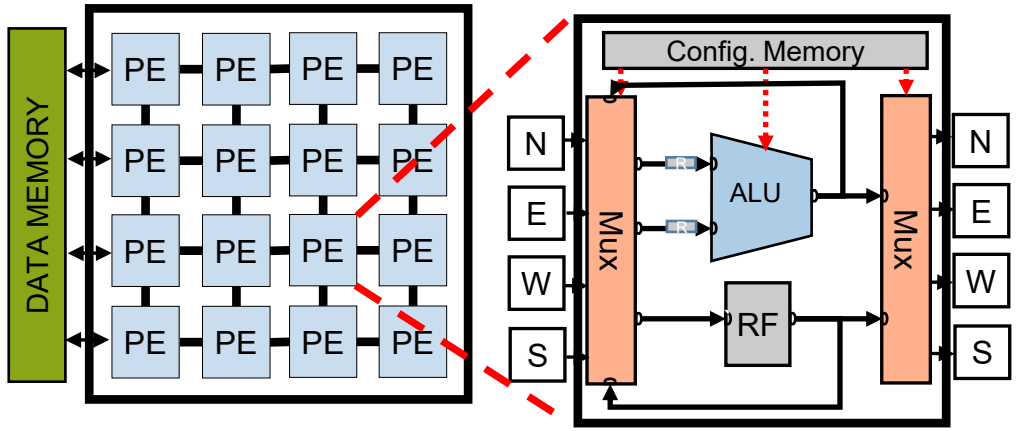
Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA

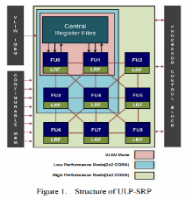
Dhananjaya Wijerathne*, Zhaoying Li *, Manupa Karunaratne , Li-Shiuan Peh *, Tulika Mitra *
School of Computing, National University of Singapore*, Advanced Micro Devices, Inc
{dmd, zhaoying, peh, tulika}@comp.nus.edu.sg, Manupa.Karunaratne@amd.com

CGRA: Coarse Grained Reconfigurable Array

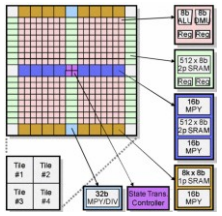
- Power-efficient reconfigurable accelerator
  - Word level reconfigurable
- Simple hardware architecture
  - Array of Processing Elements (PE) inter-connected through a reconfigurable network
  - Each PE has ALU, register file, multiplexers and configuration memory
- Fully software controlled
  - Compiler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



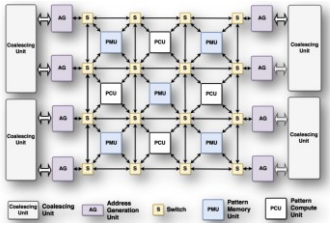
CGRA Architecture



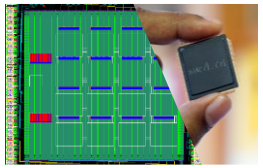
Samsung Reconfigurable Processor [FPT '12]



Renessas DRP [VLSI ' 18]



Sambanova Plasticine [ISCA'17]

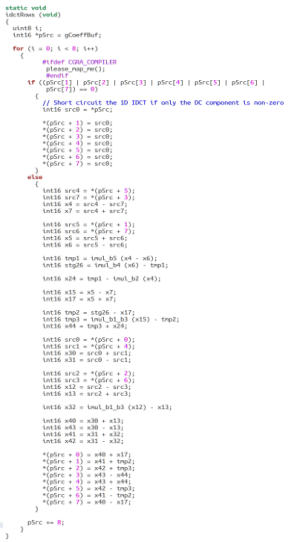


NUS HyCUBE [DAC '17, A-SSCC' 19]

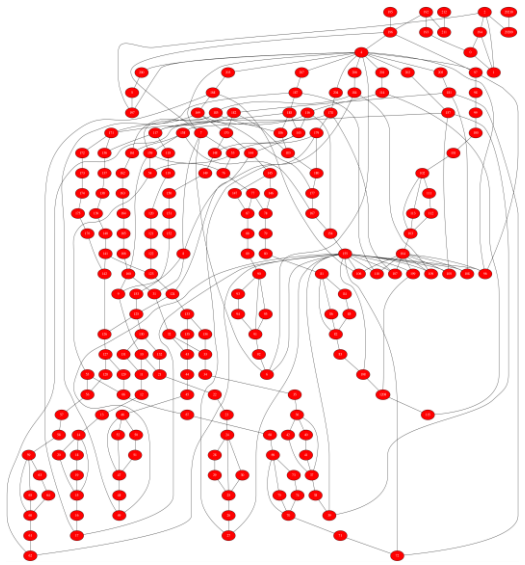
NUS National University of Singapore

Application mapping on CGRA

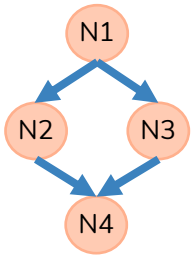
- Target: a loop kernel from applications
- Mapping the dataflow graph (DFG) of the loop body on to the CGRA
  - Placement: assigning DFG operations to ALUs
  - Routing: mapping data signals using wires and registers



