

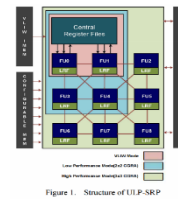
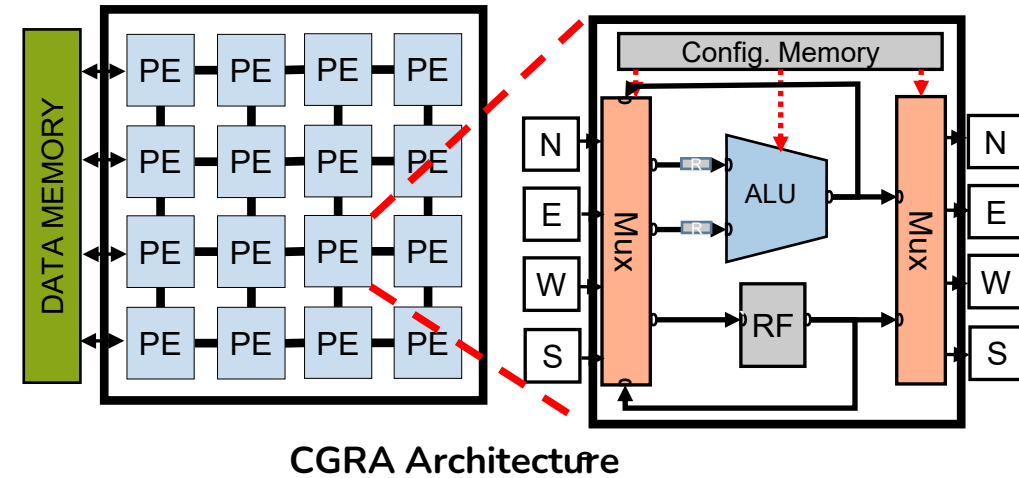
Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA

Dhananjaya Wijerathne*, Zhaoying Li*, Manupa Karunaratne, Li-Shiuan Peh*, Tulika Mitra*
School of Computing, National University of Singapore*, Advanced Micro Devices, Inc
{dmd, zhaoying, peh, tulika}@comp.nus.edu.sg, Manupa.Karunaratne@amd.com

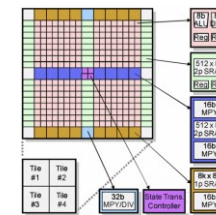


CGRA: Coarse Grained Reconfigurable Array

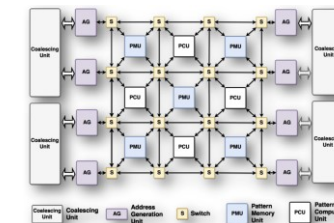
- Power-efficient reconfigurable accelerator
 - Word level reconfigurable
- Simple hardware architecture
 - Array of Processing Elements (PE) interconnected through a reconfigurable network
 - Each PE has ALU, register file, multiplexers and configuration memory
- Fully software controlled
 - Compiler statically generates the configurations
- Commercially available



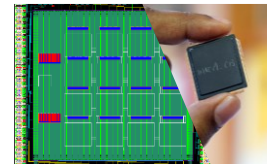
Samsung Reconfigurable Processor [FPT '12]



Renesas DRP



Sambanova Plasticine [ISCA'17]



NUS HyCUBE
[DAC '17, A-SSCC' 19]

