

Tutorial Schedule

9:00 – 9:30	Antonino Tumeo	Agile Hardware Design for Complex Data Science Applications: Opportunities and Challenges.
9:30 – 10:00	Fabrizio Ferrandi	Bambu: An Open-Source Research Framework for the High-Level Synthesis of Complex Applications.
10:00 – 10:30	Antonino Tumeo	SODA-OPT: Enabling System-Level Design in MLIR for HLS and Beyond. Hands-on: From DNN Models to ASIC Devices with SODA-OPT
10:30 – 11:00		Coffee Break
11:00 – 12:00	Serena Curzel Michele Fiorito	Hands-on: Productive High-Level Synthesis with Bambu, Compiler Based Optimizations, Tuning and Customization of Generated Accelerators
12:00 - 12:30	Antonino Tumeo & Fabrizio Ferrandi	New features in SODA-OPT and Bambu



fast machine learning
for science

AXI4MLIR: Co-designing Accelerators With Instructions and Automatic Driver Code Generation

Tutorial: SODA Synthesizer: Accelerating Artificial Intelligence
Applications with an End-to-End Silicon Compiler

September 1, 2025

Antonino Tumeo

Nicolas Bohm Agostini

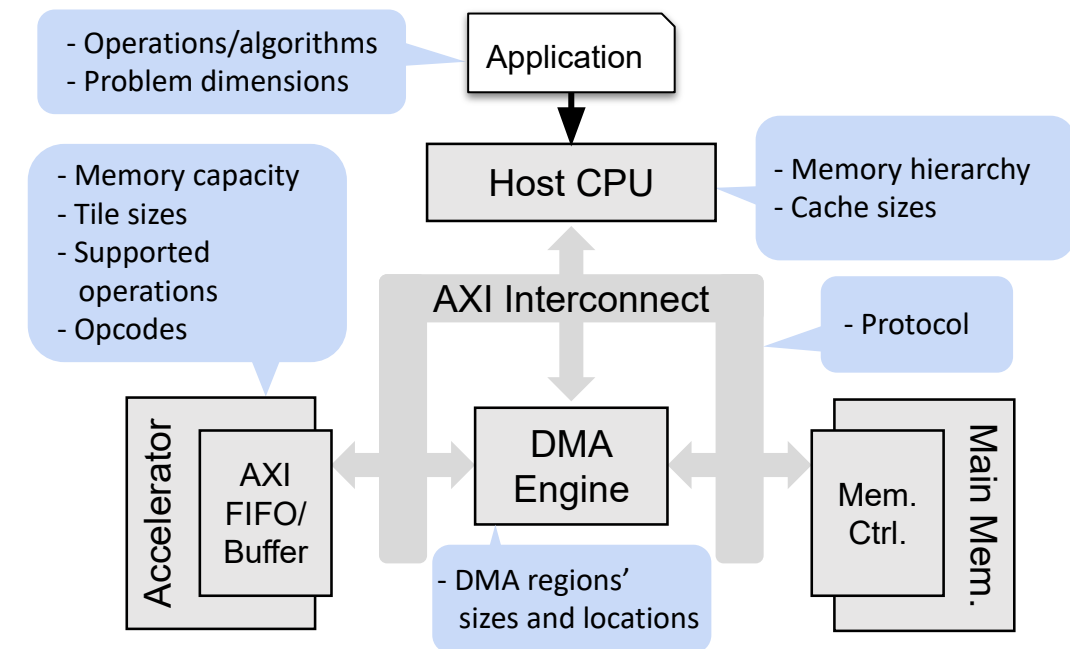
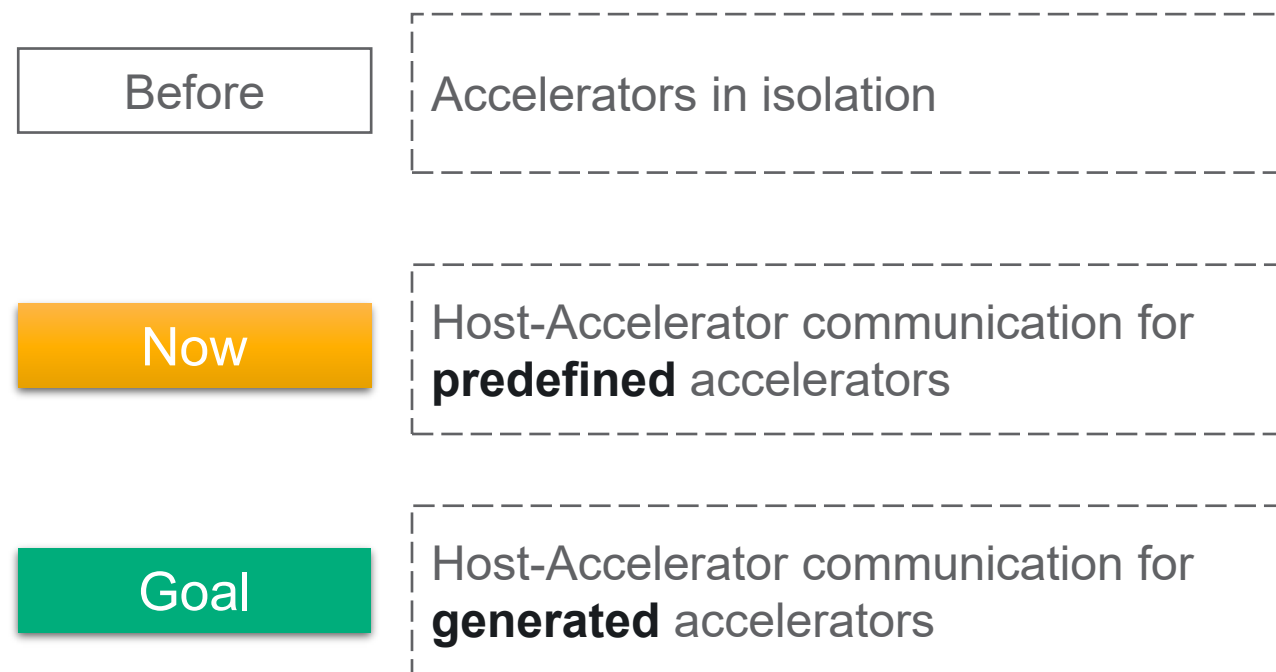


