A CGRA Front-end Compiler Enabling Extraction of General Control and Dedicated Operators

Authors: Xuchen Gao, Yunhui Qiu, Yuan Dai, Wenbo Yin, Lingli Wang*

Presenter: Xuchen Gao
Institution: Fudan University, China
Email: xcgao22@m.fudan.edu.cn

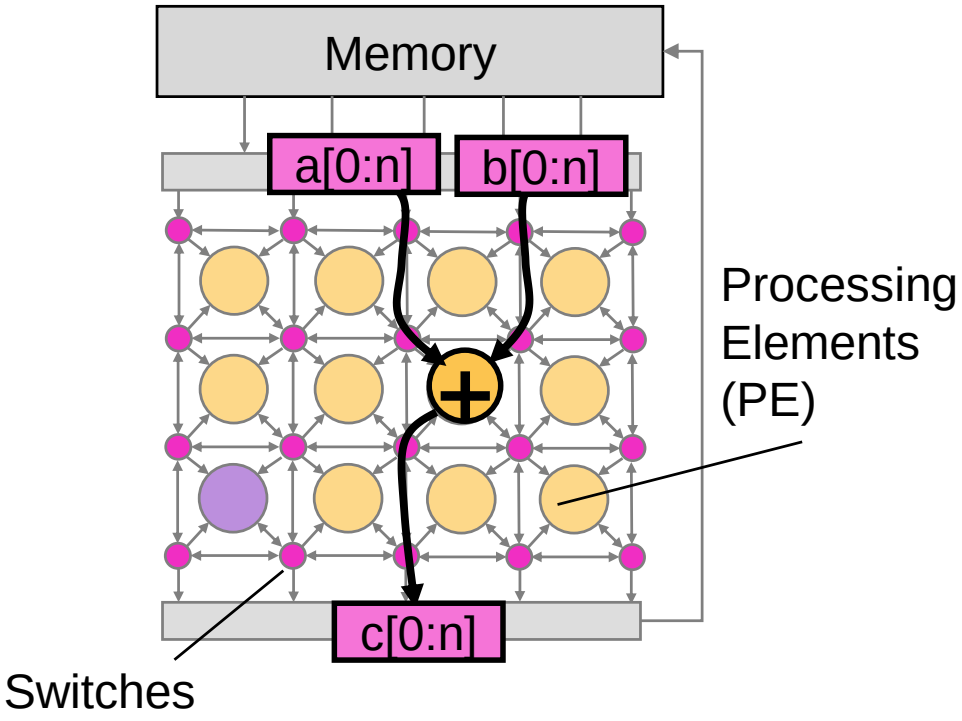
Contents

- Introduction
- Framework of our work
- Analyzing & transforming passes
- SoC runtime configuration of CGRA
- Experiment
- Conclusion

Contents

Background & Motivation

- Coarse-Grained Reconfigurable Architecture (CGRA)



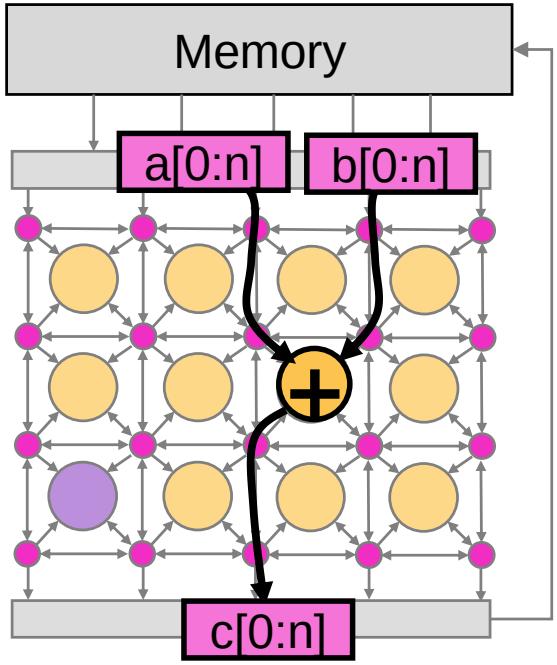
An example of CGRA

Background & Motivation

- Application deploying is a challenge to CGRA

```
for (int i = 0; i < n; ++i)
    c[i] = a[i] + b[i];
```

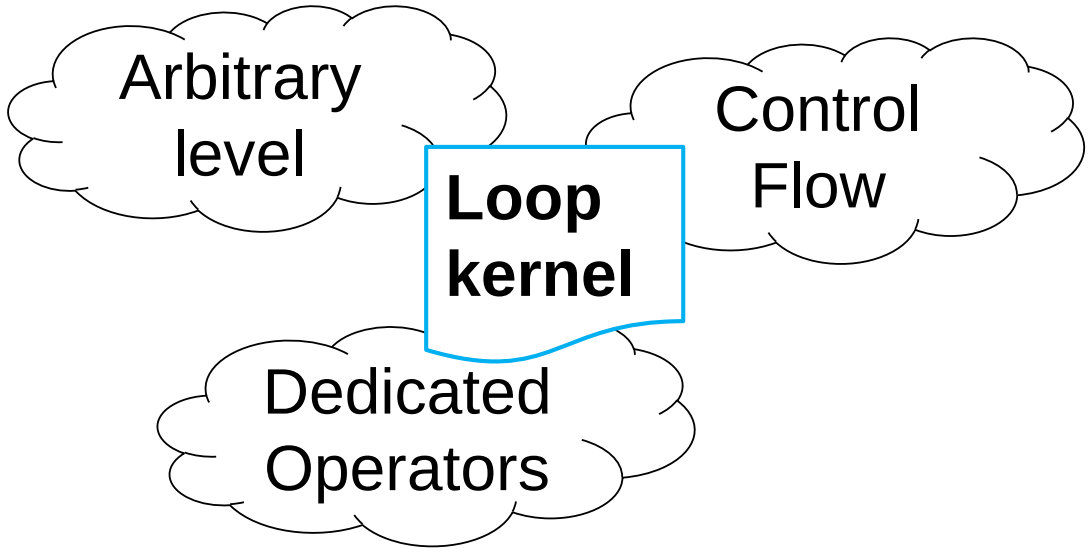
application code



An example of CGRA

Background & Motivation

- What is needed for **A FRONT-END COMPILER**?
  - Joint between application codes and target hardware
  - To cover a wide category of applications
  - To support many useful analyses and extraction of the loop kernel