Application mapping on CGRA

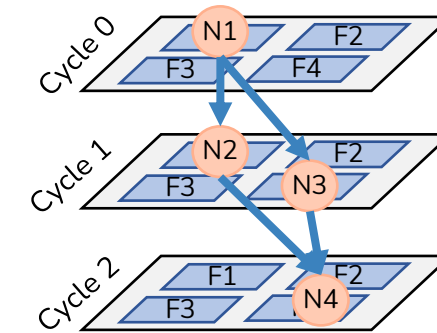
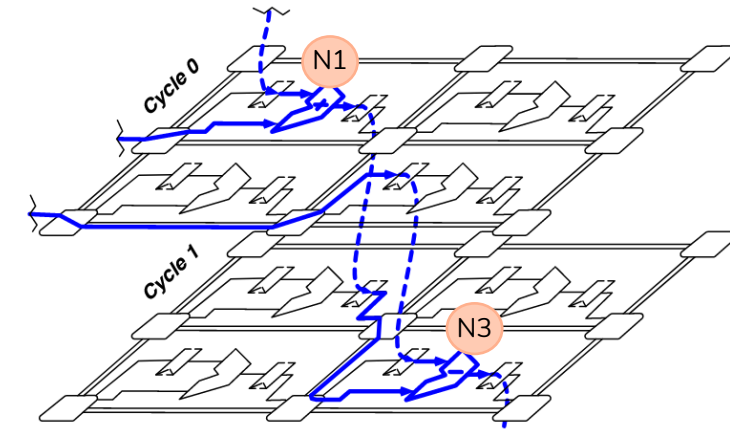
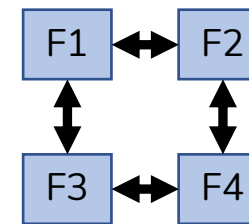
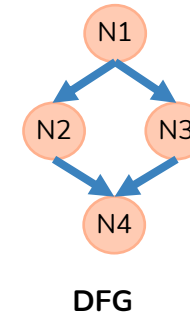
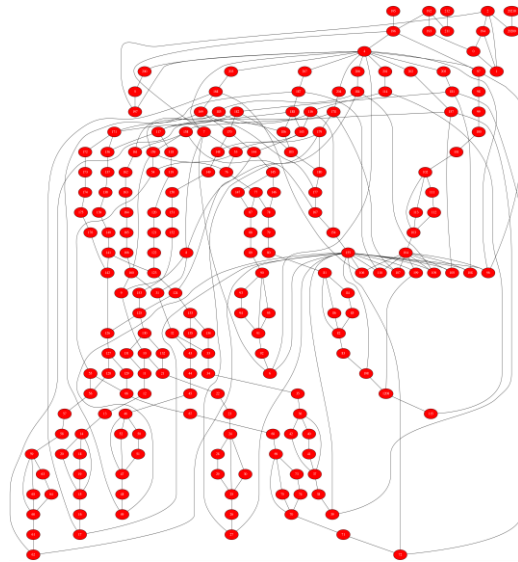
- Target: a loop kernel from applications
- Mapping the dataflow graph (DFG) of the loop body on to the CGRA
 - Placement: assigning DFG operations to ALUs
 - Routing: mapping data signals using wires and registers

```

uint16_t word;
uint16_t result;

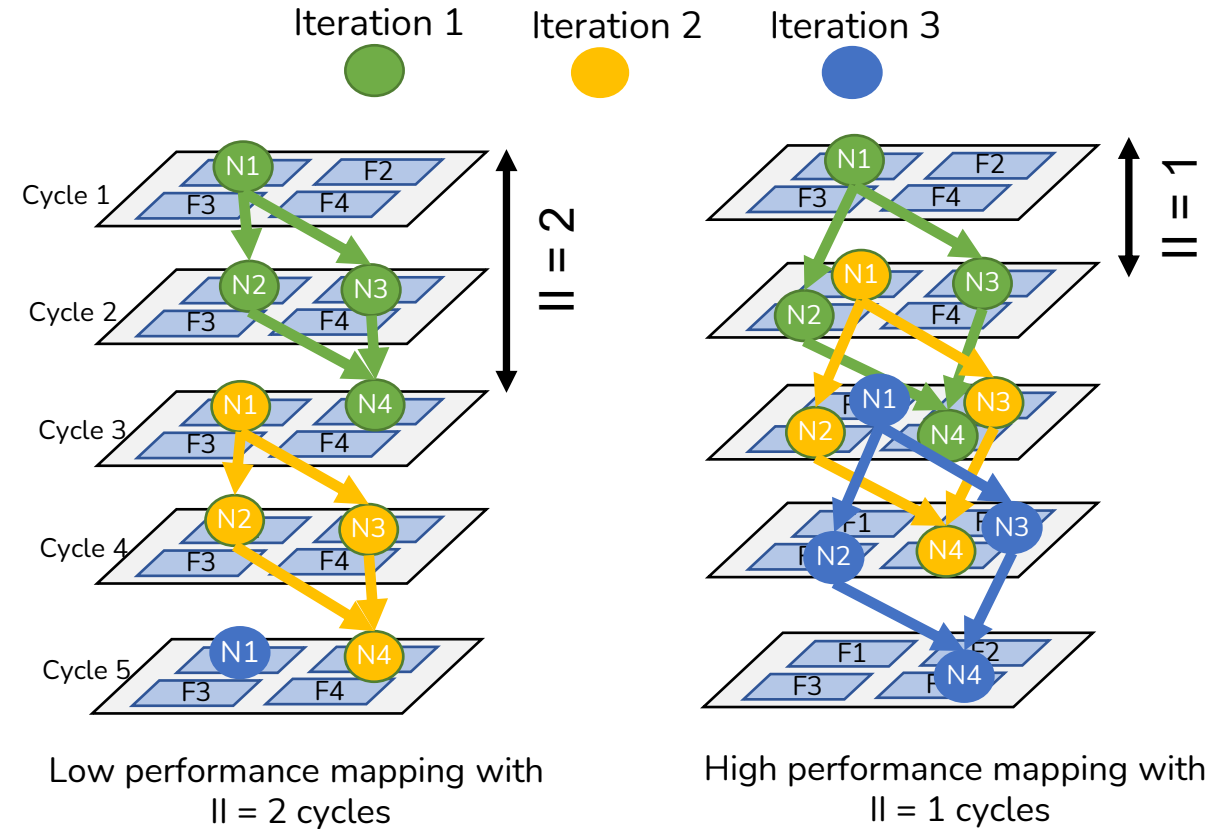
uint16_t i;
uint16_t pdcrc = 0;

for ( i = 0; i < N; i++)
{
    #pragma omp parallel
    #pragma omp num_threads(1)
    {
        #pragma omp for
        for ( j = 0; j < 16; j++)
        {
            // Short circuit the 16 DCT 1/4 only the DC component is non-zero
            if ( j == 0 )
            {
                #pragma omp critical
                {
                    pdcrc = pdcrc + word;
                }
            }
            else
            {
                uint16_t wrcr = pdcrc + 0;
                uint16_t wrcr1 = wrcr + 0;
                uint16_t wcr = wrcr1 + wrcr;
                uint16_t wcr1 = wcr + wrcr;
                uint16_t wcr2 = wcr1 + wcr;
                uint16_t wcr3 = wcr2 + wcr1;
                uint16_t wcr4 = wcr3 + wcr2;
                uint16_t wcr5 = wcr4 + wcr3;
                uint16_t wcr6 = wcr5 + wcr4;
                uint16_t wcr7 = wcr6 + wcr5;
                uint16_t wcr8 = wcr7 + wcr6;
                uint16_t wcr9 = wcr8 + wcr7;
                uint16_t wcr10 = wcr9 + wcr8;
                uint16_t wcr11 = wcr10 + wcr9;
                uint16_t wcr12 = wcr11 + wcr10;
                uint16_t wcr13 = wcr12 + wcr11;
                uint16_t wcr14 = wcr13 + wcr12;
                uint16_t wcr15 = wcr14 + wcr13;
                uint16_t wcr16 = wcr15 + wcr14;
                uint16_t wcr17 = wcr16 + wcr15;
                uint16_t wcr18 = wcr17 + wcr16;
                uint16_t wcr19 = wcr18 + wcr17;
                uint16_t wcr20 = wcr19 + wcr18;
                uint16_t wcr21 = wcr20 + wcr19;
                uint16_t wcr22 = wcr21 + wcr20;
                uint16_t wcr23 = wcr22 + wcr21;
                uint16_t wcr24 = wcr23 + wcr22;
                uint16_t wcr25 = wcr24 + wcr23;
                uint16_t wcr26 = wcr25 + wcr24;
                uint16_t wcr27 = wcr26 + wcr25;
                uint16_t wcr28 = wcr27 + wcr26;
                uint16_t wcr29 = wcr28 + wcr27;
                uint16_t wcr30 = wcr29 + wcr28;
                uint16_t wcr31 = wcr30 + wcr29;
                uint16_t wcr32 = wcr31 + wcr30;
                uint16_t wcr33 = wcr32 + wcr31;
                uint16_t wcr34 = wcr33 + wcr32;
                uint16_t wcr35 = wcr34 + wcr33;
                uint16_t wcr36 = wcr35 + wcr34;
                uint16_t wcr37 = wcr36 + wcr35;
                uint16_t wcr38 = wcr37 + wcr36;
                uint16_t wcr39 = wcr38 + wcr37;
                uint16_t wcr40 = wcr39 + wcr38;
                uint16_t wcr41 = wcr40 + wcr39;
                uint16_t wcr42 = wcr41 + wcr40;
                uint16_t wcr43 = wcr42 + wcr41;
                uint16_t wcr44 = wcr43 + wcr42;
                uint16_t wcr45 = wcr44 + wcr43;
                uint16_t wcr46 = wcr45 + wcr44;
                uint16_t wcr47 = wcr46 + wcr45;
                uint16_t wcr48 = wcr47 + wcr46;
                uint16_t wcr49 = wcr48 + wcr47;
                uint16_t wcr50 = wcr49 + wcr48;
                uint16_t wcr51 = wcr50 + wcr49;
                uint16_t wcr52 = wcr51 + wcr50;
                uint16_t wcr53 = wcr52 + wcr51;
                uint16_t wcr54 = wcr53 + wcr52;
                uint16_t wcr55 = wcr54 + wcr53;
                uint16_t wcr56 = wcr55 + wcr54;
                uint16_t wcr57 = wcr56 + wcr55;
                uint16_t wcr58 = wcr57 + wcr56;
                uint16_t wcr59 = wcr58 + wcr57;
                uint16_t wcr60 = wcr59 + wcr58;
                uint16_t wcr61 = wcr60 + wcr59;
                uint16_t wcr62 = wcr61 + wcr60;
                uint16_t wcr63 = wcr62 + wcr61;
                uint16_t wcr64 = wcr63 + wcr62;
                uint16_t wcr65 = wcr64 + wcr63;
                uint16_t wcr66 = wcr65 + wcr64;
                uint16_t wcr67 = wcr66 + wcr65;
                uint16_t wcr68 = wcr67 + wcr66;
                uint16_t wcr69 = wcr68 + wcr67;
                uint16_t wcr70 = wcr69 + wcr68;
                uint16_t wcr71 = wcr70 + wcr69;
                uint16_t wcr72 = wcr71 + wcr70;
                uint16_t wcr73 = wcr72 + wcr71;
                uint16_t wcr74 = wcr73 + wcr72;
                uint16_t wcr75 = wcr74 + wcr73;
                uint16_t wcr76 = wcr75 + wcr74;
                uint16_t wcr77 = wcr76 + wcr75;
                uint16_t wcr78 = wcr77 + wcr76;
                uint16_t wcr79 = wcr78 + wcr77;
                uint16_t wcr80 = wcr79 + wcr78;
                uint16_t wcr81 = wcr80 + wcr79;
                uint16_t wcr82 = wcr81 + wcr80;
                uint16_t wcr83 = wcr82 + wcr81;
                uint16_t wcr84 = wcr83 + wcr82;
                uint16_t wcr85 = wcr84 + wcr83;
                uint16_t wcr86 = wcr85 + wcr84;
                uint16_t wcr87 = wcr86 + wcr85;
                uint16_t wcr88 = wcr87 + wcr86;
                uint16_t wcr89 = wcr88 + wcr87;
                uint16_t wcr90 = wcr89 + wcr88;
                uint16_t wcr91 = wcr90 + wcr89;
                uint16_t wcr92 = wcr91 + wcr90;
                uint16_t wcr93 = wcr92 + wcr91;
                uint16_t wcr94 = wcr93 + wcr92;
                uint16_t wcr95 = wcr94 + wcr93;
                uint16_t wcr96 = wcr95 + wcr94;
                uint16_t wcr97 = wcr96 + wcr95;
                uint16_t wcr98 = wcr97 + wcr96;
                uint16_t wcr99 = wcr98 + wcr97;
                uint16_t wcr100 = wcr99 + wcr98;
                uint16_t wcr101 = wcr100 + wcr99;
                uint16_t wcr102 = wcr101 + wcr100;
                uint16_t wcr103 = wcr102 + wcr101;
                uint16_t wcr104 = wcr103 + wcr102;
                uint16_t wcr105 = wcr104 + wcr103;
                uint16_t wcr106 = wcr105 + wcr104;
                uint16_t wcr107 = wcr106 + wcr105;
                uint16_t wcr108 = wcr107 + wcr106;
                uint16_t wcr109 = wcr108 + wcr107;
                uint16_t wcr110 = wcr109 + wcr108;
                uint16_t wcr111 = wcr110 + wcr109;
                uint16_t wcr112 = wcr111 + wcr110;
                uint16_t wcr113 = wcr112 + wcr111;
                uint16_t wcr114 = wcr113 + wcr112;
                uint16_t wcr115 = wcr114 + wcr113;
                uint16_t wcr116 = wcr115 + wcr114;
                uint16_t wcr117 = wcr116 + wcr115;
                uint16_t wcr118 = wcr117 + wcr116;
                uint16_t wcr119 = wcr118 + wcr117;
                uint16_t wcr120 = wcr119 + wcr118;
                uint16_t wcr121 = wcr120 + wcr119;
                uint16_t wcr122 = wcr121 + wcr120;
                uint16_t wcr123 = wcr122 + wcr121;
                uint16_t wcr124 = wcr123 + wcr122;
                uint16_t wcr125 = wcr124 + wcr123;
                uint16_t wcr126 = wcr125 + wcr124;
                uint16_t wcr127 = wcr126 + wcr125;
                uint16_t wcr128 = wcr127 + wcr126;
                uint16_t wcr129 = wcr128 + wcr127;
                uint16_t wcr130 = wcr129 + wcr128;
                uint16_t wcr131 = wcr130 + wcr129;
                uint16_t wcr132 = wcr131 + wcr130;
                uint16_t wcr133 = wcr132 + wcr131;
                uint16_t wcr134 = wcr133 + wcr132;
                uint16_t wcr135 = wcr134 + wcr133;
                uint16_t wcr136 = wcr135 + wcr134;
                uint16_t wcr137 = wcr136 + wcr135;
                uint16_t wcr138 = wcr137 + wcr136;
                uint16_t wcr139 = wcr138 + wcr137;
                uint16_t wcr140 = wcr139 + wcr138;
                uint16_t wcr141 = wcr140 + wcr139;
                uint16_t wcr142 = wcr141 + wcr140;
                uint16_t wcr143 = wcr142 + wcr141;
                uint16_t wcr144 = wcr143 + wcr142;
                uint16_t wcr145 = wcr144 + wcr143;
                uint16_t wcr146 = wcr145 + wcr144;
                uint16_t wcr147 = wcr146 + wcr145;
                uint16_t wcr148 = wcr147 + wcr146;
                uint16_t wcr149 = wcr148 + wcr147;
                uint16_t wcr150 = wcr149 + wcr148;
                uint16_t wcr151 = wcr150 + wcr149;
                uint16_t wcr152 = wcr151 + wcr150;
                uint16_t wcr153 = wcr152 + wcr151;
                uint16_t wcr154 = wcr153 + wcr152;
                uint16_t wcr155 = wcr154 + wcr153;
                uint16_t wcr156 = wcr155 + wcr154;
                uint16_t wcr157 = wcr156 + wcr155;
                uint16_t wcr158 = wcr157 + wcr156;
                uint16_t wcr159 = wcr158 + wcr157;
                uint16_t wcr160 = wcr159 + wcr158;
                uint16_t wcr161 = wcr160 + wcr159;
                uint16_t wcr162 = wcr161 + wcr160;
                uint16_t wcr163 = wcr162 + wcr161;
                uint16_t wcr164 = wcr163 + wcr162;
                uint16_t wcr165 = wcr164 + wcr163;
                uint16_t wcr166 = wcr165 + wcr164;
                uint16_t wcr167 = wcr166 + wcr165;
                uint16_t wcr168 = wcr167 + wcr166;
                uint16_t wcr169 = wcr168 + wcr167;
                uint16_t wcr170 = wcr169 + wcr168;
                uint16_t wcr171 = wcr170 + wcr169;
                uint16_t wcr172 = wcr171 + wcr170;
                uint16_t wcr173 = wcr172 + wcr171;
                uint16_t wcr174 = wcr173 + wcr172;
                uint16_t wcr175 = wcr174 + wcr173;
                uint16_t wcr
```



Application mapping on CGRA

- Software pipelined schedule
- Goal: Mapping with minimum initiation interval
- Initiation interval (II) = cycle difference between initiation of consecutive iterations
- Low II \rightarrow High performance



Motivation for Morpher

- CGRA has become popular due to its excellent balance between performance, power efficiency and flexibility
- Researchers are exploring the design space to find a better mix of performance, power efficiency, and flexibility targeting diverse workloads
 - Large architecture design space
 - Workloads have complex kernels
- CGRA design frameworks that can only map simple kernels on a fixed architecture are not suitable for CGRA design space exploration
- Functional verification is important to validate the correctness

What is missing?