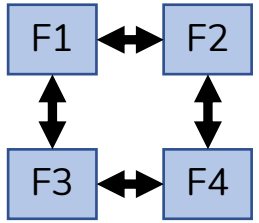
Loop kernel c code



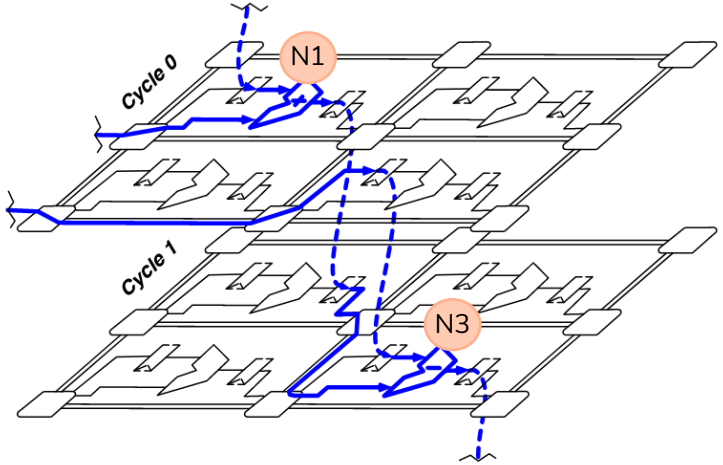
Dataflow graph (DFG) of the loop body



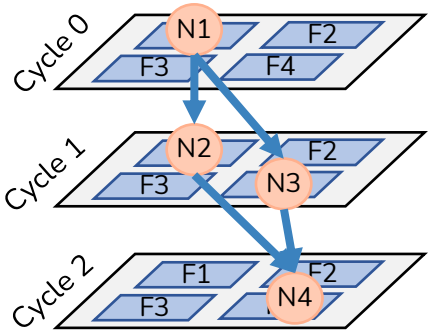
DFG



2x2 CGRA



Placement and routing

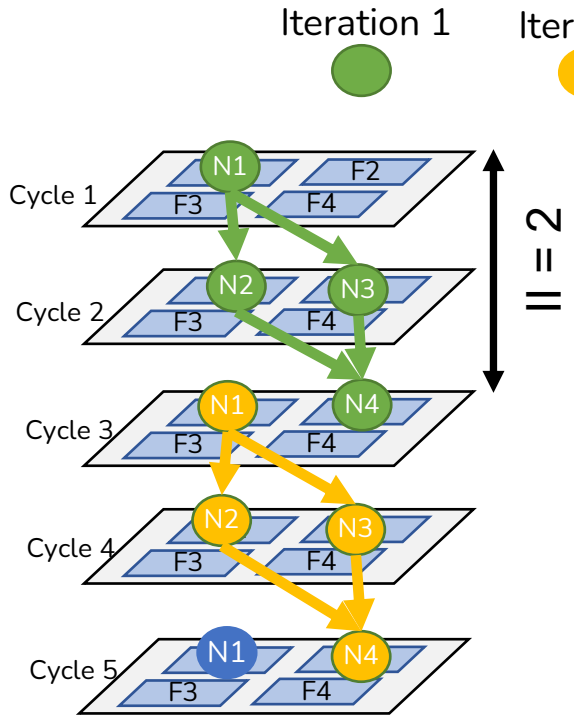


Spatio-temporal mapping

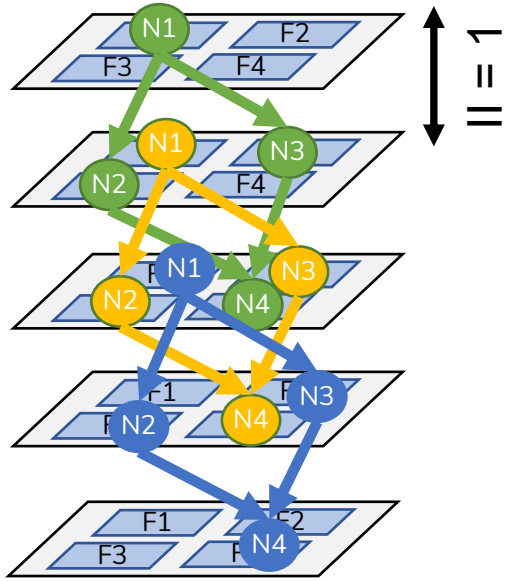
NUS National University of Singapore

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 -> High performance



Low performance mapping with II = 2 cycles



High performance mapping with II = 1 cycles

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

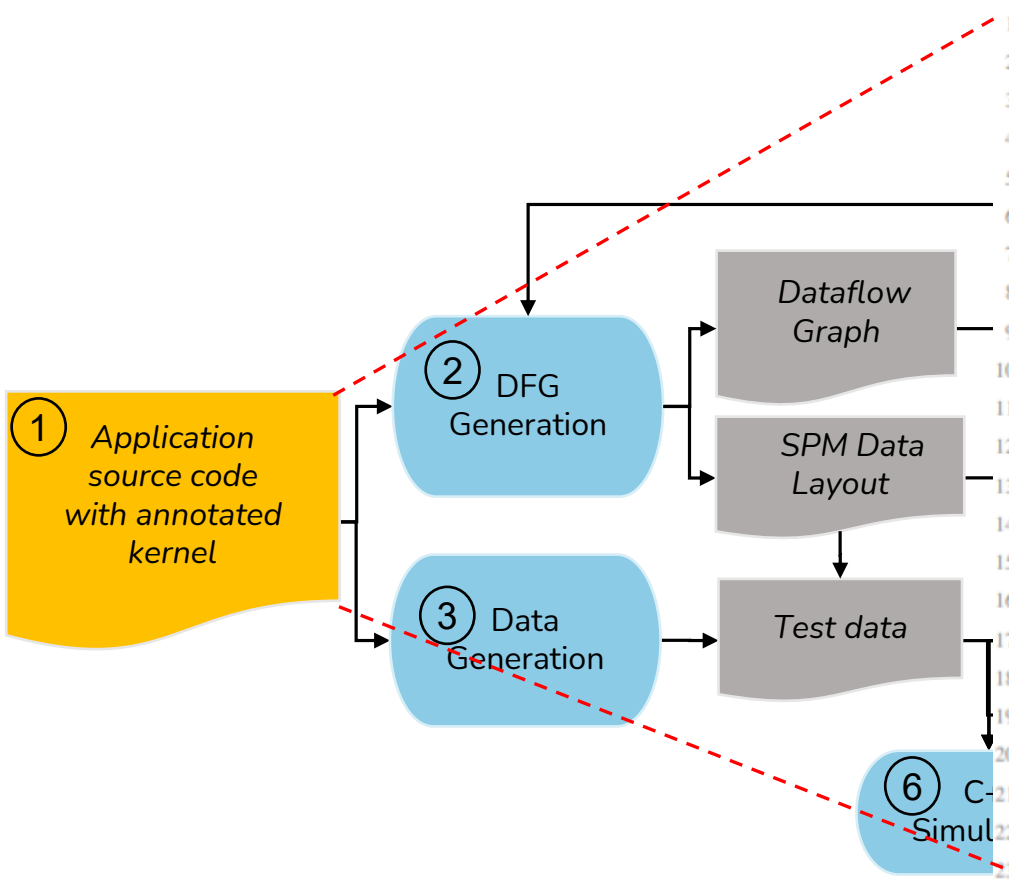
- Fully automated end-to-end CGRA compilation and simulation framework
- Flexible architecture specification language
  - Allows the user to define complex CGRA architectures
- Supports kernels with control divergence and recurrence edges
  - Allows compiling complex application kernels
- Efficient mapping 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	✗	✗	✓	✓	✓
	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	✗	✗	✗	✗	✓

SCAN ME

NUS National University of Singapore

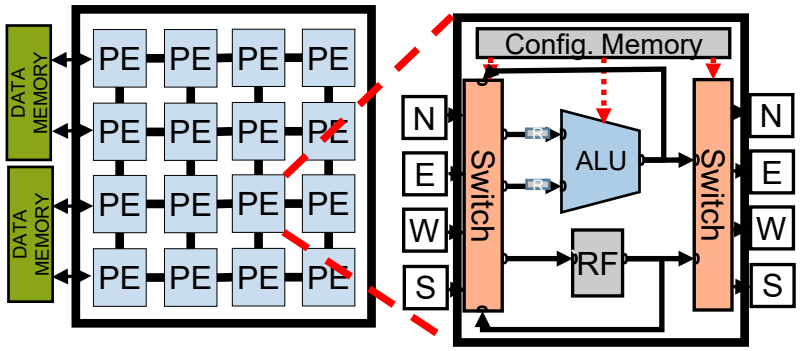
Overview

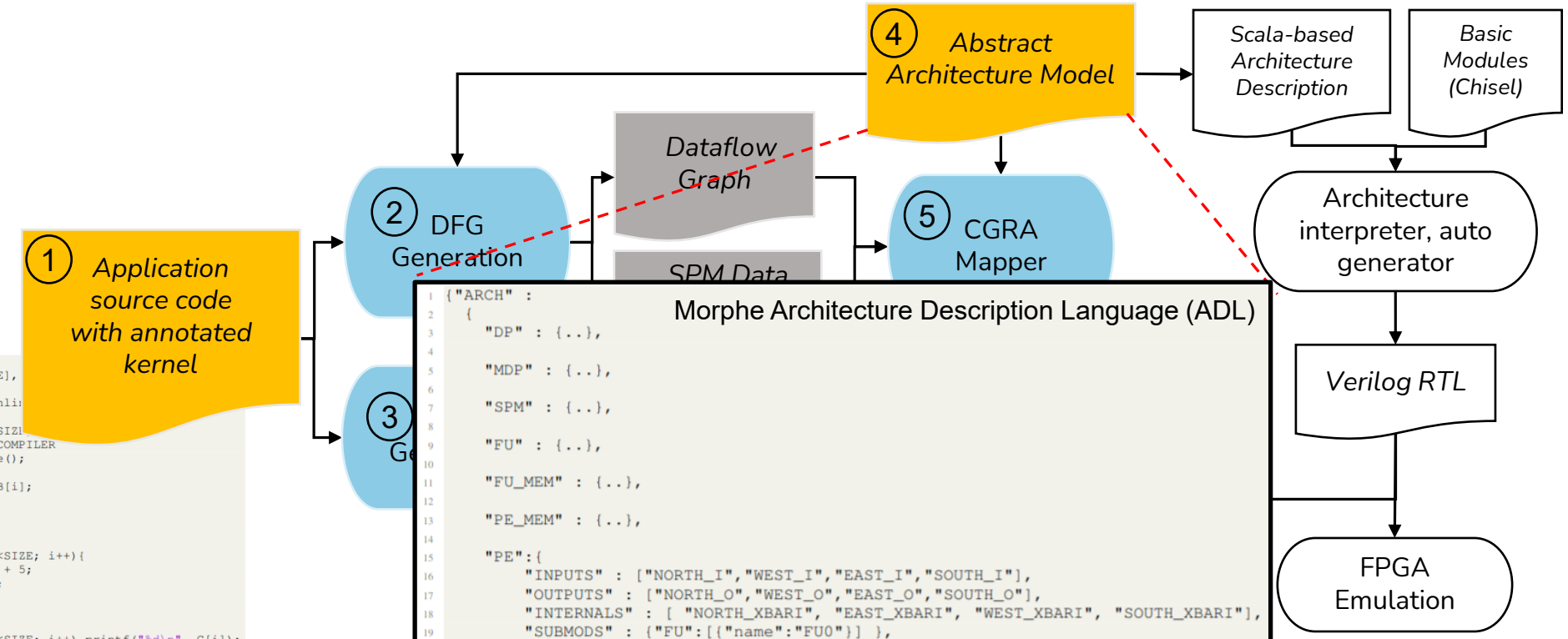



```
#define SIZE  20
int A[SIZE], B[SIZE], C[SIZE];

__attribute__((noinline))
void array_add(){
    for (int i=0;i<SIZE; i++){
      #ifdef CGRA_COMPILER
      please_map_me();
      #endif
      C[i] = A[i]+B[i];
    }
}

int main(){
    for (int i=0;i<SIZE; i++){
      A[i] = i * 2 + 5;
      B[i] = i * 3;
      C[i] = 0;
    }
    array_add();
    for (int i=0;i<SIZE; i++) printf("%d\n", C[i]);
    return 0;
}
```