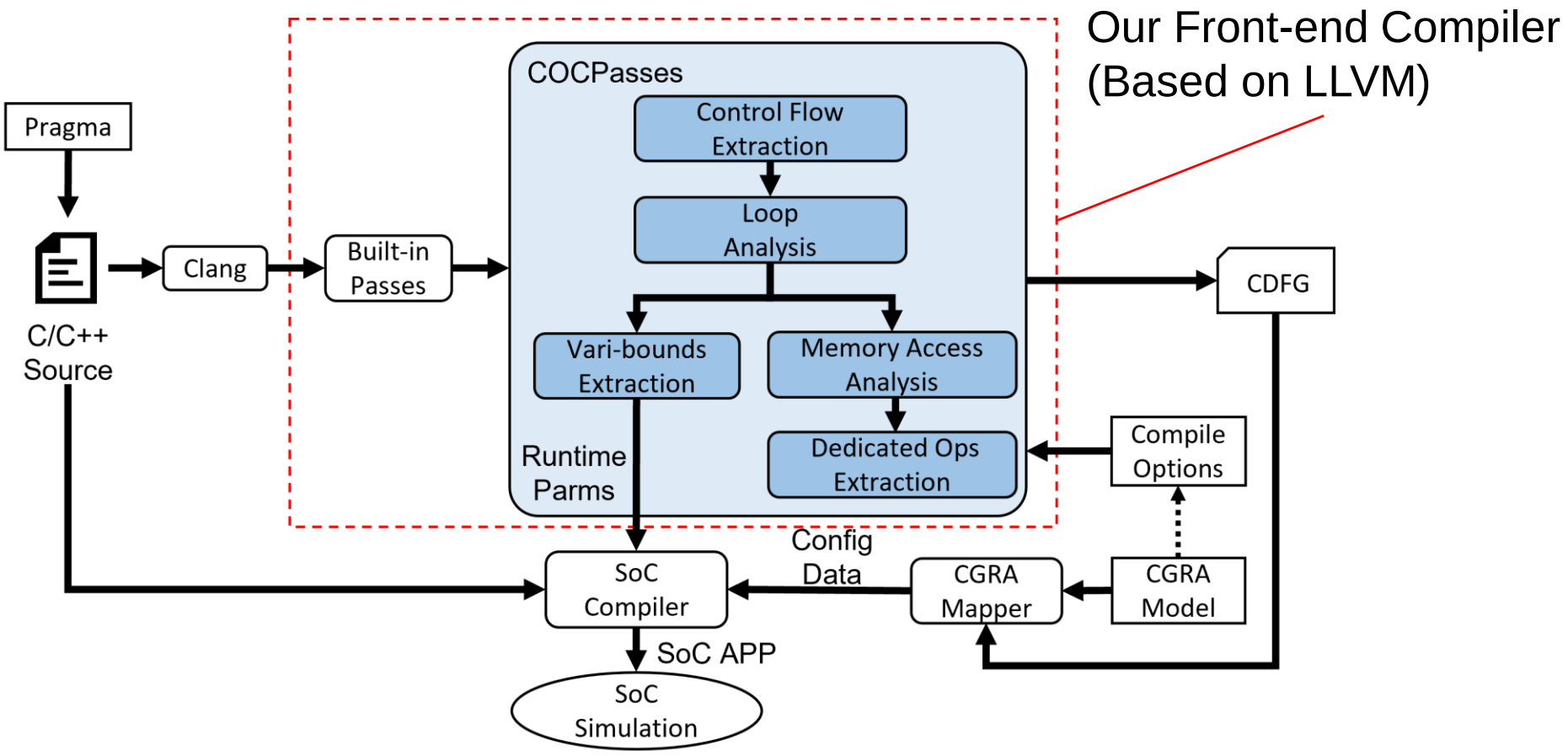


Contents

- Introduction
- Framework of our work
- Analyzing & transforming passes
- SoC runtime configuration of CGRA
- Experiment
- Conclusion

Overview

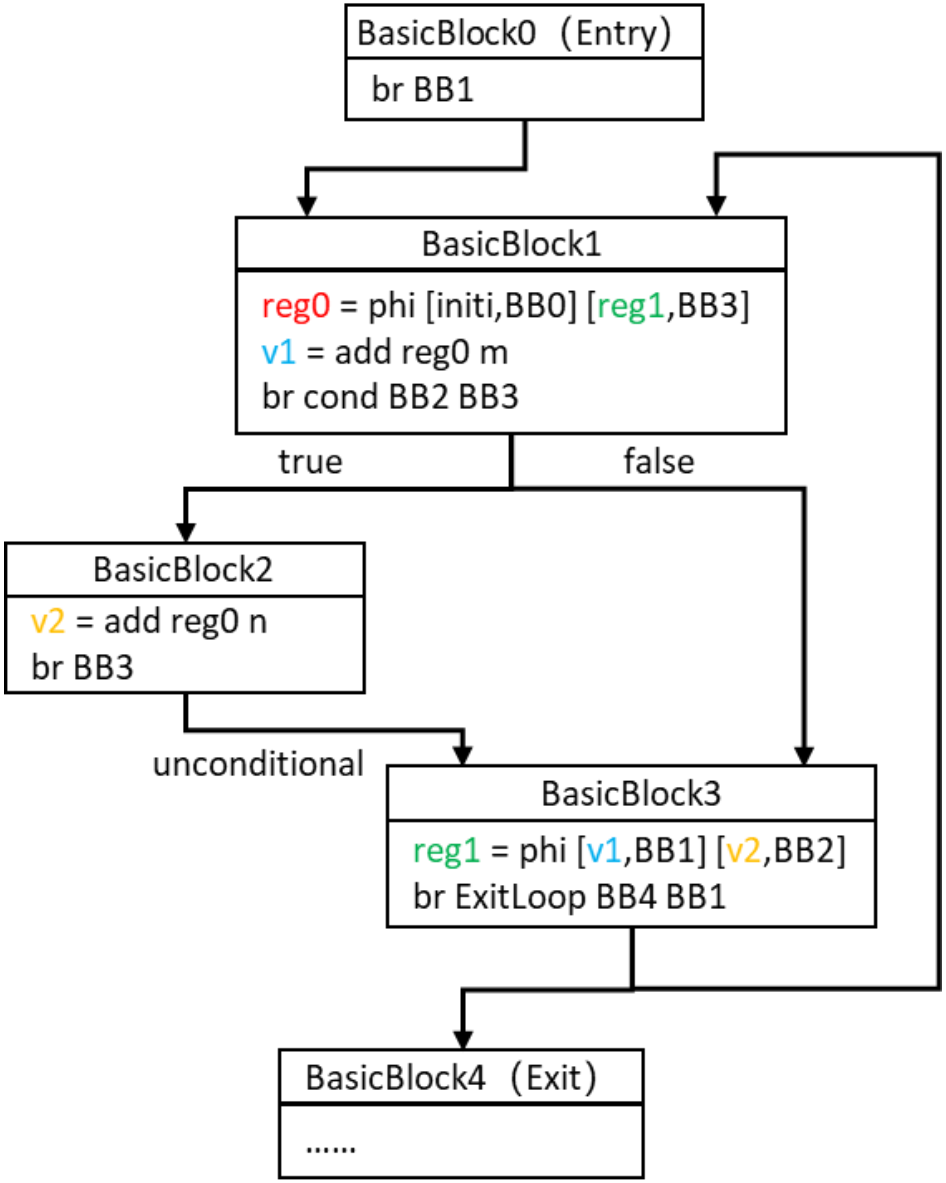
- LLVM: an open-source compilation framework