PNNL is operated by Battelle for the U.S. Department of Energy



AXI4MLIR: Automating Host-Accelerator Communication

- Work in collaboration with Northeastern University, University of Glasgow, and Universidad Católica de Murcia
- AXI4MLIR implements **host code generation** to drive accelerators connected through an AXI-Stream Interface
 - Provides Efficient Host-Accelerator execution flow



[N. Bohm Agostini, J. Haris, P. Gibson, M. Jayaweera, N. Rubin, A. Tumeo, K. Abellan, J. Cano, D. Kaeli. AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators. CGO 2024]

How to provide support for accelerators with instructions?

Implement
Conventional Compiler Support

Issues

- Time consuming

Advantages

- Extensions can be used by many frontends
- Errors at compilation time

Implement
Custom Library And Modify Application

Issues

- Lack of generality
- 1 implementation per app-acc
- Requires modification of the original application

Advantages

- Fast design prototyping

Can we provide Compiler Support with Agile Design Prototyping?

AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators

- **MLIR extensions to describe** custom accelerators with arbitrary instructions

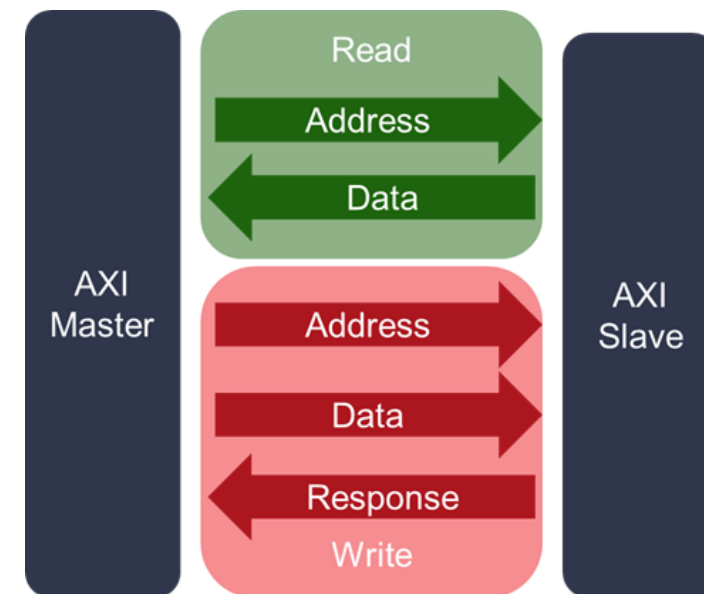


<https://mlir.llvm.org/>

Lattner, C., et al. "MLIR: Scaling compiler infrastructure for domain specific computation." CGO 2021

AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators

- **MLIR extensions to describe** custom accelerators with arbitrary instructions
- Simple Host-Accelerator **communication abstraction** and **AXI library implementation**