Target loop

Overview



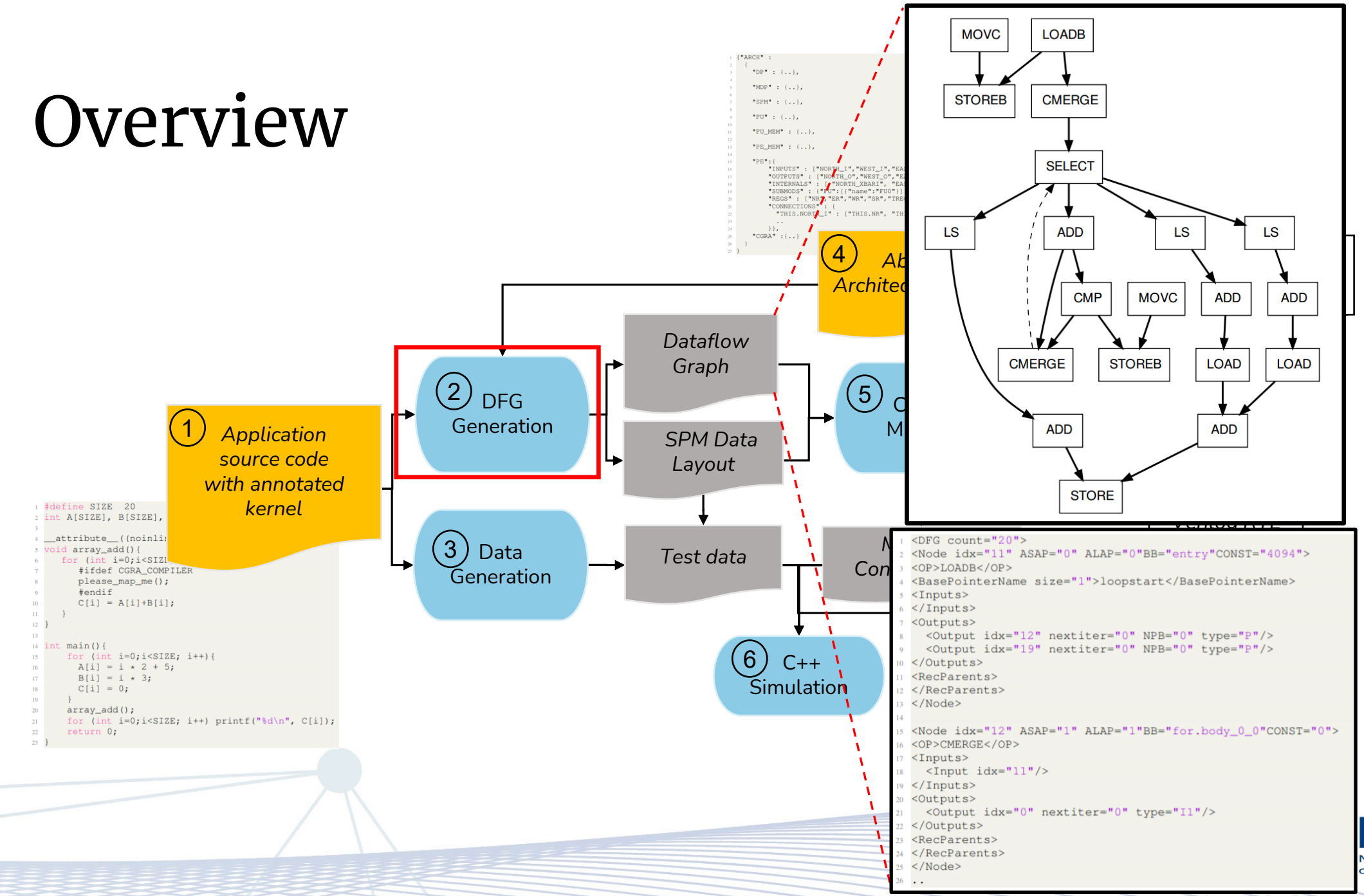



```
#define SIZE  20
int A[SIZE], B[SIZE],

__attribute__((noinli
void array_add(){
   for (int i=0;i<SIZ
      #ifdef CGRA_COMPILER
      please_map_me();
      #endif
      C[i] = A[i]+B[i];
   }
}

int main(){
    for (int i=0;i<SIZE; i++){
      A[i] = i * 2 + 5;
      B[i] = i * 3;
      C[i] = 0;
    }
    array_add();
    for (int i=0;i<SIZE; i++) printf("%d\n", C[i]);
    return 0;
}
```