- Not end-to-end
 - No open-source DFG generators (Pillars)
 - No open-source simulators (CGRA-ME)
 - No open-source architecture adaptive compiler (OpenCGRA, CCF)
- Cannot support diverse CGRA architectures:
 - Does not support architecture adaptive compilation (Open-CGRA, CCF)
 - Only covers basic architectural features in PE array (CGRA-ME)
- Cannot support complex kernels
 - Does not support kernels with control divergence and loops with recurrence edges (CGRA-ME, Pillars)
- Need to manually create test benches and test data for each application kernel

Features		CGRA-ME	Pillars	Open-CGRA	CCF
DFG Generation	Models control divergence	✗	✗	✓	✓
	Recurrence edges	✗	✗	✓	✓
Architecture Modelling	Adapt user defined architectures	✓	✓	✓	✗
	Multi-hop connections	✗	✗	✗	✗
	Different memory organizations	✗	✗	✓	✗
P&R Mapper	Architecture adaptive mapping	✓	✓	✗	✗
	Data layout aware mapping	✗	✗	✗	✗
	Recurrence aware mapping	✗	✗	✓	✓
Simulation & validation	Cycle accurate simulation	✗	✓	✓	✓
	Test data generation	✗	✗	✗	✗
	Validation against test data	✗	✗	✗	✗

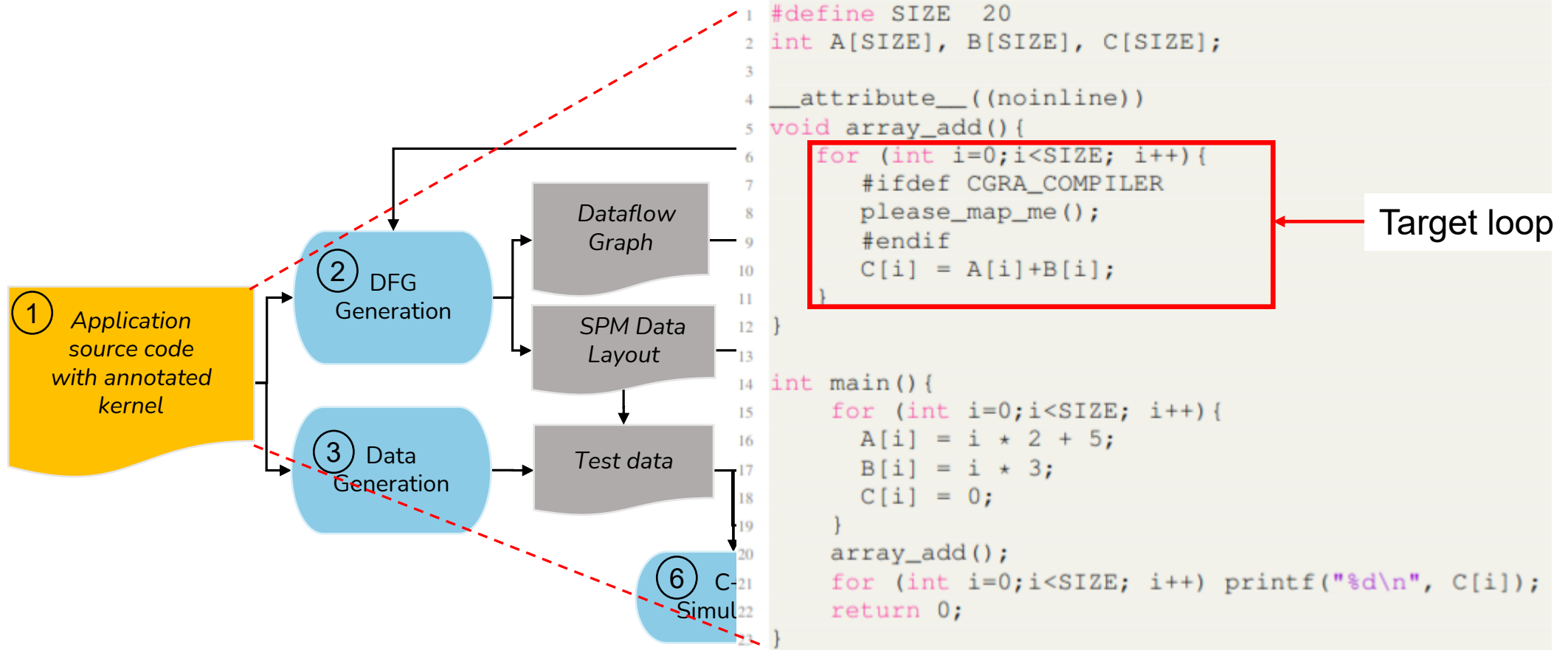
Morpher: An Integrated Compilation and Simulation Framework for CGRA

- Fully automated end-to-end CGRA compilation and simulation framework
- Flexible architecture specification language
 - Allows the user to define complex CGRA architectures
- Supports kernels with control divergence and recurrence edges
 - Allows compiling complex application kernels
- Efficient mapping algorithms
 - Higher mapping quality at shorter compilation time
- Cycle-accurate simulation to validate the compilation results
 - Automatically extract the test data from the target application
- Fully open-source with easily modifiable modular code base:
 - <https://github.com/ecolab-nus/morpher>

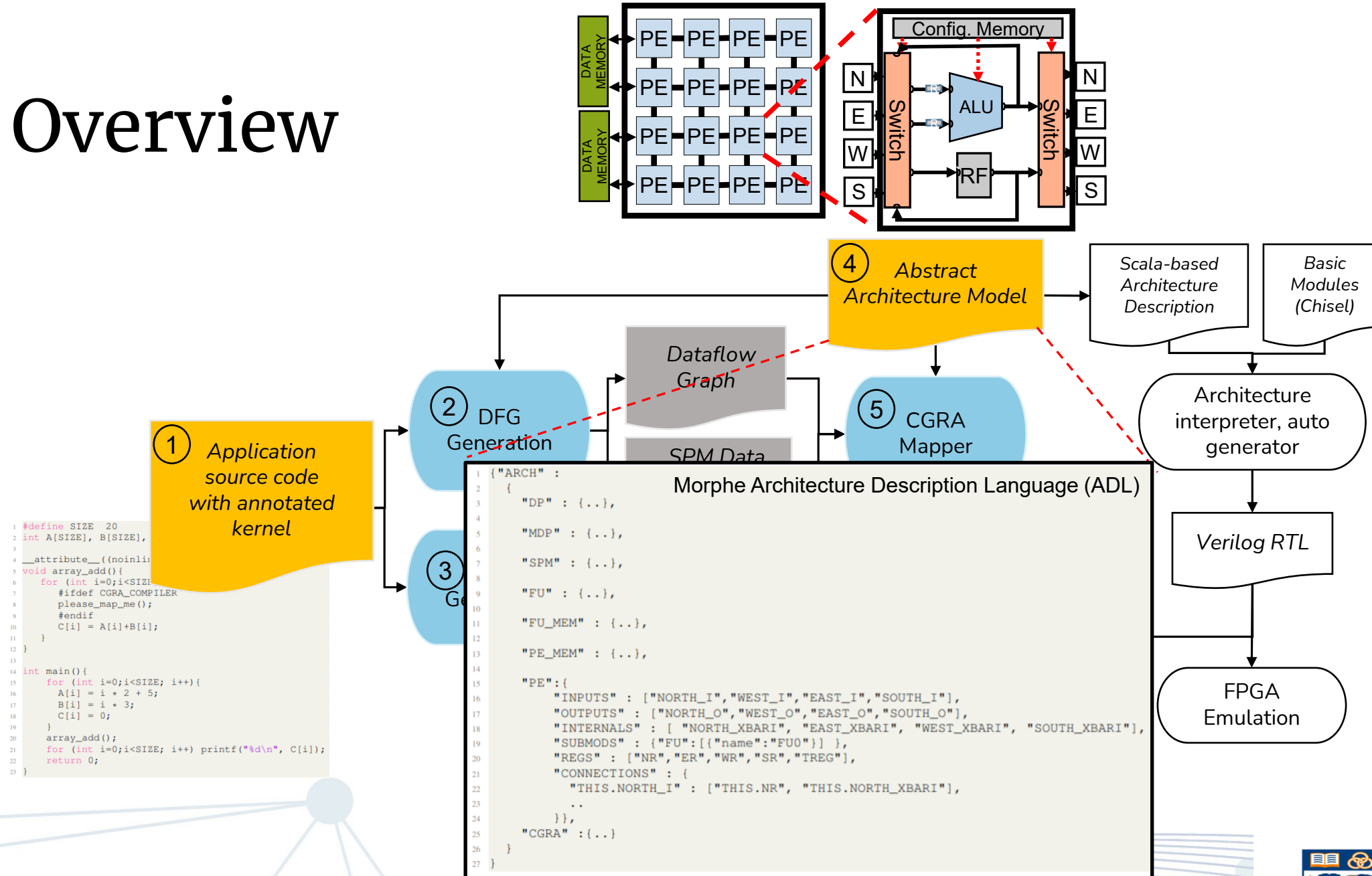
Features		CGRA-ME	Pillars	Open-CGRA	CCF	Morpher
DFG Generation	Models control divergence	X	X	✓	✓	✓
	Recurrence edges	X	X	✓	✓	✓
Architecture Modelling	Adapt user defined architectures	✓	✓	✓	X	✓
	Multi-hop connections	X	X	X	X	✓
	Different memory organizations	X	X	✓	X	✓
P&R Mapper	Architecture adaptive mapping	✓	✓	X	X	✓
	Data layout aware mapping	X	X	X	X	✓
	Recurrence aware mapping	X	X	✓	✓	✓
Simulation & validation	Cycle accurate simulation	X	✓	✓	✓	✓
	Test data generation	X	X	X	X	✓
	Validation against test data	X	X	X	X	✓