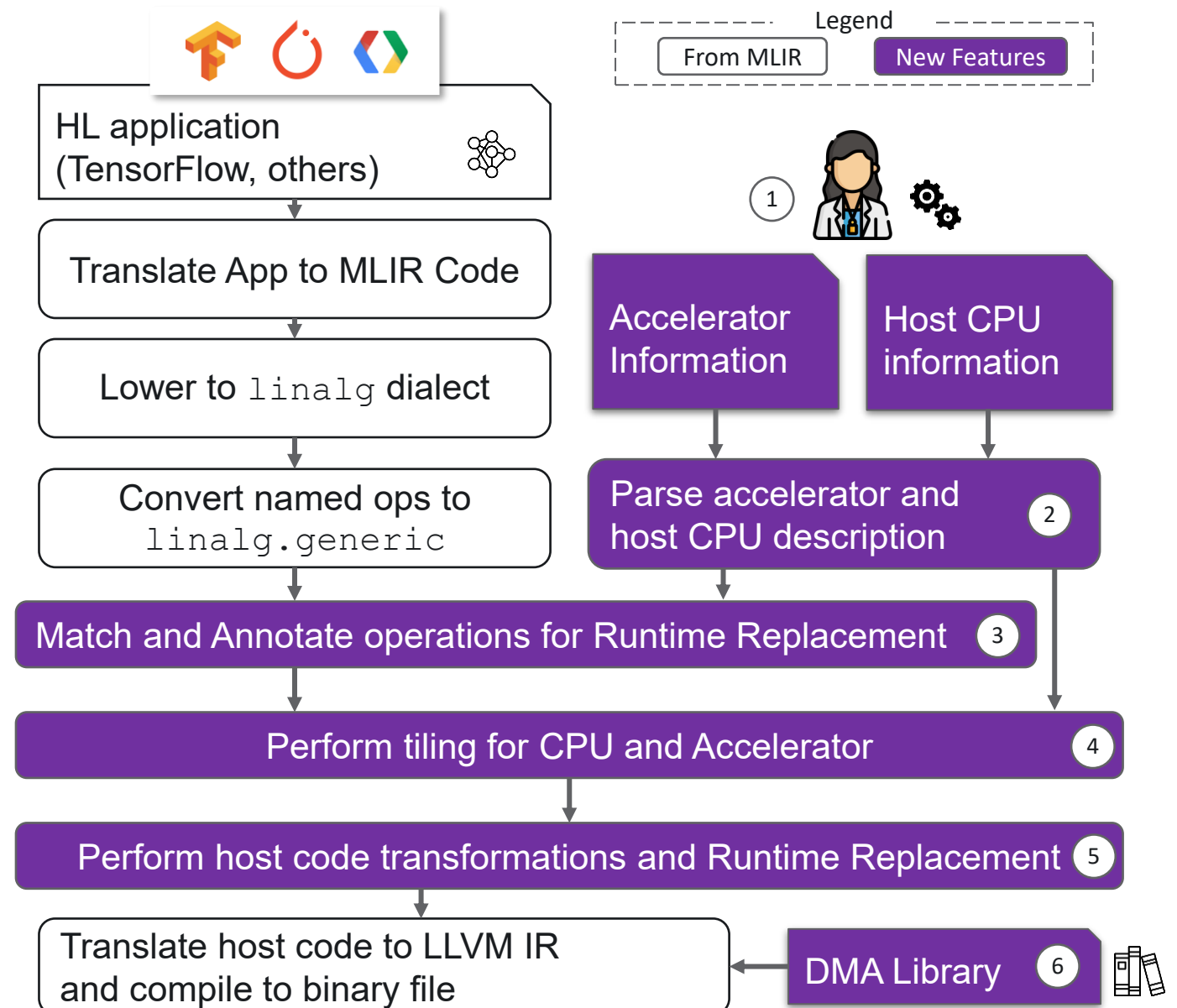
From: Werbrouck, F. "AXI Basics 1 - Introduction to AXI". 2023.
Available at: <https://support.xilinx.com/s/article/1053914>

AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators

AXI4MLIR Approach

- **MLIR extensions** to describe custom accelerators with arbitrary instructions
- Simple Host-Accelerator **communication abstraction** and **AXI library implementation**
- Implements **host code generation** to drive accelerators connected through an AXI-Stream Interface