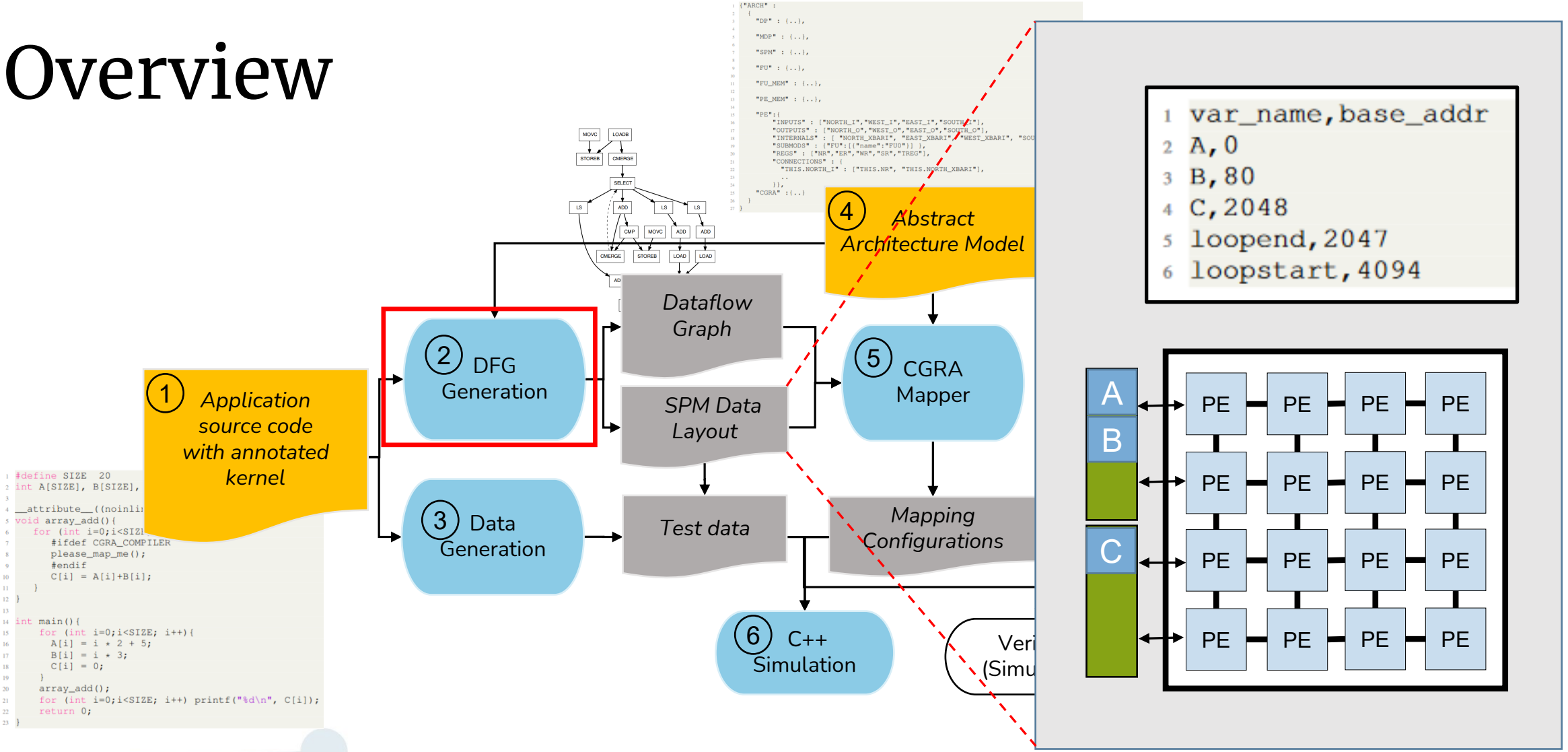
```
{"ARCH" :
  {
    "DP" : {..},

    "MDP" : {..},

    "SPM" : {..},

    "FU" : {..},

    "FU_MEM" : {..},

    "PE_MEM" : {..},

    "PE":{
        "INPUTS" : ["NORTH_I","WEST_I","EAST_I","SOUTH_I"],
        "OUTPUTS" : ["NORTH_O","WEST_O","EAST_O","SOUTH_O"],
        "INTERNALS" : [ "NORTH_XBARI", "EAST_XBARI", "WEST_XBARI", "SOUTH_XBARI"],
        "SUBMODS" : {"FU":[{"name":"FU0"}] },
        "REGS" : ["NR","ER","WR","SR","TREG"],
        "CONNECTIONS" : {
          "THIS.NORTH_I" : ["THIS.NR", "THIS.NORTH_XBARI"],
          ..
        }},
    "CGRA" :{..}
  }
}
```

NUS National University of Singapore

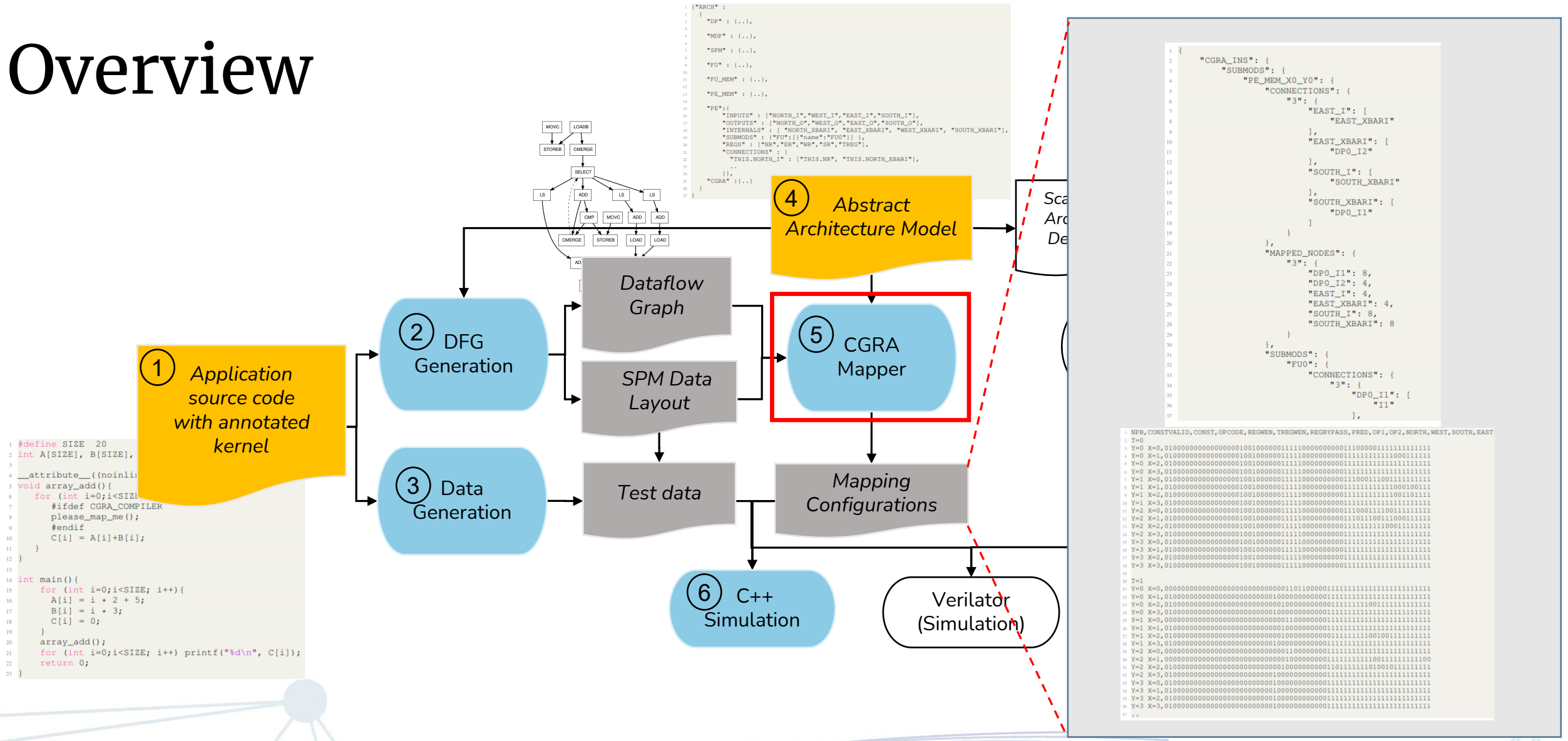
Overview


1 Application source code with annotated kernel
2 DFG Generation
3 Data Generation
4 Ab
Architec
5 C
M
6 C++ Simulation
Dataflow Graph
SPM Data Layout
Test data
MOVC
LOADB
STOREB
CMERGE
SELECT
LS
ADD
LS
LS
CMP
MOVC
ADD
ADD
CMERGE
STOREB
LOAD
LOAD
ADD
ADD
STORE
#define SIZE 20
int A[SIZE], B[SIZE],
__attribute__((noinli
void array_add(){
for (int i=0;i<SIZ
#ifdef CGRA_COMPILER
please_map_me();
#endif
C[i] = A[i]+B[i];
}
int main(){
for (int i=0;i<SIZE; i++){
A[i] = i * 2 + 5;
B[i] = i * 3;
C[i] = 0;
}
array_add();
for (int i=0;i<SIZE; i++) printf("%d\n", C[i]);
return 0;
}
<DFG count="20">
<Node idx="11" ASAP="0" ALAP="0"BB="entry"CONST="4094">
<OP>LOADB</OP>
<BasePointerName size="1">loopstart</BasePointerName>
<Inputs>
</Inputs>
<Outputs>
<Output idx="12" nextiter="0" NPB="0" type="P"/>
<Output idx="19" nextiter="0" NPB="0" type="P"/>
</Outputs>
<RecParents>
</RecParents>
</Node>
<Node idx="12" ASAP="1" ALAP="1"BB="for.body_0_0"CONST="0">
<OP>CMERGE</OP>
<Inputs>
<Input idx="11"/>
</Inputs>
<Outputs>
<Output idx="0" nextiter="0" type="I1"/>
</Outputs>
<RecParents>
</RecParents>
</Node>