Contents

- Introduction
- Framework of our work
- Analyzing & transforming passes
- SoC runtime configuration of CGRA
- Experiment
- Conclusion

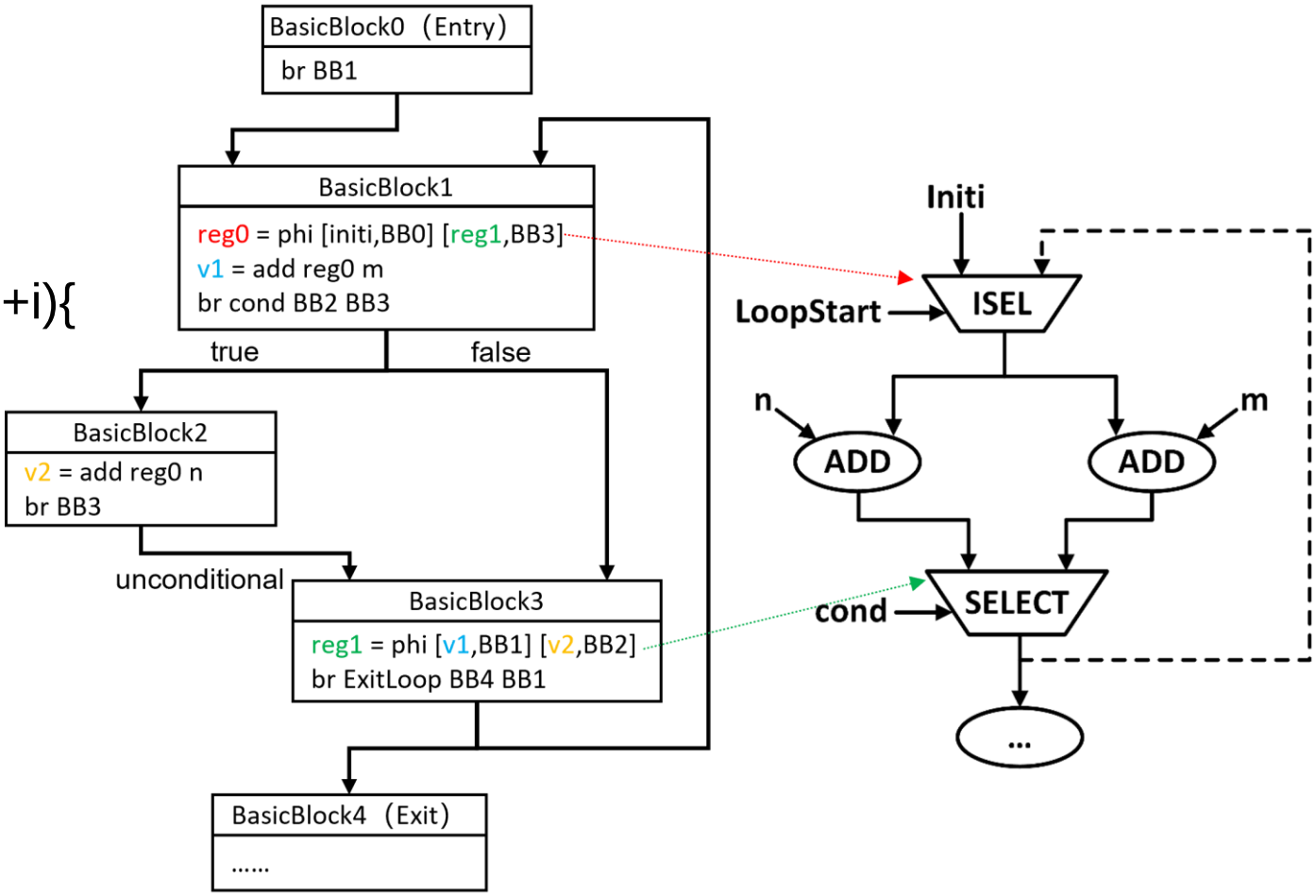
Control flow extraction



- Control flows in LLVM IR
  - *Branch*
  - *Phi instruction*
- Extraction
  - Describe the effects by **SELECT** nodes
  - Analyze accurate control signal according to the **dominator tree (DT)**

Control flow extraction

```
int reg = initi;
for (int i = 0; i < N; ++i){
    if(cond) {
        reg += n;
    } else{
        reg += m;
    }
}
…
```



An example of extracting control flow from LLVM IR

Linear Memory Access Pattern

- Memory is accessed linearly in many applications, so many studies have proposed specific I/O units

```
int A[7][10];
...
for(int i=1; i<6; i=i+2){
    for(int j=2; j>=0; j--){
        .....
        int access = A[i+1][2*j+i];
        .....
    }   //level 1
}   //level 2
```

A code segment of linear memory access

Linear Memory Access Pattern

- *N*-dimension memory access in an *M*-level loop (where values of *N* and *M* are both arbitrary)
- Linear memory access:
  - The step value of each loop index is loop-invariant
  - The bounds of each loop index are loop-invariant
  - Memory address is the affine transformation of loop indices

Linear Memory Access Pattern

- *N*-dimension memory access in an *M*-level loop (where values of *N* and *M* are both arbitrary)
- Linear memory access:

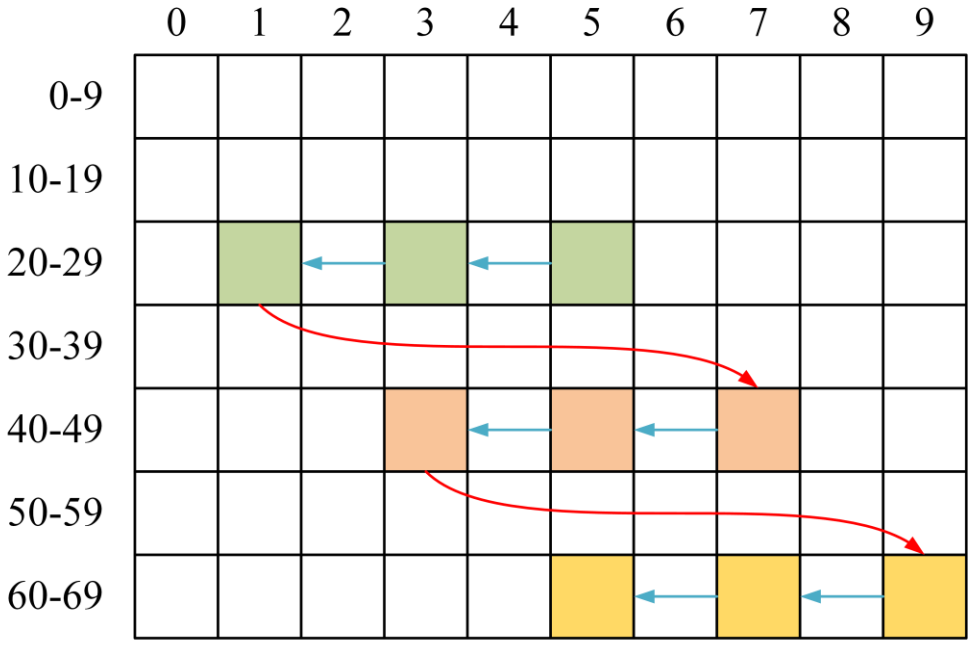
```
int A[7][10];
...
for(int i=1; i<6; i=i+2){
    for(int j=2; j>=0; j--){
        .....
        int access = A[i+1][2*j+i];
        .....
    }  //level 1
}  //level 2
```

$$\begin{cases} i \in [1,5], step_i = 2 \\ j \in [0,2], step_j = -1 \\ AddrIdx = 11i + 2j + 10 \end{cases}$$

Linear Memory Access Pattern

```
int A[7][10];
...
for(int i=1; i<6; i=i+2){
    for(int j=2; j>=0; j--){
        .....
        int access = A[i+1][2*j+i];
        .....
    }  //level 1
}  //level 2
```

start	Loop Level 1		Loop Level 2	
	stride	count	stride	count
25	-2	3	26	3



An example of linear memory access

Dedicated Operator Extraction

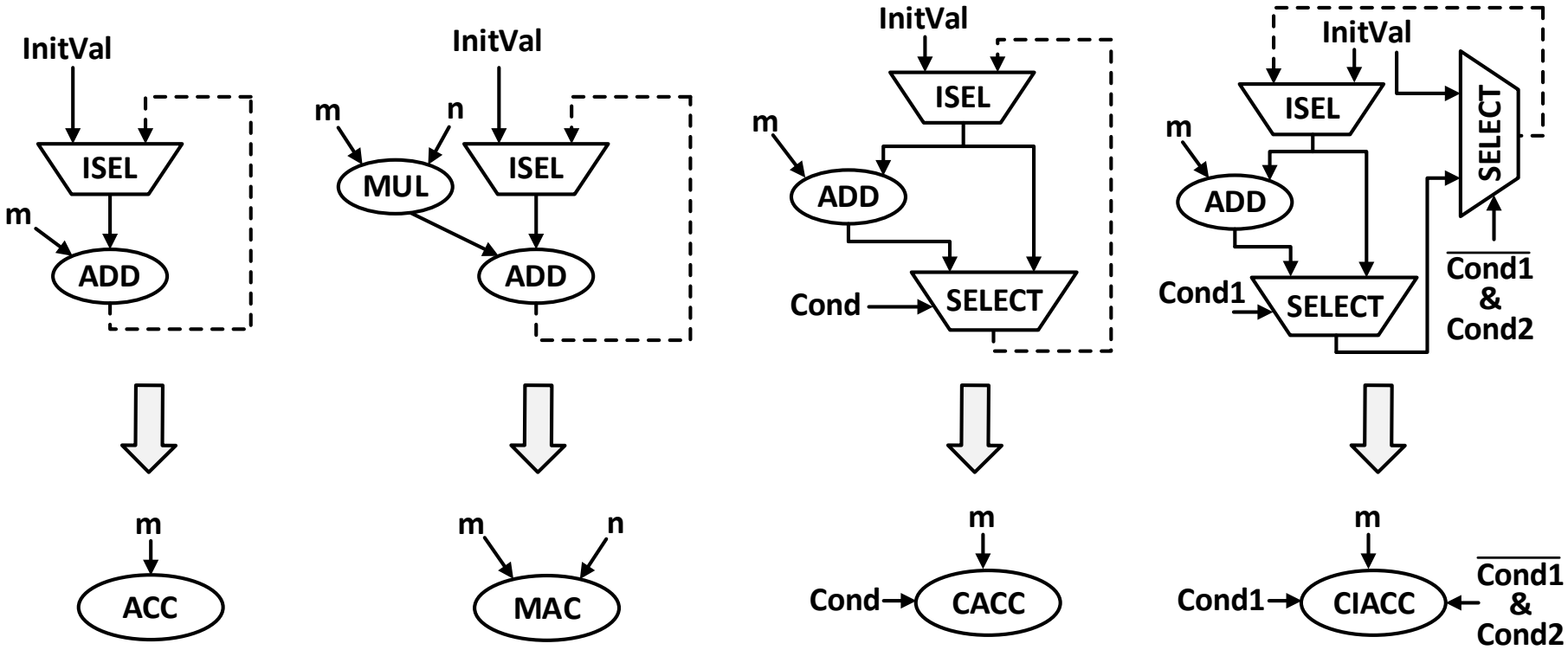
- Dedicated operators are important to enhance the flexibility and performance of CGRA
- We extracted dedicated operators on the CDFG level, including:
  - Accumulation series operators
  - Conditional memory access
  - Linear memory access