Overview



Overview



Overview

```

1 #define SIZE 20
2 int A[SIZE], B[SIZE],
3
4 __attribute__((noinline))
5 void array_add() {
6     for (int i=0; i<SIZE; i++) {
7         #ifdef CGRA_COMPILER
8             please_map_me();
9         #endif
10        C[i] = A[i]+B[i];
11    }
12 }
13
14 int main() {
15     for (int i=0; i<SIZE; i++) {
16         A[i] = i * 2 + 5;
17         B[i] = i * 3;
18         C[i] = 0;
19     }
20     array_add();
21     for (int i=0; i<SIZE; i++) printf("%d\n", C[i]);
22     return 0;
23 }

```

① Application source code with annotated kernel

② DFG Generation

③ Data Generation

Dataflow Graph

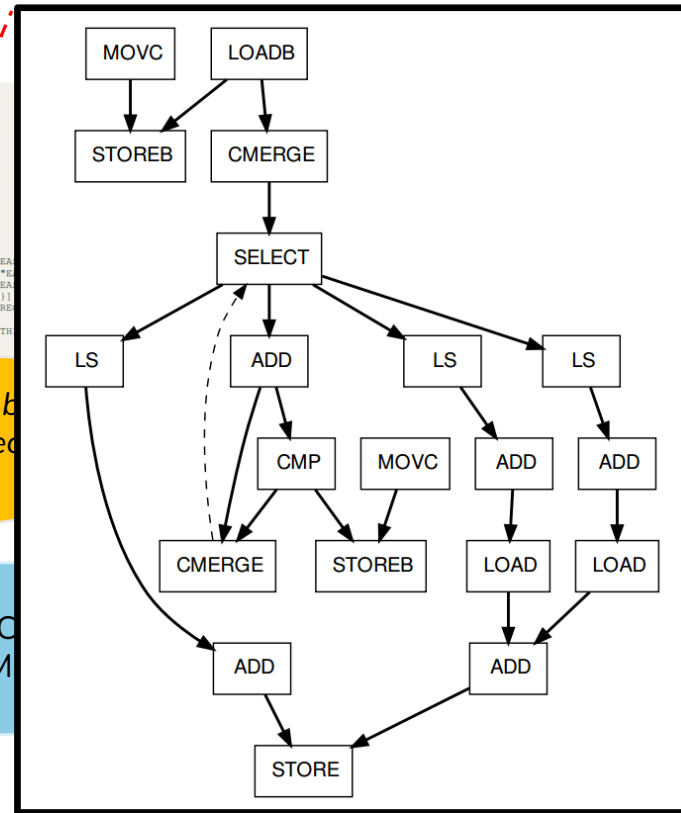
SPM Data Layout

Test data

⑥ C++ Simulation

④ Abstr. Architect

⑤ CGRA Mapping



```

1 <DFG count="20">
2 <Node idx="11" ASAP="0" ALAP="0"BB="entry"CONST="4094">
3 <OP>LOADB</OP>
4 <BasePointerName size="1">loopstart</BasePointerName>
5 <Inputs>
6 </Inputs>
7 <Outputs>
8 <Output idx="12" nextiter="0" NPB="0" type="P"/>
9 <Output idx="19" nextiter="0" NPB="0" type="P"/>
10 </Outputs>
11 <RecParents>
12 </RecParents>
13 </Node>
14
15 <Node idx="12" ASAP="1" ALAP="1"BB="for.body_0_0"CONST="0">
16 <OP>CMERGE</OP>
17 <Inputs>
18 <Input idx="11"/>
19 </Inputs>
20 <Outputs>
21 <Output idx="0" nextiter="0" type="I1"/>
22 </Outputs>
23 <RecParents>
24 </RecParents>
25 </Node>
26 ..

```

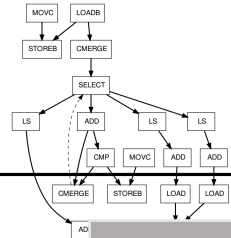
Overview

```
1 #define SIZE 20
2 int A[SIZE], B[SIZE],
3
4 __attribute__((noinline))
5 void array_add() {
6     for (int i=0; i<SIZE; i++) {
7         #ifdef CGRA_COMPILER
8             please_map_me();
9         #endif
10        C[i] = A[i]+B[i];
11    }
12 }
13
14 int main() {
15     for (int i=0; i<SIZE; i++) {
16         A[i] = i * 2 + 5;
17         B[i] = i * 3;
18         C[i] = 0;
19     }
20     array_add();
21     for (int i=0; i<SIZE; i++) printf("%d\n", C[i]);
22     return 0;
23 }
```

① Application source code with annotated kernel

② DFG Generation

③ Data Generation



Dataflow Graph

SPM Data Layout

Test data

④ Abstract Architecture Model

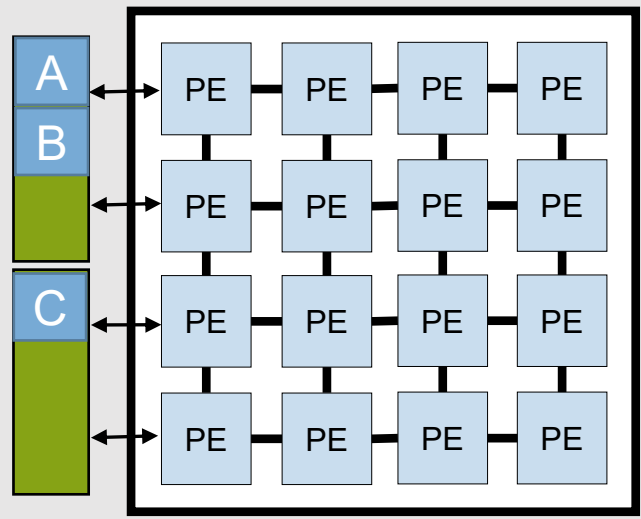
⑤ CGRA Mapper

Mapping Configurations

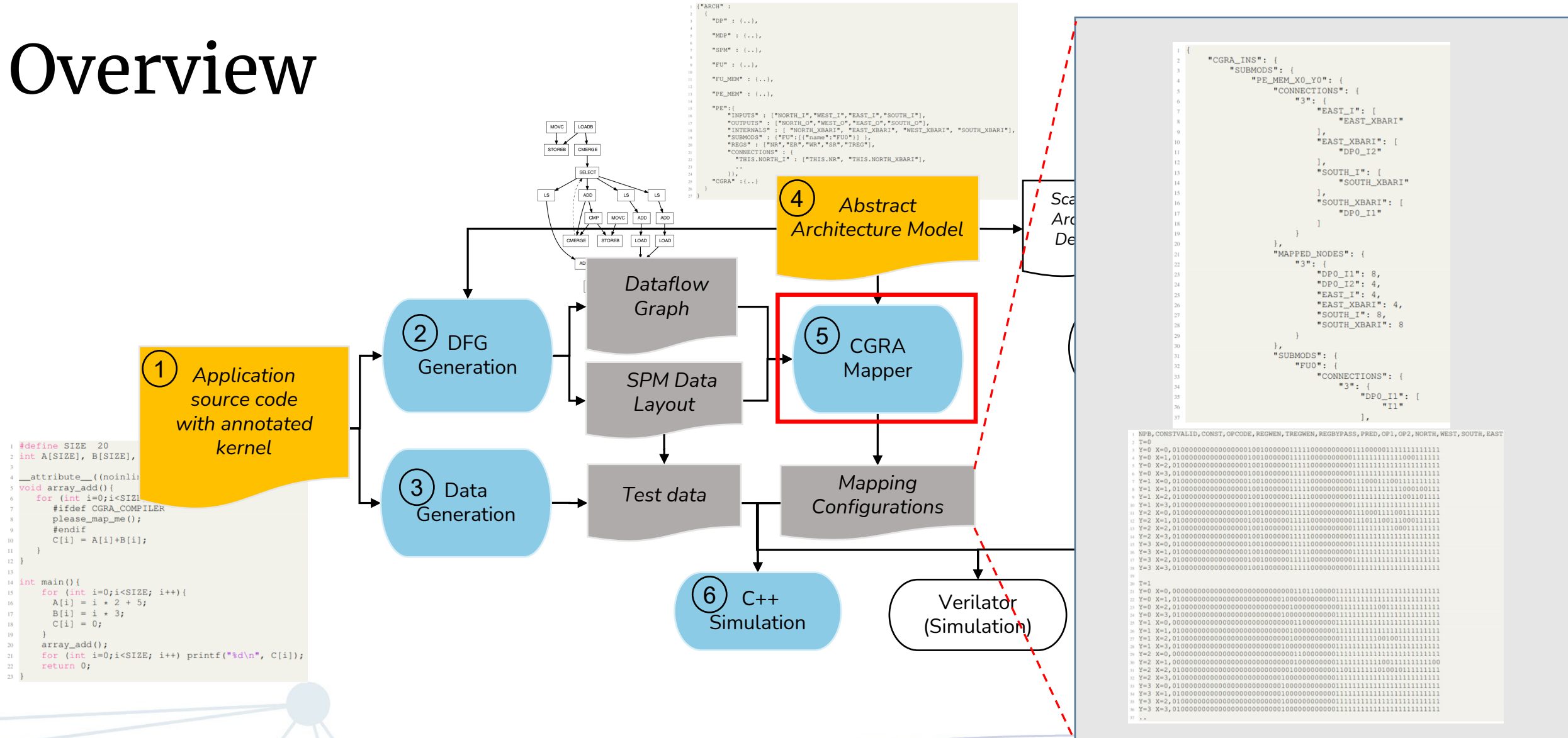
⑥ C++ Simulation

Verification (Simulation)