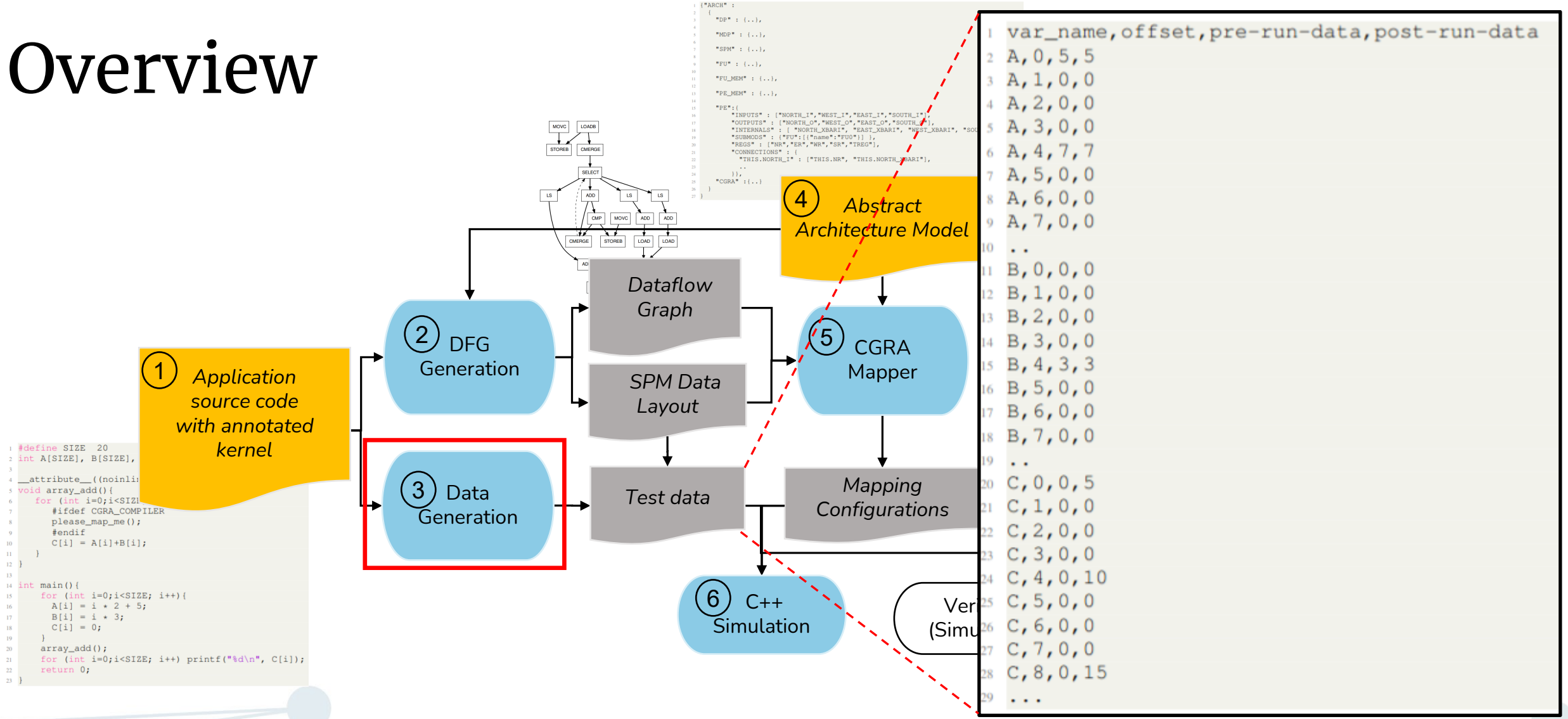
Overview


1 Application source code with annotated kernel
2 DFG Generation
3 Data Generation
4 Abstract Architecture Model
5 CGRA Mapper
6 C++ Simulation
Dataflow Graph
SPM Data Layout
Test data
Mapping Configurations
var_name,base_addr
A,0
B,80
C,2048
loopend,2047
loopstart,4094
A
B
C
PE

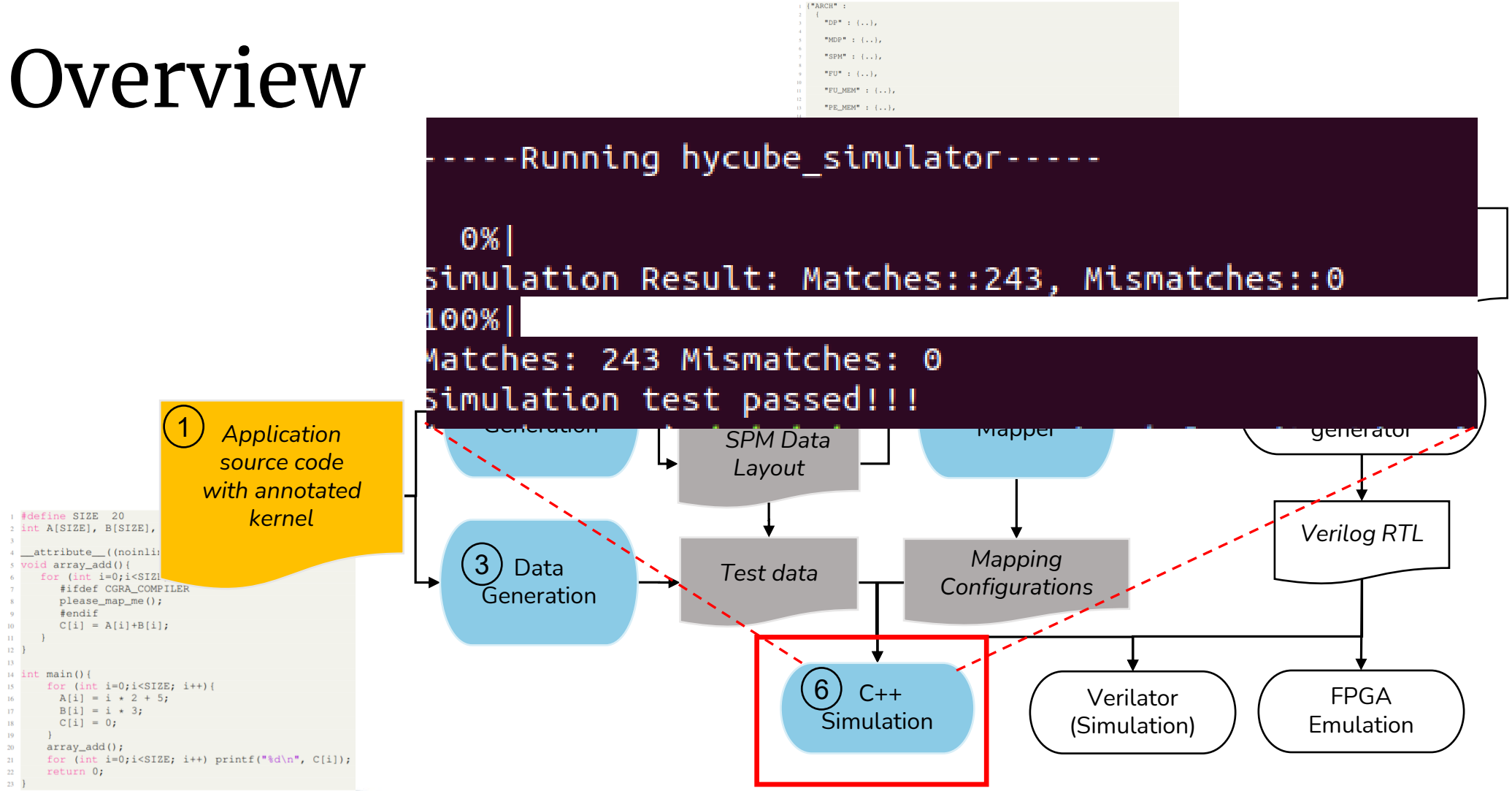
Overview


1 Application source code with annotated kernel
2 DFG Generation
3 Data Generation
4 Abstract Architecture Model
5 CGRA Mapper
6 C++ Simulation
Dataflow Graph
SPM Data Layout
Test data
Mapping Configurations
Verilator (Simulation)