Dedicated Operator Extraction

- Accumulation series operators
  - Problems of cycles in a CDFG
    - Degrade CGRA performance
    - Increase Compilation difficulty
  - Solutions
    - Hardware: Enhanced PEs
    - **Compiler: Dedicated extraction**

Dedicated Operator Extraction

- Accumulation series operators



Transformations of ACC-series nodes

Dedicated Operator Extraction

- Conditional memory access

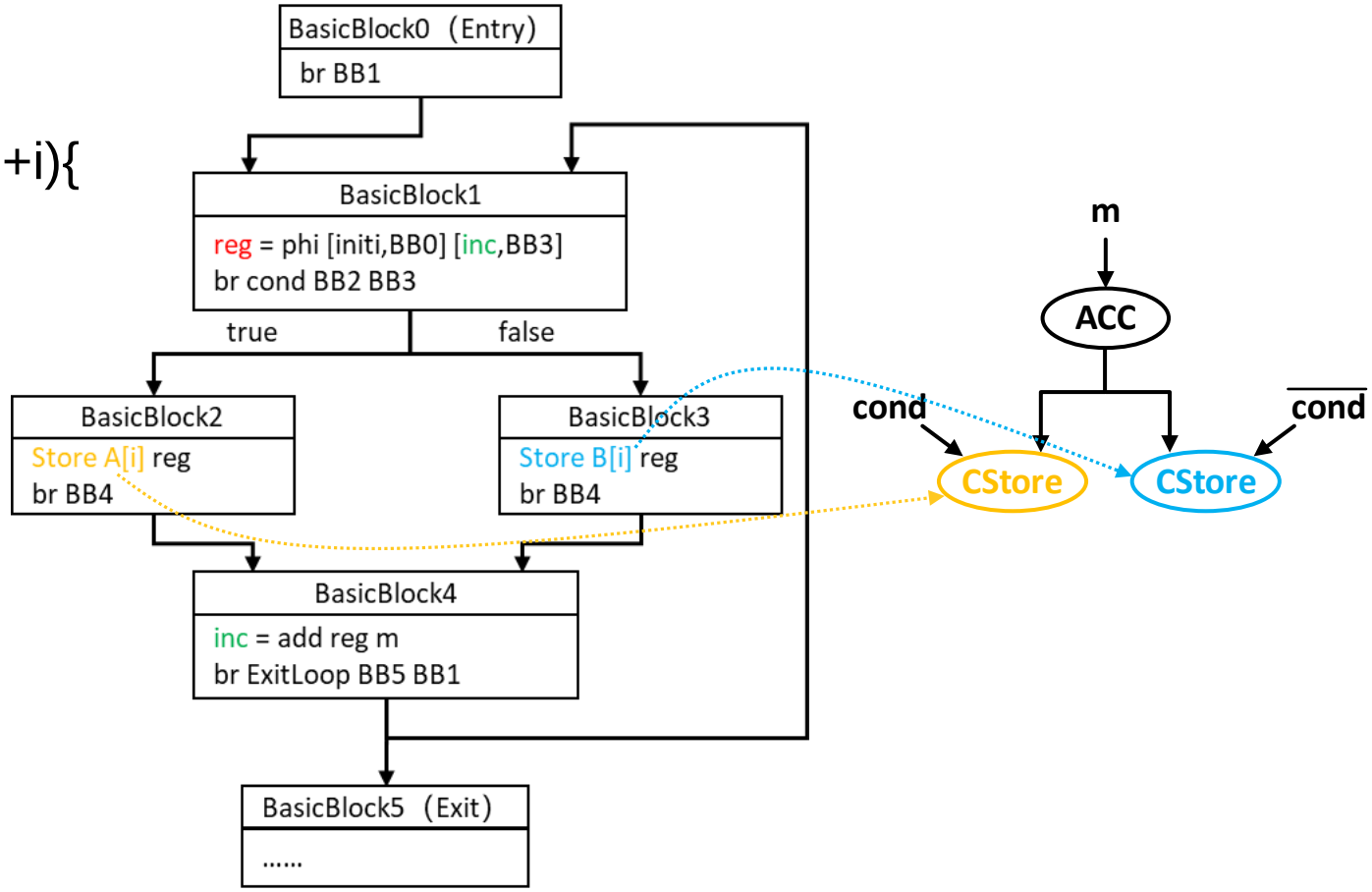
```
int reg = initi;
for (int i = 0; i < N; ++i){
    if(cond) {
        A[i] = reg;
    } else{
        B[i] = reg;
    }
    reg += m;
}
…
```