<https://github.com/AXI4MLIR/axi4mlir>



Contributions

- An MLIR dialect to abstract **send** and **recv** transactions of data packages to the accelerator
 - The **accel** dialect:
 - ❖ send, recv, send_literal, send_dim, send_idx
- A communication library
 - Lightweight DMA Engine Library

`<final offset> = accel.send(<memref or subview>, <dma offset>)`

```
%sA=memref.subview %A[%i, %j][%tile_size_I, %tile_size_J][%c1, %c1]
call @copy_to_dma_region (%sA, offset=4bytes)
call @dma_start_send(size=%tile_size_I*%tile_size_J*4bytes, offset=4bytes)
call @dma_wait_send_completion()
```

Contributions

- An MLIR dialect to abstract **send** and **recv** transactions of data packages to the accelerator
 - The **accel** dialect:
 - ❖ `send`, `recv`, `send_literal`, `send_dim`, `send_idx`
- A communication library
 - Lightweight DMA Engine Library

`<final offset> = accel.send(<memref or subview>, <dma offset>)`

```
%sA=memref.subview %A[%i, %j][%tile_size_I, %tile_size_J][%c1, %c1]
call @copy_to_dma_region (%sA, offset=4bytes)
call @dma_start_send(size=%tile_size_I*%tile_size_J*4bytes, offset=4bytes)
call @dma_wait_send_completion()
```

- A way to describe the custom ISA and how to use them in an MLIR operation
 - OpcodeMap** and **OpcodeFlow**

OpcodeMap and OpcodeFlow

```
1 #matmul_accel_trait = {
2
3   dma_init_config = {
4     id = 0x0,
5     inputAddress = 0x42,     inputBufferSize = 0xFF00,
6     outputAddress = 0xFF42, outputBufferSize = 0xFF00,
7   },
8
9   // Opcodes sent once. Tokens defined in opcode_map
10  init_opcodes = init_opcodes < (reset) >,
11
12  accel_dim = map<(m, n, k) -> (4, 4, 4)>, // Tiling
13
14  // Permutation and who can be stationary
15  permutation_map = affine_map<(m, n, k) -> (m, k, n)>,
16
17  opcode_map = opcode_map < // Valid Opcodes
18    sA      = [send_literal(0x22), send(0)],
19    sB      = [send_literal(0x23), send(1)],
20    sB_cC_rC = [send_literal(0x25), send(1), recv(2)],
21    reset   = [send_literal(0xFF)],
22  >,
23
24  // Flow to implement. Tokens defined in opcode_map
25  opcode_flow = opcode_flow < (sA (sB_cC_rC)) >,
26
27  // Includes original indexing_maps, iterator_types ...
28 }
```

New Infrastructure to Describe Accelerators' ISA

```

1 #matmul_accel_trait = {
2
3   dma_init_config = {
4     id = 0x0,
5     inputAddress = 0x42,    inputBufferSize = 0xFF00,
6     outputAddress = 0xFF42, outputBufferSize = 0xFF00,
7   },
8
9   // Opcodes sent once. Tokens defined in opcode_map
10  init_opcodes = init_opcodes < (reset) >,
11
12  accel_dim = map<(m, n, k) -> (4, 4, 4)>, // Tiling
13
14  // Permutation and who can be stationary
15  permutation_map = affine_map<(m, n, k) -> (m, k, n)>,
16
17  opcode_map = opcode_map < // Valid Opcodes
18    sA      = [send_literal(0x22), send(0)],
19    sB      = [send_literal(0x23), send(1)],
20    sB_cC_rC = [send_literal(0x25), send(1), recv(2)],
21    reset   = [send_literal(0xFF)],
22  >,
23
24  // Flow to implement. Tokens defined in opcode_map
25  opcode_flow = opcode_flow < (sA (sB_cC_rC)) >,
26
27  // Includes original indexing_maps, iterator_types ...
28 }

```

Syntax of New MLIR Attributes

opcode_map to describe available opcodes and respective actions

```

1 opcode_dict ::=
2   `opcode_map` `<` opcode-entry (``,` opcode-entry)* `>`
3
4 opcode_entry ::= (bare-id | string-literal) `=` opcode_list
5
6 opcode_list ::= `[` opcode_expr (``,` opcode_expr)* `]`
7
8 opcode_expr ::= `send` `(` bare-id `)`
9               | `send_literal` `(` integer-literal `)`
10              | `send_dim` `(` bare-id `)`
11              | `send_idx` `(` bare-id `)`
12              | `recv` `(` bare-id `)`