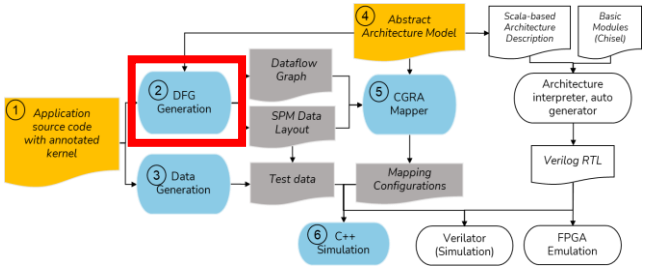
Overview


1 Application source code with annotated kernel
2 DFG Generation
3 Data Generation
4 Abstract Architecture Model
5 CGRA Mapper
6 C++ Simulation
Dataflow Graph
SPM Data Layout
Test data
Mapping Configurations
Ver
(Simu
var_name,offset,pre-run-data,post-run-data
A,0,5,5
A,1,0,0
A,2,0,0
A,3,0,0
A,4,7,7
A,5,0,0
A,6,0,0
A,7,0,0
..
B,0,0,0
B,1,0,0
B,2,0,0
B,3,0,0
B,4,3,3
B,5,0,0
B,6,0,0
B,7,0,0
..
C,0,0,5
C,1,0,0
C,2,0,0
C,3,0,0
C,4,0,10
C,5,0,0
C,6,0,0
C,7,0,0
C,8,0,15
...

Overview

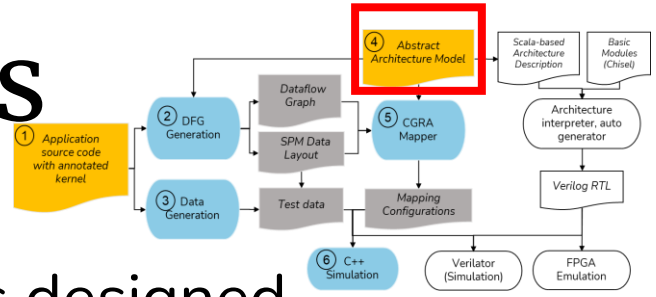

{"ARCH" :
{
"DP" : {..},
"MDP" : {..},
"SPM" : {..},
"FU" : {..},
"FU_MEM" : {..},
"PE_MEM" : {..},
-----Running hycube_simulator-----
0%|
Simulation Result: Matches::243, Mismatches::0
100%|
Matches: 243 Mismatches: 0
Simulation test passed!!!
1 Application source code with annotated kernel
#define SIZE 20
int A[SIZE], B[SIZE],
__attribute__((noinli
void array_add(){
for (int i=0;i<SIZ
#ifdef CGRA_COMPILER
please_map_me();
#endif
C[i] = A[i]+B[i];
}
int main(){
for (int i=0;i<SIZE; i++){
A[i] = i * 2 + 5;
B[i] = i * 3;
C[i] = 0;
}
array_add();
for (int i=0;i<SIZE; i++) printf("%d\n", C[i]);
return 0;
}
SPM Data Layout
Mapper
generator
3 Data Generation
Test data
Mapping Configurations
Verilog RTL
6 C++ Simulation
Verilator (Simulation)
FPGA Emulation

DFG and Data Layout Generation



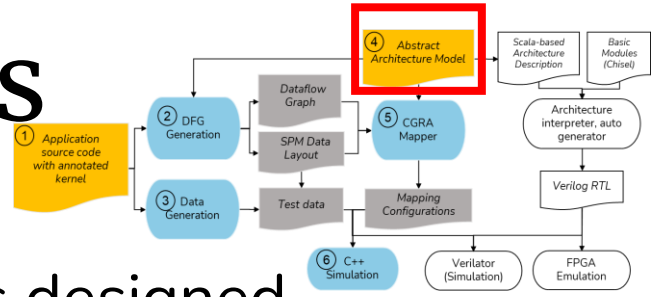
- DFG generation: supports DFG generation for multiple execution paradigms
  - Control-flow handling techniques
    - Partial predication, Full predication, Dual-issue
  - Load store address generation techniques
    - On array address generation, decoupled access/execute address generation
  - Memory 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

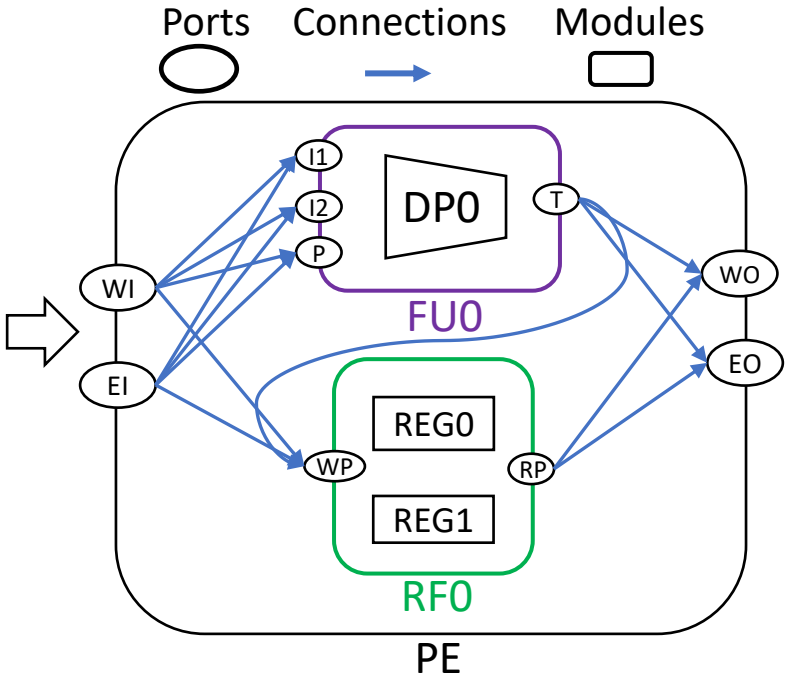
Abstract Architecture Specifications



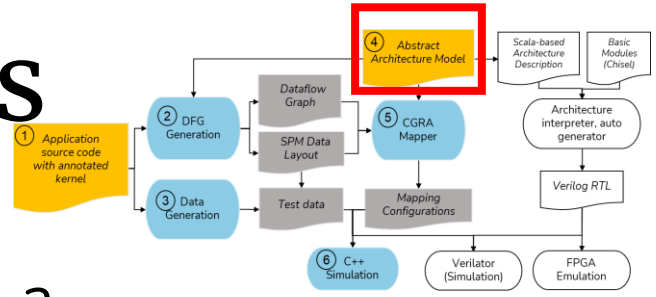
- Morpher abstract Architecture Description Language (ADL) is designed to cover diverse CGRA architectures

```
"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.RP0" : ["FU0.I1","FU0.I2","FU0.P","THIS.EO","THIS.WO"],
       "FU0.T" : ["RF0.WP","THIS.EO","THIS.WO"]
     }
},
```