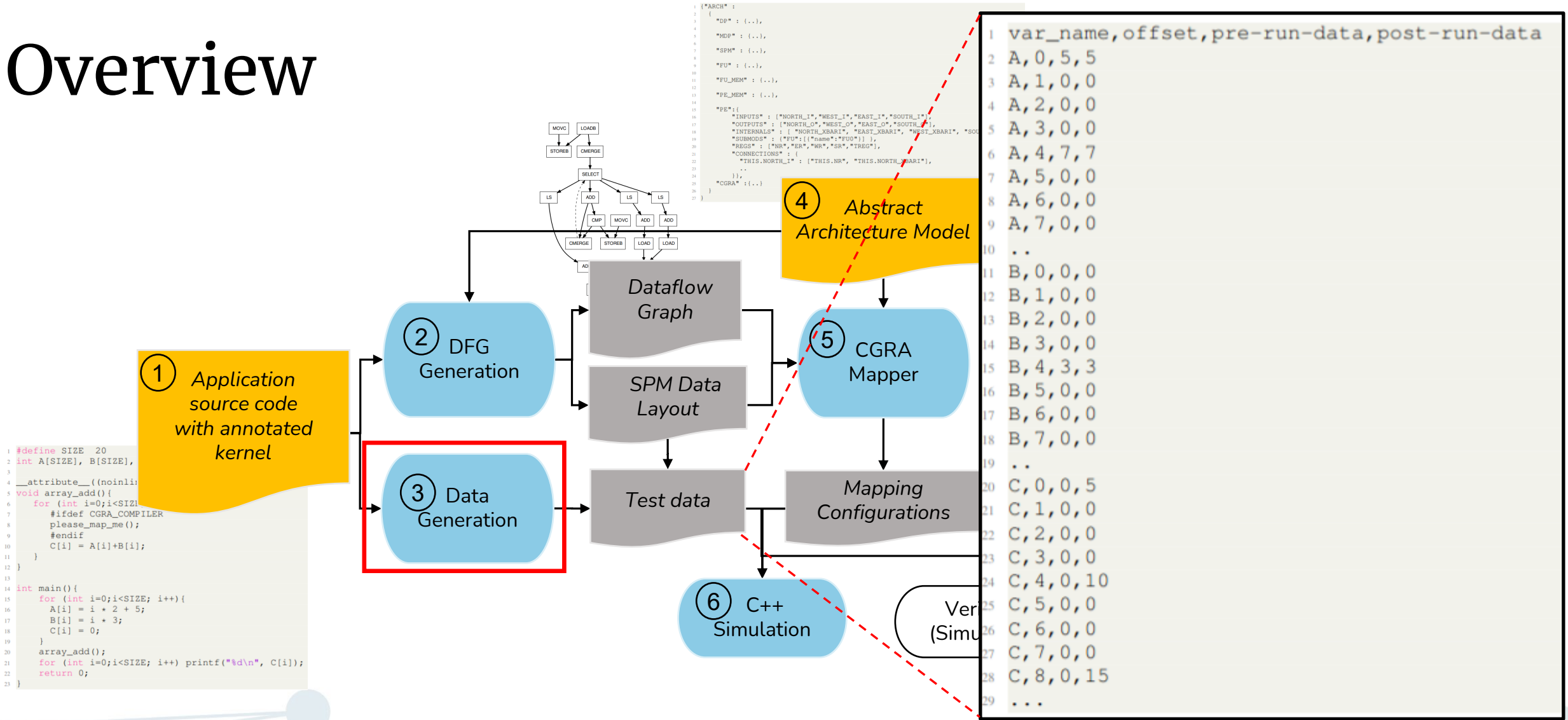
```
1 var_name, base_addr
2 A, 0
3 B, 80
4 C, 2048
5 loopend, 2047
6 loopstart, 4094
```



Overview



Overview



Overview

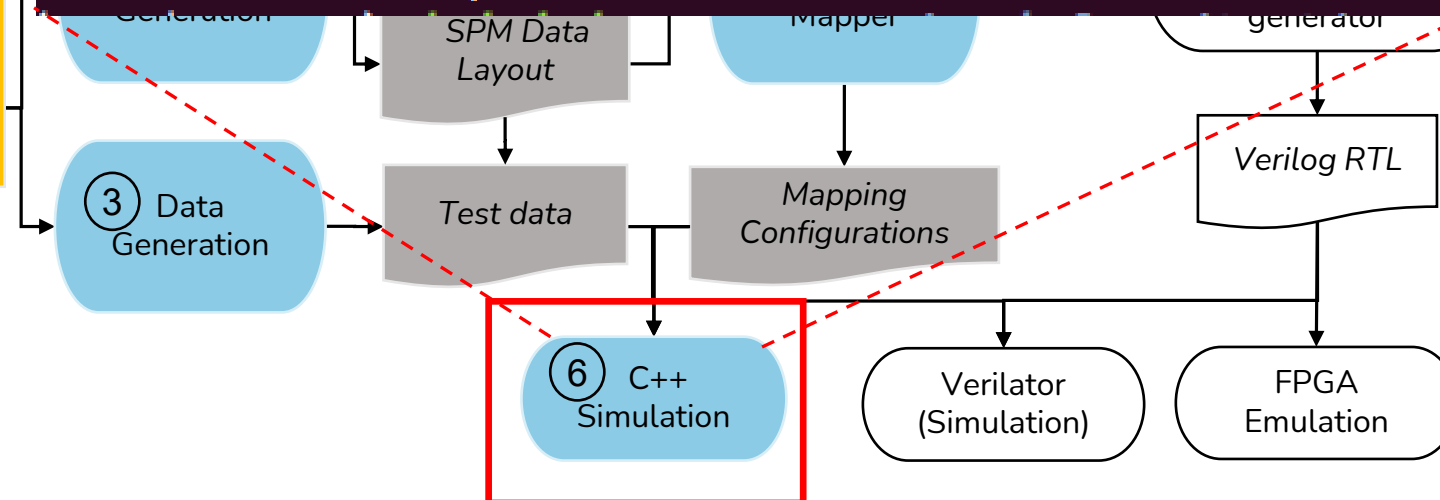
```
1 ["ARCH" :  
2 {  
3   "DP" : {...},  
4   "MDP" : {...},  
5   "SPM" : {...},  
6   "FU" : {...},  
7   "FU_MEM" : {...},  
8   "PE_MEM" : {...},  
9 }
```

-----Running hycube_simulator-----

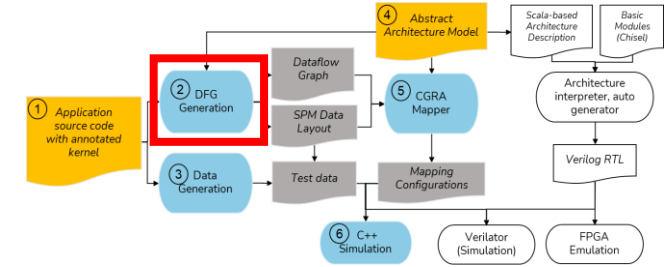
```
0%|  
Simulation Result: Matches::243, Mismatches::0  
100%|  
Matches: 243 Mismatches: 0  
Simulation test passed!!!
```

① Application source code with annotated kernel

```
1 #define SIZE 20  
2 int A[SIZE], B[SIZE],  
3  
4 __attribute__((noinline))  
5 void array_add() {  
6   for (int i=0; i<SIZE; i++)  
7     #ifdef CGRA_COMPILER  
8       please_map_me();  
9     #endif  
10    C[i] = A[i]+B[i];  
11  }  
12  
13  
14 int main() {  
15   for (int i=0; i<SIZE; i++) {  
16     A[i] = i * 2 + 5;  
17     B[i] = i * 3;  
18     C[i] = 0;  
19   }  
20   array_add();  
21   for (int i=0; i<SIZE; i++) printf("%d\n", C[i]);  
22   return 0;  
23 }
```

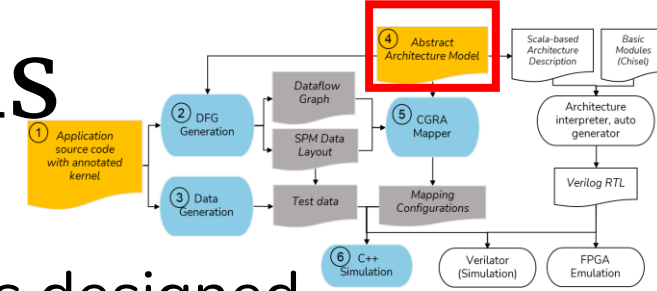


DFG and Data Layout Generation



- DFG generation: supports DFG generation for multiple execution paradigms
 - Control-flow handling techniques
 - Partial predication, Full predication, Dual-issue
 - Load store address generation techniques
 - On array address generation, decoupled access/execute address generation
 - Memory models
 - Memory-mapped, tightly coupled
- Data layout generation: creates multi-bank data layout by allocating live-in/live-out variables (scalar and arrays) in the CGRA data memory
 - Supports simple data placement policies