A code segment of conditional memory access

Dedicated Operator Extraction

- Conditional memory access

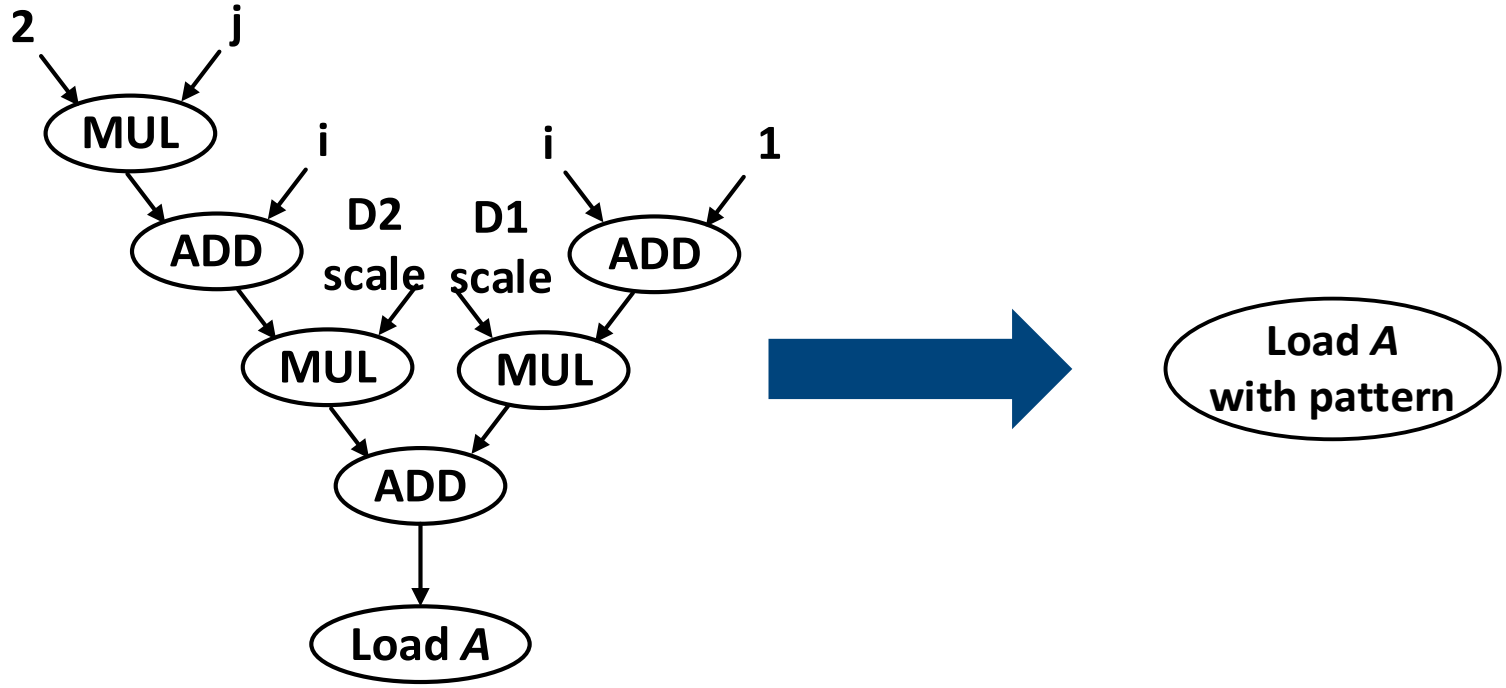
```
int reg = initi;
for (int i = 0; i < N; ++i){
    if(cond) {
        A[i] = reg;
    } else{
        B[i] = reg;
    }
    reg += m;
}
…
```



Dedicated Operator Extraction

- Linear memory access

Original Multi-Add tree for address calculation



Effection of **computing resources saving**

Contents

- Introduction
- Framework of our work
- Analyzing & transforming passes
- SoC runtime configuration of CGRA
- Experiment
- Conclusion

Target issue

- Many applications have **loop-invariant** variables in their kernel, but the value will be settled at runtime

```
int A[10][20];
int B[10][20];
.....
kernel(int arg0, int arg1){
    int i, j;
    for(i=0; i<arg0; i++){
        for(j=arg1; j>=0; j--){
            B[i][j] = 2 * A[i][j];
        }
    }
}
```

Kernel loop

Methodology

- Treat the input loop-invariant variables as the constants and generate the CDFG
- Maintain these variables in related parameters lazily
- Run compilation flow and generate static CGRA configuration
- Insert substitution instructions in SoC program

Methodology

Access Pattern	Stride1	Count1	Stride2	Count2
	-4	1+arg0	20*arg1+4*(1+arg0)	arg1
ACC *i*	InitiValue	Count	Interval	Repeat
	0	arg1	arg0+1	1
ACC *j*	InitiValue	Count	Interval	Repeat
	arg0	1+arg0	1	arg1

Parameters in the example

```
int A[10][20];
int B[10][20];
.....
kernel(int arg0, int arg1){
   int i, j;
   for(i=0; i<arg0; i++){
      for(j=arg1; j>=0; j--){
         B[i][j] = 2 * A[i][j];
      }
   }
}
```

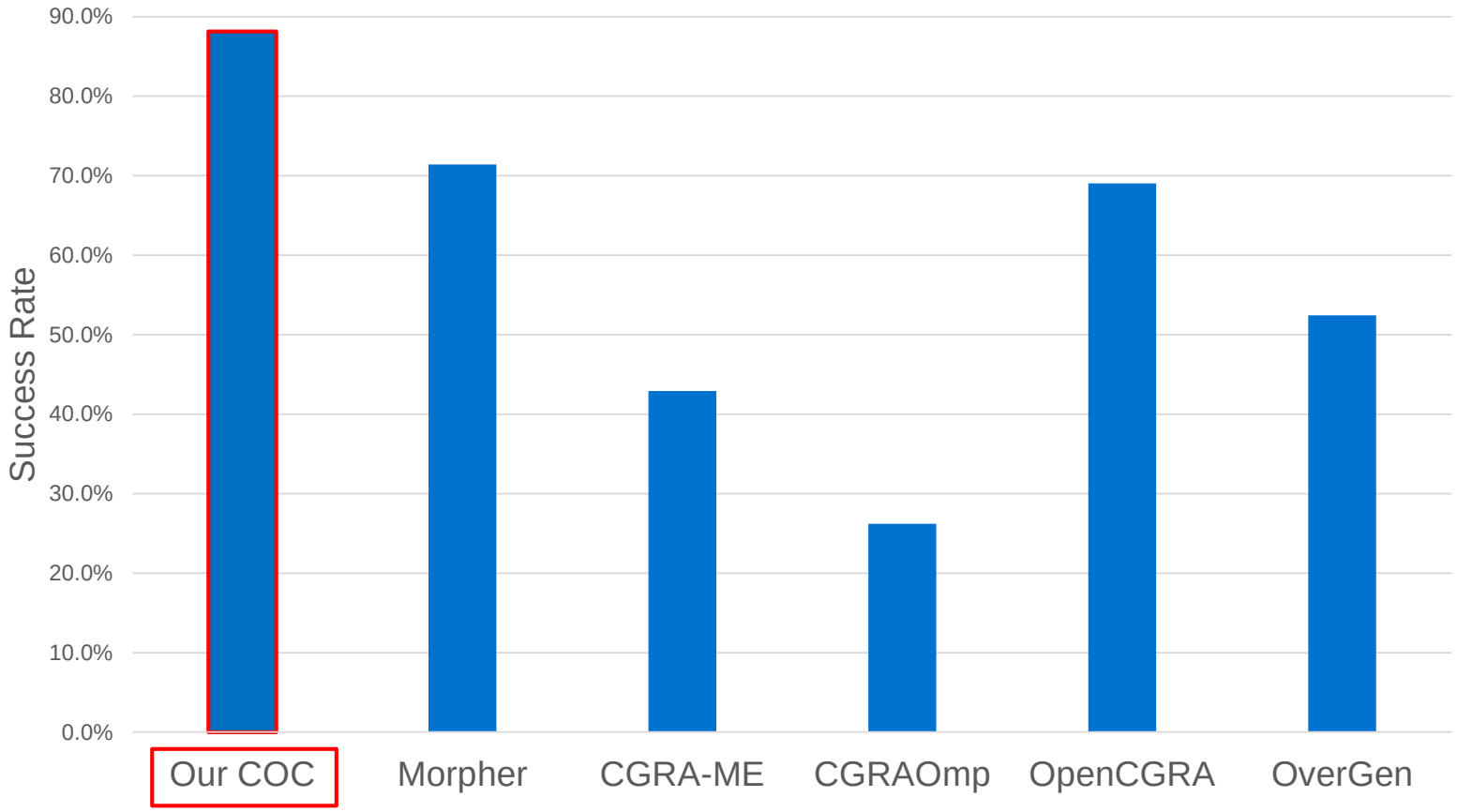
```
int A[10][20];
int B[10][20];
.....
kernel(int arg0, int arg1){
   replaceConfig(Config, arg0, arg1);
   ConfigCGRA(Config);
   data_in(A);
   cgra_execute();
   data_out(B);
}
```