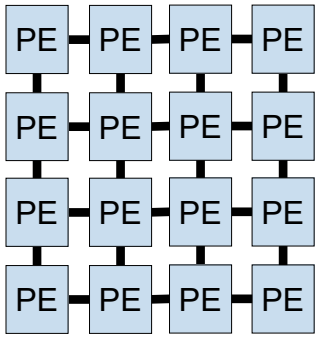
PE described in Morpher ADL



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

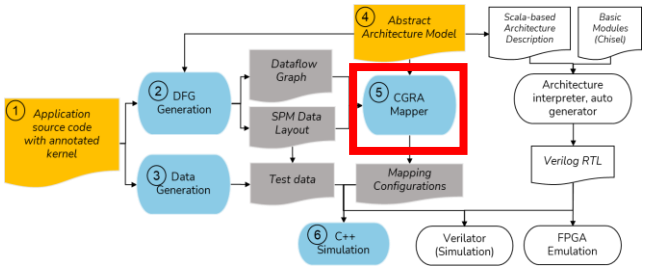


PEs connected in a 2-D “GRID” Pattern

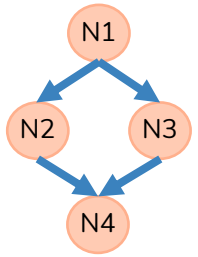
```
"CGRA" :{
  "SUBMODS" :[
     { "PATTERN" : "GRID",
       "DIMS" : {"X" : 4,"Y" : 4},
       "MODS" : [
         {"X":0,"Y":0,"MOD":"PE_MEM"}, {"X":1,"Y":0,"MOD":"PE"},  {"X":2,"Y":0,"MOD":"PE"}, {"X":3,"Y":0,"MOD":"PE"},
         {"X":0,"Y":1,"MOD":"PE_MEM"}, {"X":1,"Y":1,"MOD":"PE"},  {"X":2,"Y":1,"MOD":"PE"}, {"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_O", "TO_X" : "X", "TO_Y" : "Y-1", "TO_PORT" : "SOUTH_I"},
         {"FROM_X" : "X", "FROM_Y" : "Y", "FROM_PORT" : "EAST_O",  "TO_X" : "X+1", "TO_Y" : "Y", "TO_PORT" : "WEST_I"},
         {"FROM_X" : "X", "FROM_Y" : "Y", "FROM_PORT" : "WEST_O",  "TO_X" : "X-1", "TO_Y" : "Y", "TO_PORT" : "EAST_I"},
         {"FROM_X" : "X", "FROM_Y" : "Y", "FROM_PORT" : "SOUTH_O", "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)

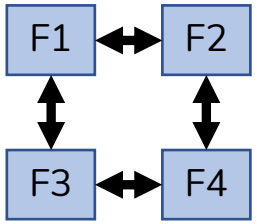
CGRA Mapper



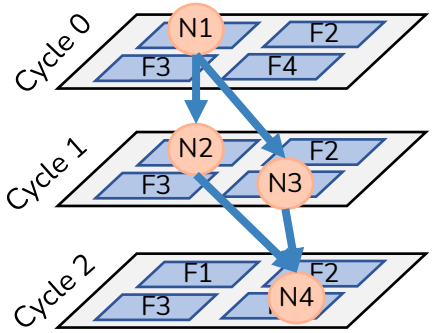
- Mapper generates the configurations of hardware elements (ALU, Mux, RF read/write)
  - Goal of the mapper is to generate a mapping with the lowest possible II (Minimum II)
- Algorithm:
  - Minimum II is calculated
    - based on the CGRA array size, number of DFG nodes, and the recurrence dependencies 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



DFG

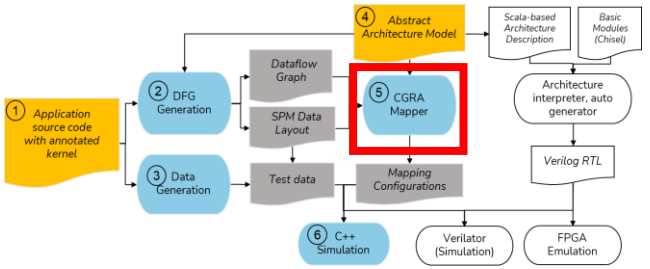


2x2 CGRA



Spatio-temporal mapping

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]

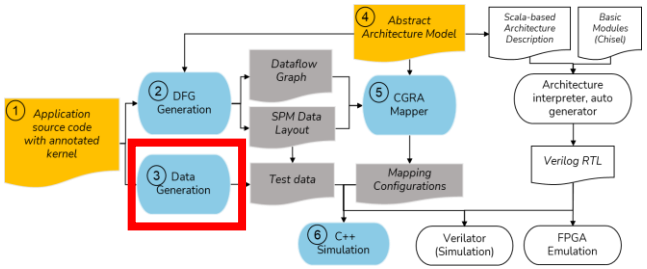
[1] Z. Li, D. Wu, D. Wijerathne, and T. Mitra, "LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*

[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

NUS National University of Singapore

Test Data Generation



- Instrument, i.e., insert data recording functions to the C source code to capture live- rnel
- Instrumente to record th
- Simulator us
  - Live-in value
  - Live-out vari

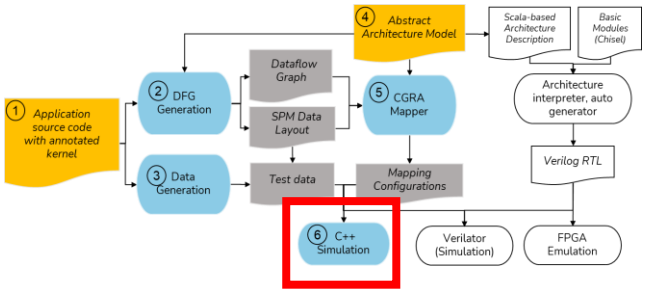
```
define dso_local void @array_add() local_unnamed_addr #0 {
entry:
  call void @loopStart(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @0, i32 0, i32 0))
  call void @LiveInReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @1, i32 0, i32 0)  i8* bitc
    ([20 x i32]* @C to i8*), i32 80)
  call void @LiveInReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @2, i32 0, i32 0)  i8* bitc
    ([20 x i32]* @A to i8*), i32 80)
  call void @LiveInReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @3, i32 0, i32 0)  i8* bitc
    ([20 x i32]* @B to i8*), i32 80)
  br label %for.body

for.body:                                         ; preds = %for.body, %entry
  %i.08 = phi i32 [ 0, %entry ], [ %inc, %for.body ]
  %arrayidx = getelementptr inbounds [20 x i32], [20 x i32]* @A, i32 0, i32 %i.08
  %manupa0 = load i32, i32* %arrayidx, align 4, !tbaa !3
  %arrayidx1 = getelementptr inbounds [20 x i32], [20 x i32]* @B, i32 0, i32 %i.08
  %manupa1 = load i32, i32* %arrayidx1, align 4, !tbaa !3
  %add = add nsw i32 %manupa1, %manupa0
  %arrayidx2 = getelementptr inbounds [20 x i32], [20 x i32]* @C, i32 0, i32 %i.08
  store i32 %add, i32* %arrayidx2, align 4, !tbaa !3
  %inc = add nuw nsw i32 %i.08, 1
  %exitcond = icmp eq i32 %inc, 20
  br i1 %exitcond, label %for.cond.cleanup, label %for.body

for.cond.cleanup:                                 ; preds = %for.body
  call void @LiveOutReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @1, i32 0, i32 0  i8* bit
    ([20 x i32]* @C to i8*), i32 80)
  call void @LiveOutReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @2, i32 0, i32 0  i8* bit
    ([20 x i32]* @A to i8*), i32 80)
  call void @LiveOutReport(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @3, i32 0, i32 0  i8* bit
    ([20 x i32]* @B to i8*), i32 80)
  call void @loopEnd(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @4, i32 0, i32 0))
  ret void