Abstract Architecture Specifications



- Morpher abstract Architecture Description Language (ADL) is designed to cover diverse CGRA architectures
- Three main components
 - Modules: Represent hardware blocks (PEs, RFs, ALUs, LSUs, and Memories)
 - Primitive modules:
 - Functional Units (FU)
 - Register Files (RF)
 - Memory Units (MU)
 - Ports: Entry and exit points of the modules that carry data
 - Connections: Describe the connectivity among ports
- Multiplexers are automatically inferred through the connections


```

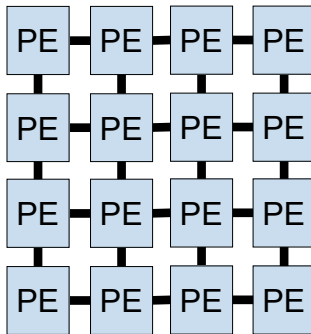
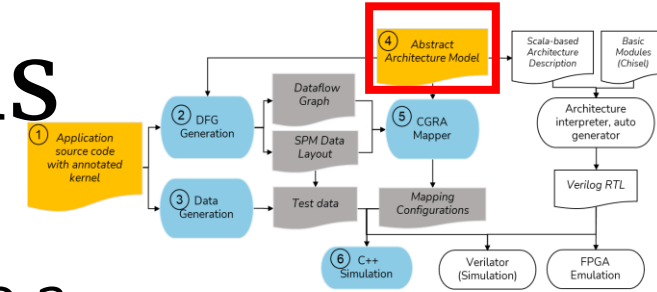
graph LR
    1[1 Application source code with annotated kernel] --> 2[2 DFG Generation]
    1 --> 3[3 Data Generation]
    2 --> DG[Dataflow Graph]
    2 --> SPM[SPM Data Layout]
    3 --> TD[Test data]
    DG --> 5[5 CGRA Mapper]
    SPM --> 5
    TD --> MC[Mapping Configurations]
    5 --> MC
    MC --> 6[6 C++ Simulation]
    5 --> 4[4 Abstract Architecture Model]
    4 --> SAD[Scale-based Architecture Description]
    4 --> BM[Basic Modules Chisel]
    SAD --> AIG[Architecture interpreter, auto generator]
    AIG --> VRTL[Verilog RTL]
    VRTL --> V[Verilator Simulation]
    VRTL --> FE[FPGA Emulation]
  
```

- ```
"PE":{
 "INPUTS" : ["WI","EI"] ,
 "OUTPUTS" : ["WO","EO"] ,
 "SUBMODS" : {
 "FU":[{"name":"FU0"}], "RF":[{"name":"RF0"}]},
 "CONNECTIONS" : {
 "THIS.EI" : ["FU0.I1","FU0.I2","FU0.P", "RF0.WP"],
 "THIS.WI" : ["FU0.I1","FU0.I2","FU0.P", "RF0.WP"],
 "RF0.RPO" : ["FU0.I1","FU0.I2","FU0.P","THIS.EO","THIS.WO"],
 "FU0.T" : ["RF0.WP","THIS.EO","THIS.WO"]
 }
},
```

Figure 1 illustrates the architecture of a Processing Element (PE). The PE is a large rounded rectangle containing several internal components. On the left, there are two input ports labeled WI (Write Input) and EI (Execute Input). On the right, there are two output ports labeled WO (Write Output) and EO (Execute Output). Inside the PE, there are two main functional units: FU0 (Function Unit 0) and RF0 (Register File 0). FU0 is a purple-outlined box containing a DPO (Data Path Operator) and a T (Tag) module. RF0 is a green-outlined box containing two registers, REG0 and REG1, and a RP (Read Pointer) module. Connections are shown between the ports and the internal modules. A large arrow on the left indicates the flow of data into the PE.

# Abstract Architecture Specifications

- Morpher ADL provides a special syntax to connect the PEs to a given interconnection pattern automatically
  - Eliminates the need for the user to specify all the connections



PEs connected in a 2-D “GRID” Pattern

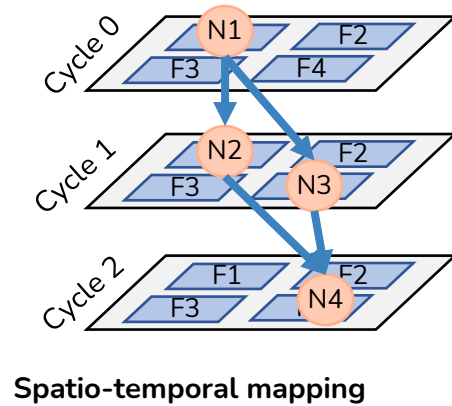
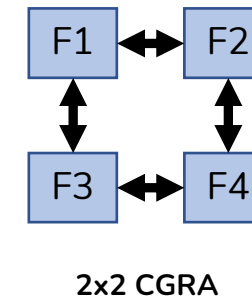
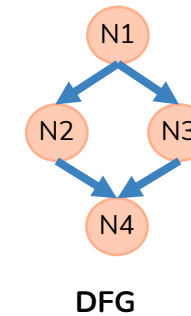
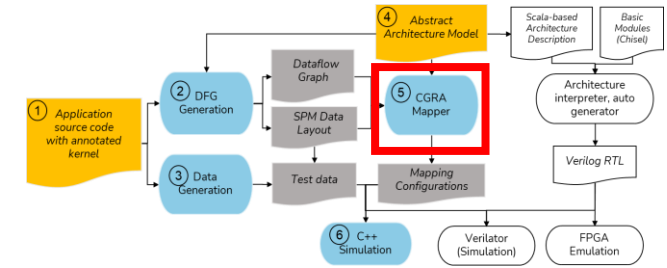
```
"CGRA" : {
 "SUBMODS" : [
 { "PATTERN" : "GRID",
 "DIMS" : { "X" : 4, "Y" : 4 },
 "MODS" : [
 { "X":0, "Y":0, "MOD": "PE_MEM" }, { "X":1, "Y":0, "MOD": "PE" }, { "X":2, "Y":0, "MOD": "PE" }, { "X":3, "Y":0, "MOD": "PE" },
 { "X":0, "Y":1, "MOD": "PE_MEM" }, { "X":1, "Y":1, "MOD": "PE" }, { "X":2, "Y":1, "MOD": "PE" }, { "X":3, "Y":1, "MOD": "PE" },
 { "X":0, "Y":2, "MOD": "PE_MEM" }, { "X":1, "Y":2, "MOD": "PE" }, { "X":2, "Y":2, "MOD": "PE" }, { "X":3, "Y":2, "MOD": "PE" },
 { "X":0, "Y":3, "MOD": "PE_MEM" }, { "X":1, "Y":3, "MOD": "PE" }, { "X":2, "Y":3, "MOD": "PE" }, { "X":3, "Y":3, "MOD": "PE" }
],
 "CONNECTIONS" : [
 { "FROM_X" : "X", "FROM_Y" : "Y", "FROM_PORT" : "NORTH_0", "TO_X" : "X", "TO_Y" : "Y-1", "TO_PORT" : "SOUTH_I" },
 { "FROM_X" : "X", "FROM_Y" : "Y", "FROM_PORT" : "EAST_0", "TO_X" : "X+1", "TO_Y" : "Y", "TO_PORT" : "WEST_I" },
 { "FROM_X" : "X", "FROM_Y" : "Y", "FROM_PORT" : "WEST_0", "TO_X" : "X-1", "TO_Y" : "Y", "TO_PORT" : "EAST_I" },
 { "FROM_X" : "X", "FROM_Y" : "Y", "FROM_PORT" : "SOUTH_0", "TO_X" : "X", "TO_Y" : "Y+1", "TO_PORT" : "NORTH_I" }
]
 }
]
}
```

Syntax for the “GRID” Pattern

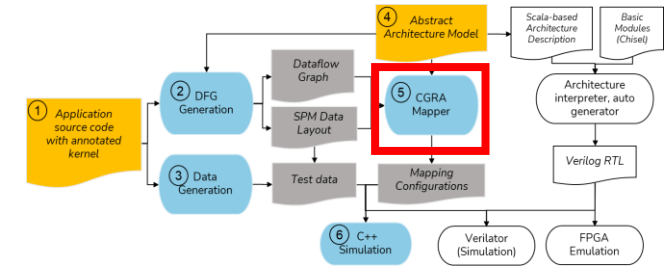
([https://github.com/ecolab-nus/morpher/Morpher\\_CGRA\\_Mapper/json\\_arch/stdnoc.json](https://github.com/ecolab-nus/morpher/Morpher_CGRA_Mapper/json_arch/stdnoc.json))

# CGRA Mapper

- Mapper generates the configurations of hardware elements (ALU, Mux, RF read/write)
  - Goal of the mapper is to generate a mapping with the lowest possible II (Minimum II)
- Algorithm:
  - Minimum II is calculated
    - based on the CGRA array size, number of DFG nodes, and the recurrence dependencies in the kernel
  - Create Modulo Routing Resource Graph (MRRG) with MII
  - Create the initial mapping allowing resource overuse:
    - Create a scheduling list: sort the nodes of the DFG in a topological ordering
    - Map each DFG node to a MRRG node with the least accumulated routing cost
  - Iteratively resolves the resource overuse
  - Increase II and redo the mapping till a feasible schedule is obtained



# CGRA Mapper



- Morpher currently supports three approaches to resolve resource overuse
  - Adaptive-heuristic-based approach inspired by SPR
    - Cost of the over-subscribed ports is increased for the next mapping iterations
    - Resources with more demand would be available for routing the dependencies with fewer options
  - Simulated Annealing (SA) based approach
    - Node placement is changed based on an SA-based cooling schedule
  - Learning-induced (LISA) approach [1]
    - Node placement is guided by labels inferred from a trained Graph Neural Network (GNN) model
- Our modular code base allows researchers to add their mapping methods
  - In the future, we plan to incorporate hierarchical mapping approaches for better scalability [2, 3]

