mapped first onto `PE1` at time 6, and then at time 13. The letter `F` indicates that corresponding PEs are free at that time. After time 13, all the operations are mapped, and the schedule corresponding to prologue ends. Then, the file shows the kernel. The II of the mapped DDG is 7 due to a loop-carried dependence with a path delay of 7. Meaning, every operation will repeat its execution after 7 cycles.

This loop iterates in total of 32 times, in which the kernel repeats for 30 times. In the kernel shown, we can see that the operations are labeled a number inside a the parentheses, which indicates the result of which operation is occupied during loop execution. Operation values scheduled at a distance are typically routed through the register file. For example, dependency `10→15` is routed through a register (`10` produces its output at modulo time 1, which is read by the operation `15` at modulo time 3).

As shown previously in Figure 2.3, the user can not only configure the CPU-CGRA platform parameters, they can also tweak the mapping algorithm's parameters. Table 3.2 shows the parameters the user can configure for their mapping algorithm.

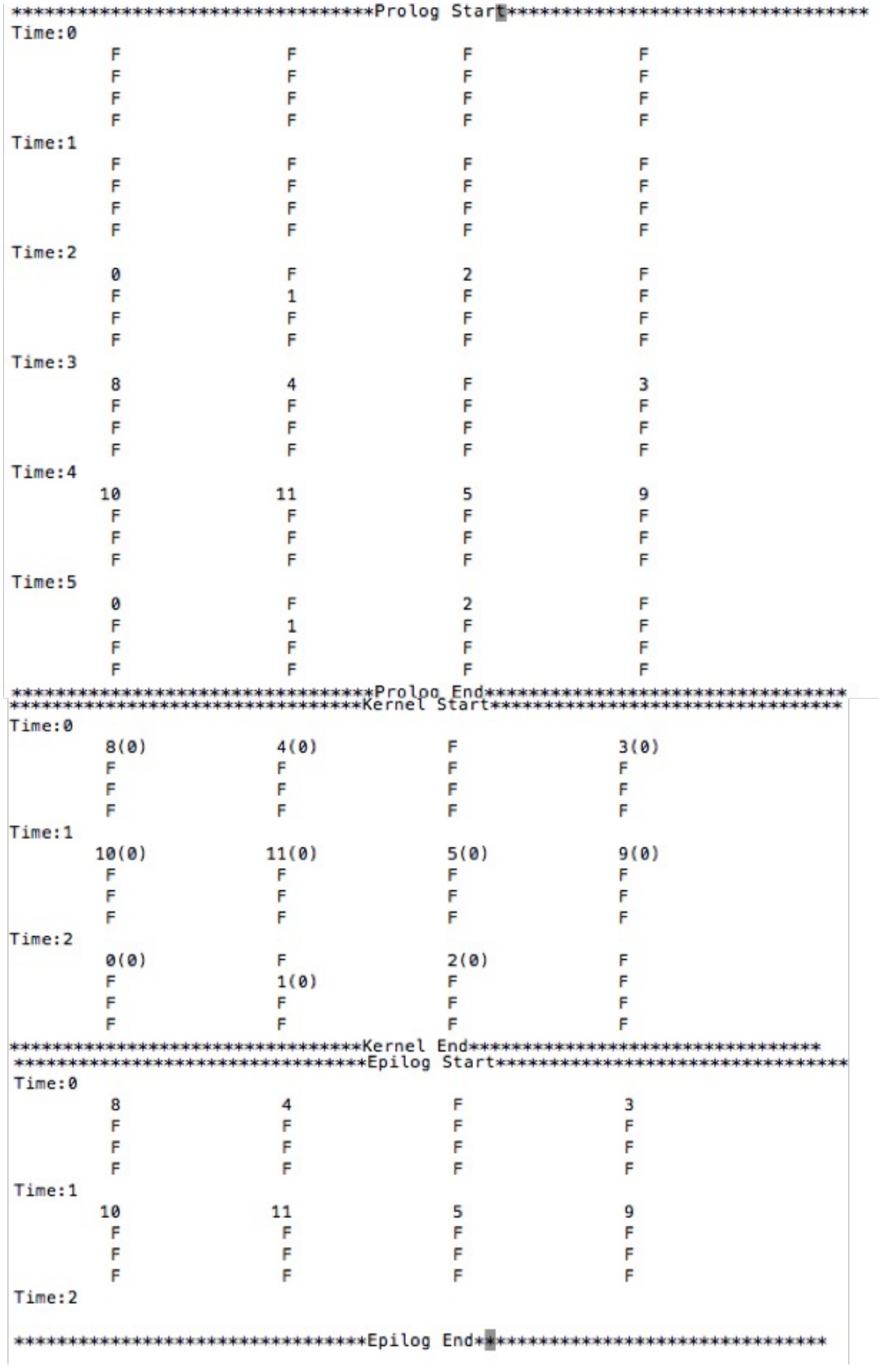

FIGURE 3.5: Mapping by RAMP for the Fibonacci loop.

```
*******PROLOG*********
0: 10e004000
1: 10e004000
2: 10e004000
3: 10e004000
4: 10e004000
5: 10e004000
6: 10e004000
7: 10e004000
8: 10e004000
9: 10e004000
10: 10e004000
11: 10e004000
12: 10e004000
13: 10e004000
14: 10e004000
15: 10e004000
16: 11e00c001
17: 11e00c644
```

FIGURE 3.6: Sample Prolog CGRA machine code.