An example of runtime configuration

Contents

- Introduction
- Framework of our work
- Analyzing & transforming passes
- SoC runtime configuration of CGRA
- Experiment
- Conclusion

Success Rates in CDFG Generation



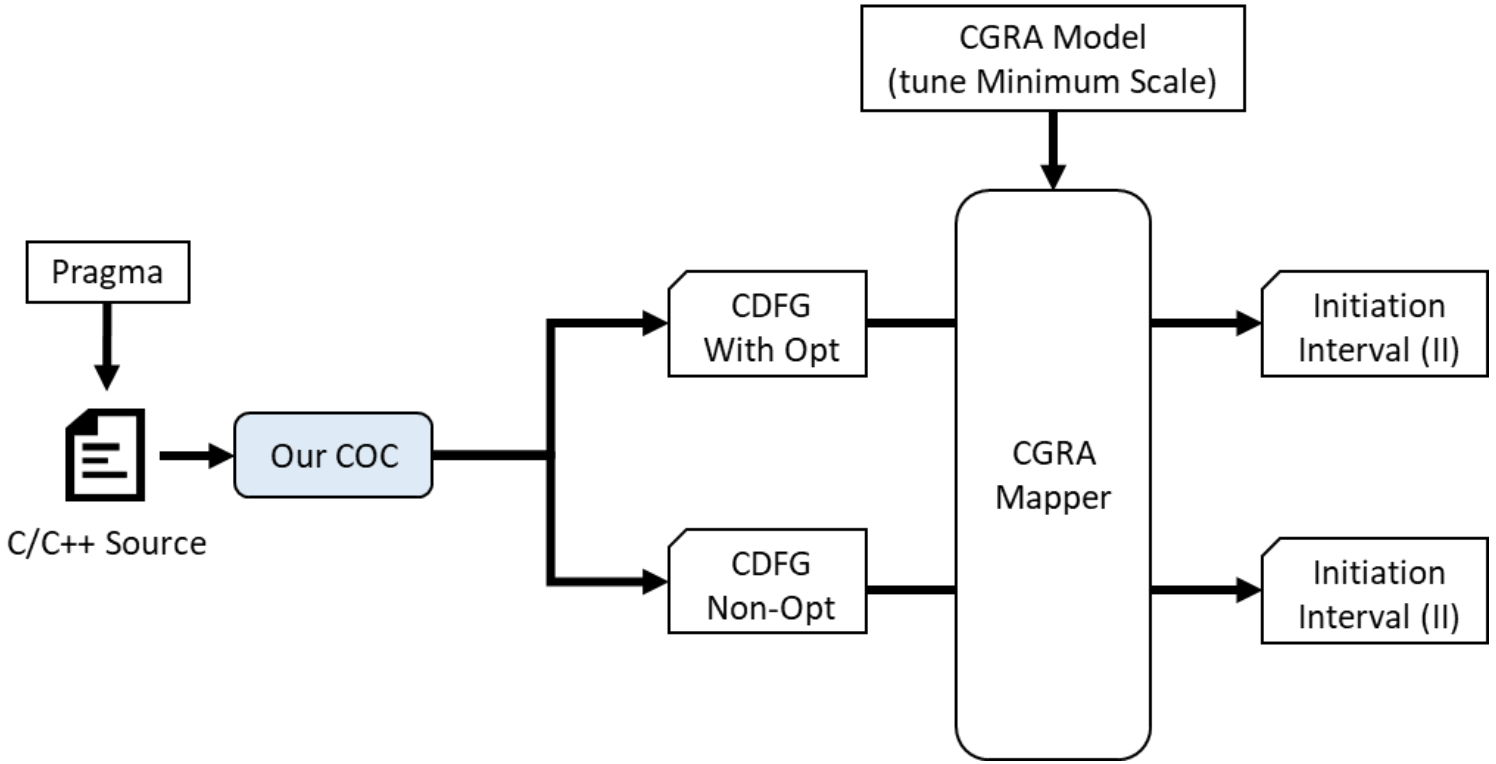
Comparison of Characteristics

Compiler	Access Pattern	General Control	Accumulative Operators	Imperfect Statements	Variable Bounds
Our COC	✓	✓	ACC-series	✓	✓
Morpher	×	✓	×	DFG I/O[5]	×
CGRA-ME	×	×	×	DFG I/O[5]	×
CGRAOmp	✓[1]	×	×	×	×
OpenCGRA	×	✓[2]	×	×	×
OverGen	✓[3]	×	ACC[4]	DFG I/O[5]	×

1. CGRAOmp **only** handles linear memory accesses
2. OpenCGRA does not handle *phi instructions* in front-end compiler
3. OverGen decouples the DFG with memory accesses, so there is no clear access information
4. OverGen's ACC execution pattern is not clear
5. They consider imperfect statements as DFG inputs/outputs of the innermost loop