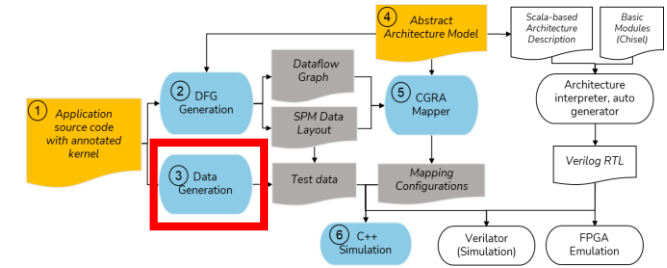
[1] Z. Li, D. Wu, D. Wijerathne, and T. Mitra, "LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*

[2] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "HiMap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[3] D. Wijerathne, Z. Li, T. K. Bandara, and T. Mitra, "PANORAMA: Divide-and-Conquer Approach for Mapping Complex Loop Kernels on CGRA," in *Proceedings of the 59th Annual Design Automation Conference 2022*



# Test Data Generation



- Instrument, i.e., insert data recording functions to the C source code to capture live-in and live-out values

- Instrumented C source code to record the live-in variable recording functions

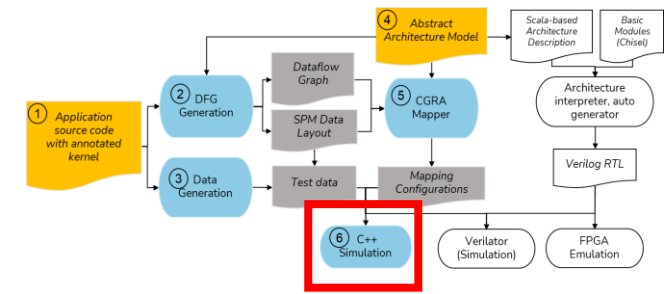
- Simulator uses the target loop
  - Live-in value
  - Live-out variable

```

1 define dso_local void @array_add() local_unnamed_addr #0 {
2 entry:
3 call void @loopStart(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @0, i32 0, i32 0))
4 call void @LiveInReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @1, i32 0, i32 0), i8* bitcast([20 x i32]* @C to i8*), i32 80)
5 call void @LiveInReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @2, i32 0, i32 0), i8* bitcast([20 x i32]* @A to i8*), i32 80)
6 call void @LiveInReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @3, i32 0, i32 0), i8* bitcast([20 x i32]* @B to i8*), i32 80)
7 br label %for.body
8
9 for.body:
10 ; preds = %entry, %for.body
11 %i.08 = phi i32 [0, %entry], [%inc, %for.body]
12 %arrayidx = getelementptr inbounds [20 x i32], [20 x i32]* @A, i32 0, i32 %i.08
13 %manupa0 = load i32, i32* %arrayidx, align 4, !tbaa !3
14 %arrayidx1 = getelementptr inbounds [20 x i32], [20 x i32]* @B, i32 0, i32 %i.08
15 %manupa1 = load i32, i32* %arrayidx1, align 4, !tbaa !3
16 %add = add nsw i32 %manupa1, %manupa0
17 %arrayidx2 = getelementptr inbounds [20 x i32], [20 x i32]* @C, i32 0, i32 %i.08
18 store i32 %add, i32* %arrayidx2, align 4, !tbaa !3
19 %inc = add nuw nsw i32 %i.08, 1
20 %exitcond = icmp eq i32 %inc, 20
21 br i1 %exitcond, label %for.cond.cleanup, label %for.body
22
23 for.cond.cleanup:
24 ; preds = %for.body
25 call void @LiveOutReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @1, i32 0, i32 0), i8* bitcast([20 x i32]* @C to i8*), i32 80)
26 call void @LiveOutReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @2, i32 0, i32 0), i8* bitcast([20 x i32]* @A to i8*), i32 80)
27 call void @LiveOutReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @3, i32 0, i32 0), i8* bitcast([20 x i32]* @B to i8*), i32 80)
28 call void @loopEnd(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @4, i32 0, i32 0))
29 ret void
30 }

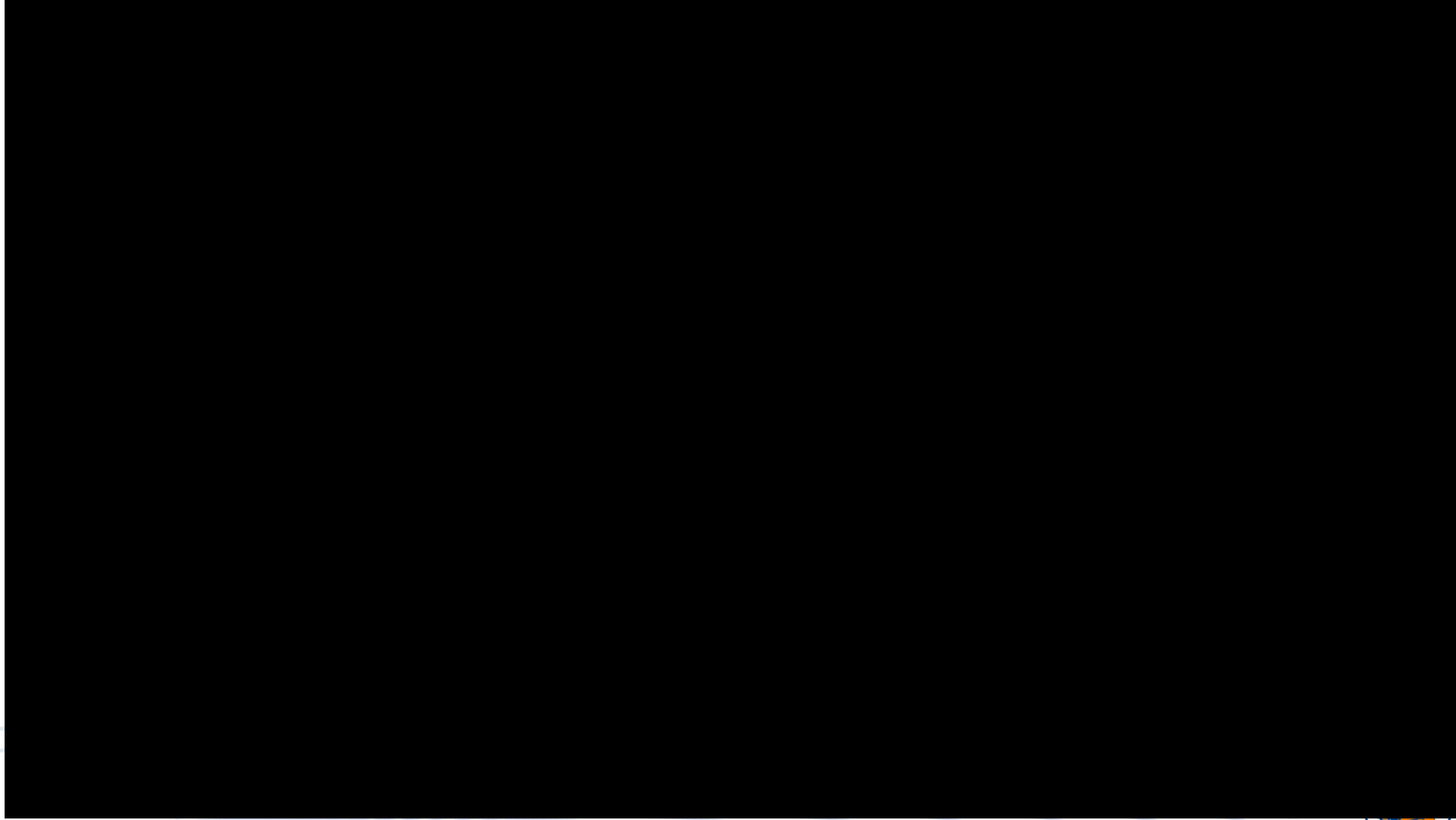
```

# Simulation and Verification



- Morpher Simulator is a model of the CGRA composed of Fus, registers, multiplexers, and memories
- Currently Morpher simulator only models the variations of HyCUBE CGRA architecture
- CGRA model acts as a memory-mapped slave device to a host processor
  - The live-in variables are loaded for each memory unit
  - Simulator executes operations mapped on FUs, multiplexes the data, and writes data to registers/memories on a cycle basis according to the mapping configurations
  - Post-simulation memory content is validated against the expected results

# Open-source Artifact Demonstration



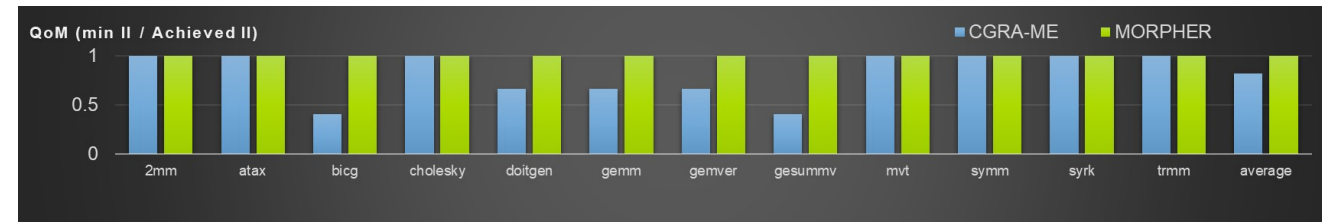
# Experimental Study

1. Mapping quality and compilation time
2. Map real applications kernels with automated verification
3. The ability to model diverse CGRA architectures

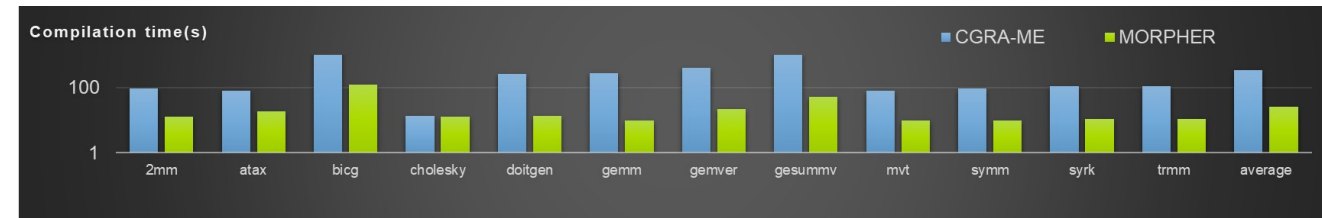


# Mapping Quality and Compilation Time

- Comparison with CGRA-ME, the only open-source framework available for architecture-adaptive compilation
- Target architecture: 4x4 CGRA
  - Interconnected in N2N connections
  - Each PE has RF with four registers
- Kernels: Polybench benchmark suite
- Results:
  - Morpher achieves the minimum possible II for all kernels, while CGRA-ME failed to do so for five kernels.
  - Morpher LISA mapping is 13.3x faster than CGRA-ME on average.