3.5 Generating CGRA Machine Instructions

This phase generates the machine instructions to configure PEs, load live values into CGRA registers during prologue, and store live-out data during epilogue. The current CGRA instruction-set architecture (ISA) supports two different formats, including support for byte-level memory accesses.

Figure 3.6 shows a snippet of the output file describing the decoded CGRA instructions All of the machine instructions and their breakdowns are shown in this file. The directory `CGRAExec/L1` also contains the binaries corresponding to the prologue, kernel, and epilogue.

3.5.1 CGRA Instruction Formats

The CGRA Instructions are generated based on two formats: R-Type (Regular) and P-Type. Instruction formats are decided based on virtual opcodes or based on the desired functionality of the instruction. For example, most of the operations generate R-Type instructions. However, for pre-loading live values, setting RF configurations during prologue, or doing predicated execution for a conditional statement, a P-Type is generated.

31:28	27	26:24	23:21	20:19	18:17	16:15	14	13	12	11:0
Opcode	P	LMux	RMux	R1	R2	RW	WE	AB	DB	Immediate

TABLE 3.3: R-Type Instruction Format

Table 3.3 breaks down the R-Type instruction. Below are details for each component of the instruction:

- The field Opcode defines the functionality performed by the PE
- P determines the instruction format. If P=1, the instruction is decoded as a P-Type
- LMux indicates the input source for the left multiplexer of the PE
- RMux indicates the input source for the right multiplexer of the PE

- R1 indicates the register number for input1 if LMux indicates the register file as the source
- R2 indicates the register number for input1 if RMux indicates the register file as the source
- RW indicates the register number of the register file where result should be written to
- WE determines whether the PE should write the result back to the register file
- AB indicates that the PE will assert address bus for memory access
- DB indicates that the PE will assert data bus for memory access
- Immediate defines a static constant value, which can be supplied to the PE

31:28	27	26:24	23:21	20:19	18:17	16:15	14:12	11:0
Opcode	P	LMux	RMux	R1	R2	RP	PMux	Immediate

TABLE 3.4: P-type Instruction Format

Table 3.4 breaks down the P-Type instruction. Below are details for each component of the instruction:

- PMux indicates the input source for the predicated multiplexer
- RP indicates the register number for input3 if PMux indicates the register file as the source

Table 3.5 defines the various source inputs for the multiplexers in the PEs.

	Source
0	Register
1	Left Neighboring PE
2	Right Neighboring PE
3	Upper Neighboring PE
4	Bottom Neighboring PE
5	Data Bus
6	Immediate
7	Self (Output Latch)

TABLE 3.5: Input Sources for Multiplexers in a PE

Table 3.6 summarizes the instructions performed by the PEs. PEs can perform fixed-point signed arithmetic, floating-point, logical, and memory operations with 1-cycle latency. For future work, we will allow instructions to take a variable length.

Instruction Format	Opcode	Machine Opcode	Virtual Opcode s	Note
R-Type	0	Add	add ld_data st_data route	
	1	Sub	sub	
	2	Mult	mult	
	3	AND	andop	
	4	OR	orop	
	5	XOR	xorop	
	6	cgraASR	shiftr	
	7	NOP	-	
	8	cgraASL	shiftl	
	9	Div	div	
	10	Rem	rem	
	11	LSHR	shiftr_logical	
	12	EQ	cmpEQ	
	13	NEQ	cmpNEQ	
	14	GT	cmpSGT cmpSGEQ cmpUGT cmpUGEQ	
	15	LT	cmpSLT cmpSLEQ cmpULT cmpULEQ	
P-Type	0	setConfigBoundary	-	
	1	LDI	select	Pre-load Live values in CGRA Registers
	2	LDMI	-	
	3	LDUI	-	
	4	sel	cond_select	Predicated Execution
	5	loopexit	loopctrl	
	6	address_generator	ld_add st_add	
	7	NOP	-	
	8	signExtend	sext	
	9-15	Reserved	-	

Table 3.6: Translation of Machine Opcode from Virtual Opcode

References

[1] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7 (3):21, 2014.

[2] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. RAMP: resource-aware mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018.

[3] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[4] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[5] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020. URL https://arxiv.org/abs/2007.03152.

[6] B Ramakrishna Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1), 1996.