Minimum II and Utilization

- Self evaluation



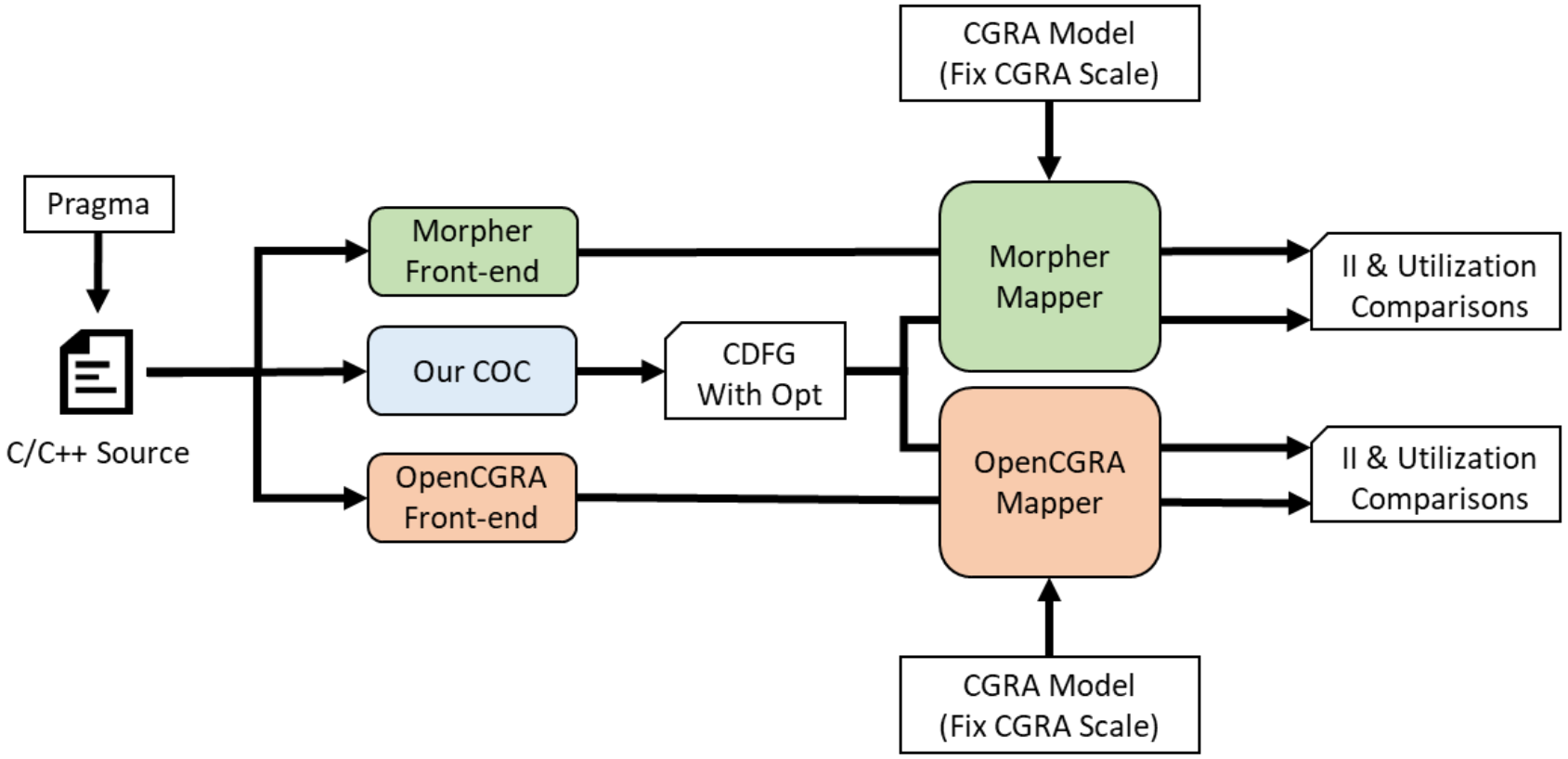
Minimum II and Utilization

- Self evaluation

Benchmark	Node Num.		II		Min. Scale	
	Opt.	Non-Opt.	Opt.	Non-Opt.	Opt.	Non-Opt.
conv2d_3x3	27	56	1	2	5×4 (0.42)	6×8 (1)
fir	19	36	1	7	5×4 (0.67)	5×6 (1)
gemm	17	50	1	10	5×4 (0.37)	6×9 (1)
mvt	20	44	1	10	5×4 (0.50)	5×8 (1)
matching	9	15	1	3	3×4 (0.48)	5×5 (1)
deinterleaving	19	24	1	4	4×6 (0.96)	5×5 (1)
Average	**18.5**	**37.5**	**1**	**6.33**	**0.57**	**1**

Minimum II and Utilization

- Compare with other front-end compilers



D. Wijerathne *et al.*, “Morpher: An open-source integrated compilation and simulation framework for cgra,” in Fifth Workshop on Open-Source EDA Technology (WOSET).

C. Tan *et al.*, “Opencgra: An opensource unified framework for modeling, testing, and evaluating cgras,” in 2020 IEEE 38th International Conference on Computer Design (ICCD), pp. 381–388.

Minimum II and Utilization

- Compare with other front-end compilers

(a) Morpher Mapper

Benchmark	Minimum II		Spatio-Temporal Utilization	
	COC	Morpher	COC	Morpher
Conv2d_3x3	1	4	90% (1.08)	83% (1)
Gemm	1	7	85% (2.24)	38% (1)
Jacobi-2d	1	4	92% (1.02)	90% (1)
Pedometer	1	4	94% (1.16)	81% (1)
Stencil3d	1	4	90% (1.15)	78% (1)
Average	**1**	**4.6**	**1.33**	**1**

(b) OpenCGRA Mapper

Benchmark	Minimum II		Spatio-Temporal Utilization	
	COC	OpenCGRA	COC	OpenCGRA
Conv2d_3x3	1	4	77% (1.18)	65% (1)
Gemm	1	6	85% (4.72)	18% (1)
Jacobi-2d	1	4	73% (1.92)	38% (1)
Pedometer	1	4	75% (1.15)	65% (1)
Stencil3d	1	4	72% (1.33)	54% (1)
Average	**1**	**4.4**	**2.06**	**1**

Contents

- Introduction
- Framework of our work
- Analyzing & transforming passes
- SoC runtime configuration of CGRA
- Experiment
- Conclusion

Conclusion

- An open-source front-end compiler for CGRA (from C/C++ to CDFG)
- Supporting general control statements, nested loop with arbitrary levels, and imperfect statements
- Analysis of multi-dimension memory access & extraction of various dedicated operators
- A methodology to handle kernels with variable parameters for the SoC runtime configuration of CGRA

Thanks

Xuchen Gao, Yunhui Qiu, Yuan Dai, Wenbo Yin, Lingli Wang*

State Key Laboratory of Integrated Chips and Systems
Fudan University, Shanghai, China
Email: xcgao22@m.fudan.edu.cn, *llwang@fudan.edu.cn