}
```

Live-in variable recording functions

Target loop

Live-out variable recording functions

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

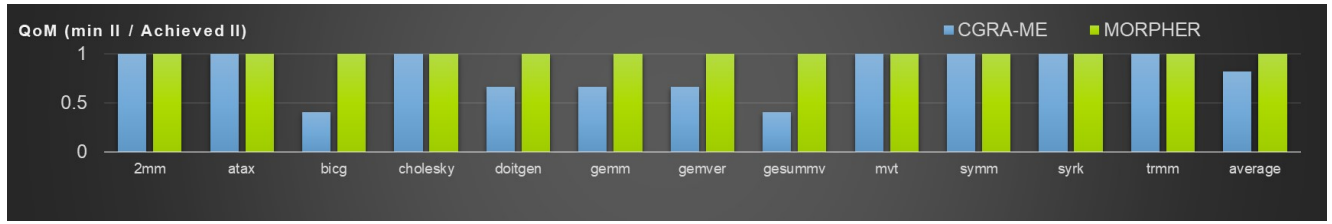
Open-source Artifact Demonstration

Experimental Study

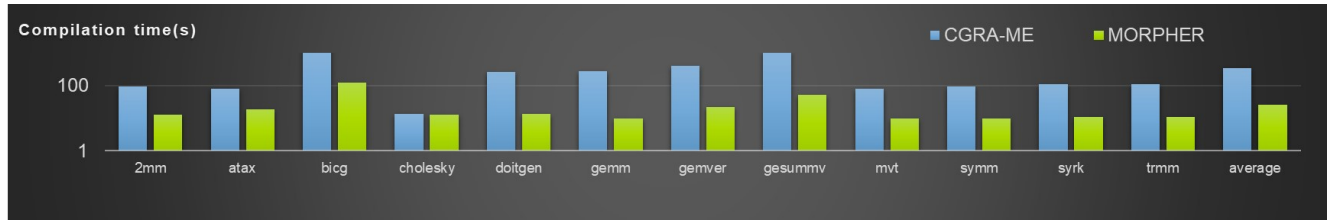
1. Mapping quality and compilation time
2. Map real applications kernels with automated verification
3. The ability to model diverse CGRA architectures

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.



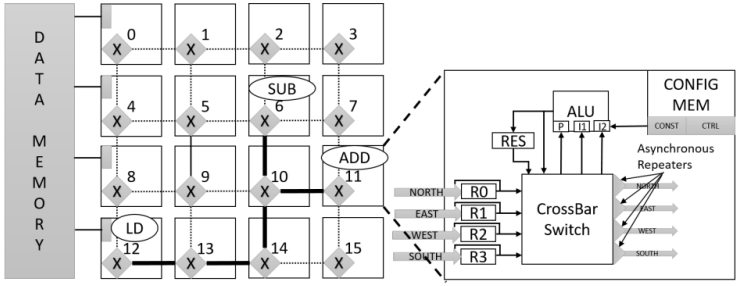
Quality of Mapping (min II/ Achieved II)



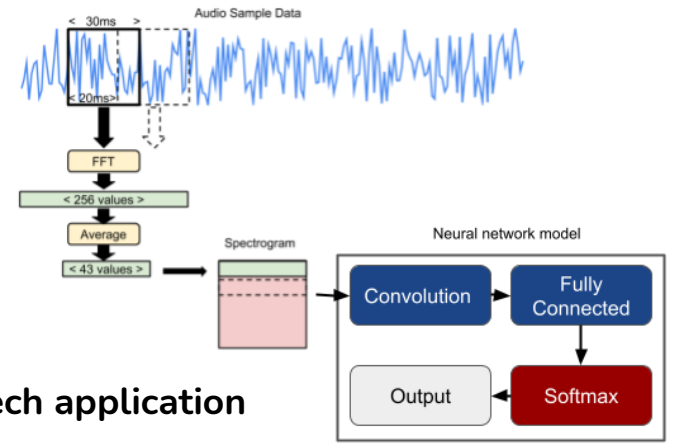
Compilation Time (s)

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



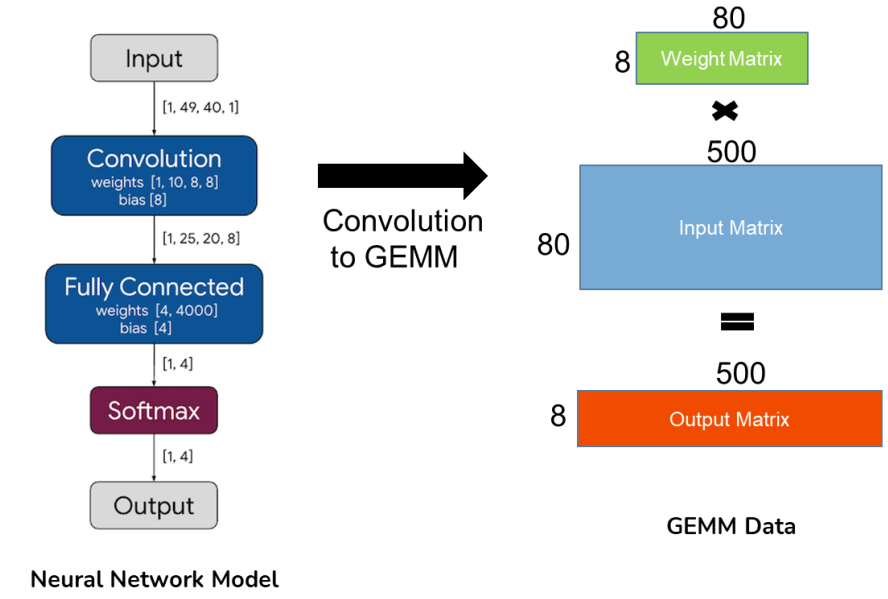
HyCUBE CGRA



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

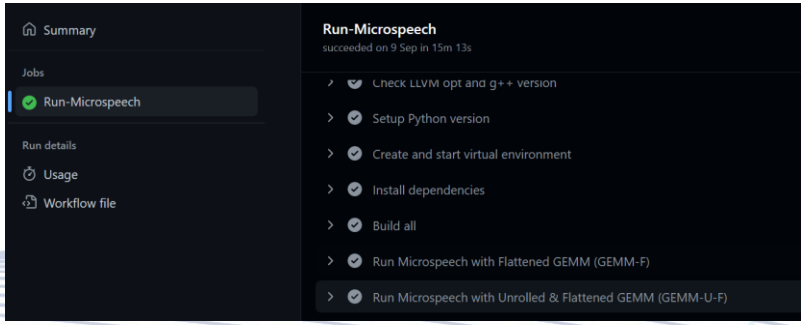
- We consider three variations of the GEMM kernel
  - GEMM
  - GEMM-U: unrolled four times
  - GEMM-U-F: unrolled and flattened
    - Flattening introduces conditional statements inside the loop body and adds a long recurrence edge
    - Flattened kernel does not have invocation overheads
- Results:
  - 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

```
for (i=0;i<R1; i++)
 for (j=0;j<C2; j++)
  for (k=0;k<C1; k++): //map this (GEMM)
   O[i][j] += W[i][k]* I[k][j];

for (i=0;i<R1; i++)
 for (j=0;j<C2; j++)
  for (k=0;k<C1; k=k+4): //map this (GEMM-U)
   O[i][j] += W[i][k]* I[k][j]
            + W[i][k+1]* I[k+1][j]
            + W[i][k+2]* I[k+2][j]
            + W[i][k+3]* I[k+3][j];

for (n=0;n<R1*C2*C1; n++){: //map this (GEMM-U-F)
   O[i][j] += W[i][k]* I[k][j]
            +  W[i][k+1]* I[k+1][j]
            +  W[i][k+2]* I[k+2][j]
            +  W[i][k+3]* I[k+3][j];
   k+=4;
   if(k+1>=C1) {k=0; ++j;}
   if(j==C2) {j=0; ++i;}
}
```

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

NUS National University of Singapore

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

NUS National University of Singapore

Thank you!

NUS
National University
of Singapore