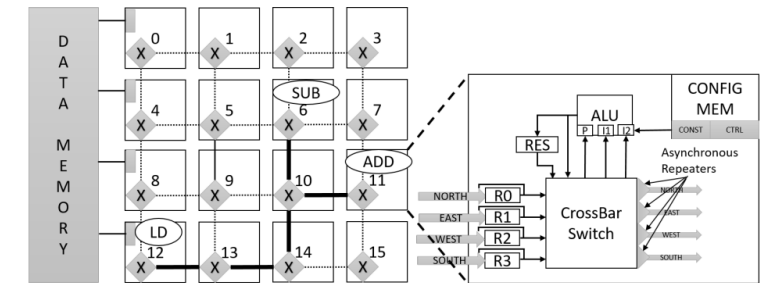
Quality of Mapping (min II/ Achieved II)



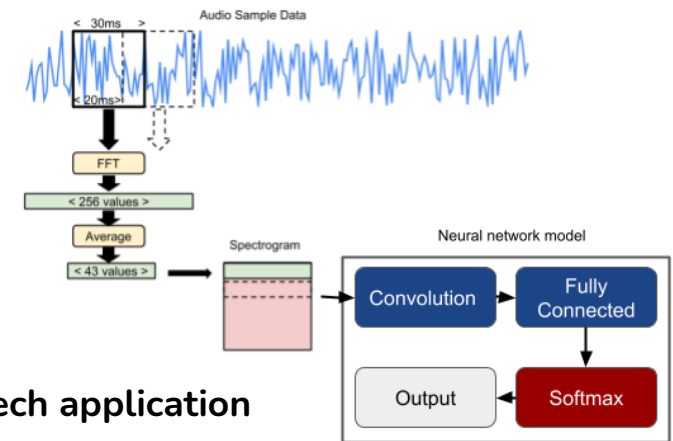
Compilation Time (s)

# Accelerating Real Applications on CGRA

- This study demonstrates
  - How Morpher can be used to accelerate real application kernels on CGRA-based systems
  - The importance of supporting control divergence and recurrence edges
- Target CGRA: 4x4 HyCUBE with a partial predication-based execution model
- Target application: Microspeech (wake-word detection) from tinyML benchmark suite
  - Target kernel: Convolution layer



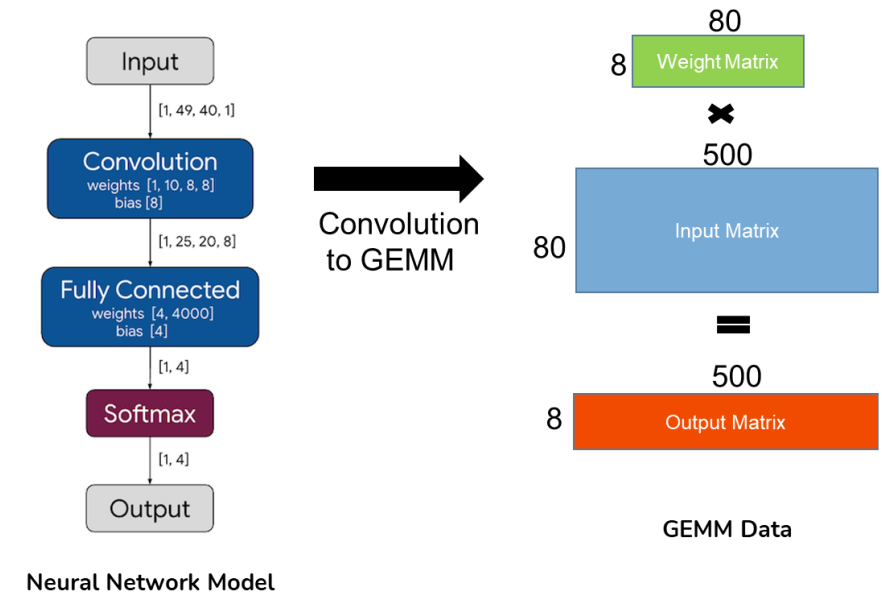
HyCUBE CGRA



Microspeech application

# Accelerating Real Applications on CGRA

- This study demonstrates
  - How Morpher can be used to accelerate real application kernels on CGRA-based systems
  - The importance of supporting control divergence and recurrence edges
- Target CGRA: 4x4 HyCUBE with a partial predication-based execution model
- Target application: Microspeech (wake-word detection) from tinyML benchmark suite
  - Target kernel: Convolution layer
  - Convolution layer is lowered to GEMM kernel



# Accelerating Real Applications on CGRA

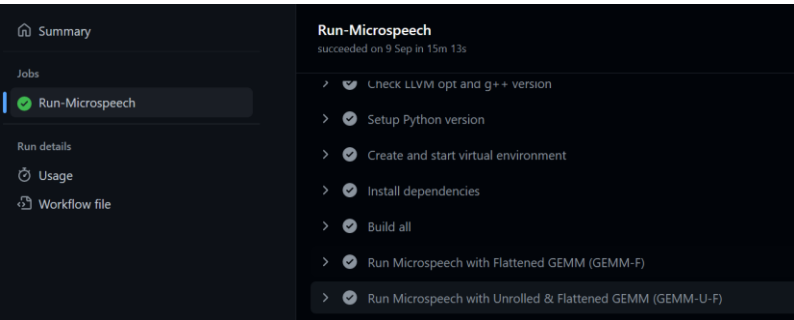
- We consider three variations of the GEMM kernel
  - GEMM
  - GEMM-U: unrolled four times
  - GEMM-U-F: unrolled and flattened
    - Flattening introduces conditional statements inside the loop body and adds a long recurrence edge
    - Flattened kernel does not have invocation overheads
- Results:
  - Morpher can successfully compile all three kernels
  - GEMM-U kernel reduces the compute time by nearly half compared to the GEMM
  - GEMM-U-F offers the best total execution time
  - Automatically verifies the compiled kernels by running simulations using pre-processed audio data extracted from the application

```
1 for (i=0; i<R1; i++)
2 for (j=0; j<C2; j++)
3 for (k=0; k<C1; k++): //map this (GEMM)
4 O[i][j] += W[i][k] * I[k][j];
5
6 for (i=0; i<R1; i++)
7 for (j=0; j<C2; j++)
8 for (k=0; k<C1; k+=4): //map this (GEMM-U)
9 O[i][j] += W[i][k] * I[k][j]
10 + W[i][k+1] * I[k+1][j]
11 + W[i][k+2] * I[k+2][j]
12 + W[i][k+3] * I[k+3][j];
13
14 for (n=0; n<R1*C2*C1; n++) { //map this (GEMM-U-F)
15 O[i][j] += W[i][k] * I[k][j]
16 + W[i][k+1] * I[k+1][j]
17 + W[i][k+2] * I[k+2][j]
18 + W[i][k+3] * I[k+3][j];
19 k+=4;
20 if (k+1>=C1) {k=0; ++j;}
21 if (j==C2) {j=0; ++i;}
22 }
```

GEMM kernels

| Kernel   | Nodes | II (MII) | Compute time (ms) | Data transfer time (ms) | Total execution time (ms) |
|----------|-------|----------|-------------------|-------------------------|---------------------------|
| GEMM     | 26    | 4 (4)    | 2.70              | 3.39                    | 6.09                      |
| GEMM-U   | 58    | 6 (4)    | 1.17              | 3.39                    | 4.56                      |
| GEMM-U-F | 79    | 8 (8)    | 1.31              | 1.79                    | 3.10                      |

Execution time



CI workflow for microspeech kernels

# Modelling Complex CGRAs

|                               |        | adpcm |                  | aes |                  | dct |                  | fft |                  | gsm |                  |
|-------------------------------|--------|-------|------------------|-----|------------------|-----|------------------|-----|------------------|-----|------------------|
| MII                           |        | 7     |                  | 10  |                  | 9   |                  | 4   |                  | 6   |                  |
| Architecture case studies     |        | II    | Compile Time (m) | II  | Compile Time (m) | II  | Compile Time (m) | II  | Compile Time (m) | II  | Compile Time (m) |
| Generic CGRA                  |        | 17    | 323              | 24  | 410              | 23  | 367              | 11  | 467              | 10  | 409              |
| ALU-Independent Routing       |        | 9     | 14               | 19  | 79               | 15  | 22               | 5   | 16               | 8   | 88               |
| Multi-hop Routing             | 2-Hops | 9     | 8                | 15  | 17               | 12  | 12               | 5   | 2                | 8   | 96               |
|                               | 3-Hops | 9     | 4                | 13  | 5                | 10  | 8                | 5   | 10               | 7   | 39               |
|                               | 4-Hops | 8     | 1                | 13  | 3                | 11  | 4                | 5   | 10               | 10  | 175              |
| Multi-Hop Scattered Registers | 4-Hops | 8     | 1                | 12  | 2                | 11  | 3                | 5   | 7                | 7   | 18               |



# Conclusion

- Morpher CGRA compilation and simulation framework
  - Flexible to model modern CGRA architectures
  - Map complex workloads with a higher mapping quality at a shorter compilation time
  - Automatically validate the correctness of the mapping results through cycle-accurate simulation
- Fully open-source
  - Modular codebase
  - Easy to modify
- Open source repository:
  - <https://github.com/ecolab-nus/morpher>





Thank you!