```

opcode_flow to describe when to use the opcodes

```

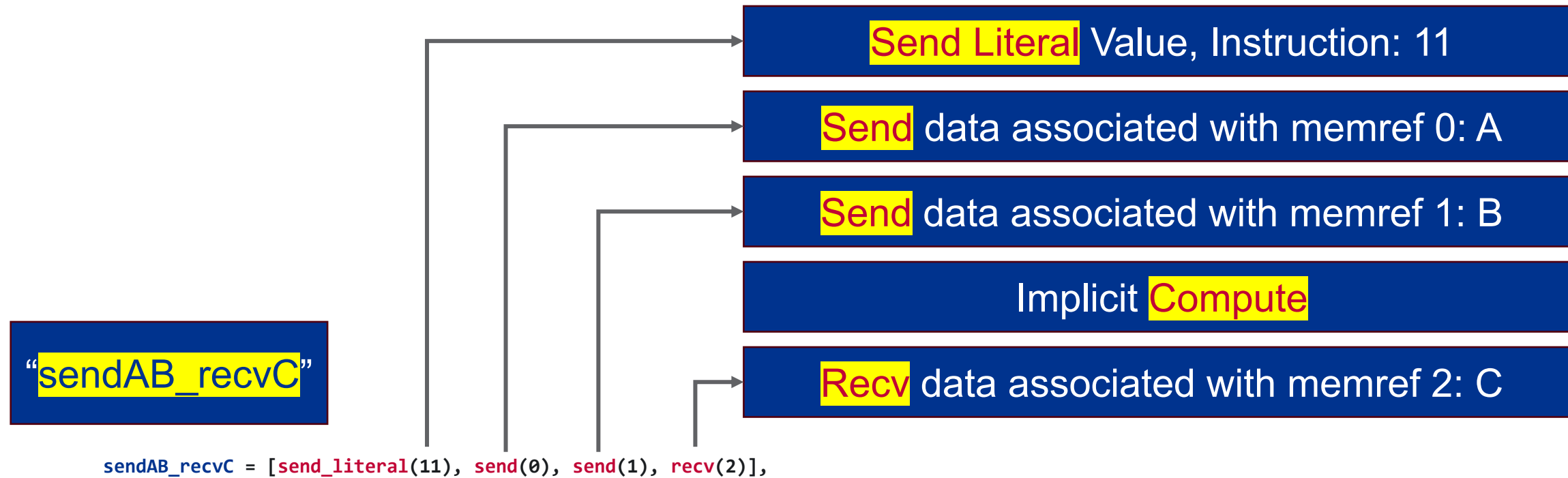
1 opcode_flow ::= `opcode_flow` `<` flow_expr >
2
3 flow_expr ::= `(` flow_expr `)` | bare_id (` ` bare_id)*

```

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
```

```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```



>

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
```

```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```

(sendAB_recvC)

Opcode “sendAB_recvC” can be used to transmit its commands in the innermost loop

```
sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
```

```
scf.for %mm = %c0 to %c16 step %c4 {
  scf.for %nn = %c0 to %c8 step %c4 {
    scf.for %kk = %c0 to %c32 step %c4 {
      %0 = accel.sendLiteral %c11_i32, %c0_i32 : (i32, i32) -> i32
      %1 = memref.subview %A_arg0[%mm, %kk] ...
      %2 = memref.cast %1 : memref<4x4xi32, #map0> to memref<?x?xi32, #map0>
      %3 = accel.send %2, %0 : (memref<?x?xi32, #map0>, i32) -> i32

      %5 = memref.subview %B_arg1[%kk, %nn] ] ...
      %6 = memref.cast %5 : memref<4x4xi32, #map1> to memref<?x?xi32, #map1>
      %7 = accel.send %6, %4 : (memref<?x?xi32, #map1>, i32) -> i32

      %10 = memref.subview %C_arg2[%mm, %nn] ...
      %11 = memref.alloca() : memref<4x4xi32>
      %12 = accel.recv %11, %9 : (memref<4x4xi32>, i32) -> i32

      // Accumulation of received C (%12) on the CPU subview of C (%10)
      linalg.generic ... ins(%11 : memref<4x4xi32>) outs(%10 : memref<4x4xi32, #map1>) {...}
    }
  }
}
```

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
```

```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```

(sendAB_recvC)

Opcode "sendAB_recvC" can be used to transmit its commands in the innermost loop

```
sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
```

```

scf.for %mm = %c0 to %c16 step %c4 {
  scf.for %nn = %c0 to %c8 step %c4 {
    scf.for %kk = %c0 to %c32 step %c4 {
      %0 = accel.sendLiteral %c11_i32, %c0_i32 : (i32, i32) -> i32
      %1 = memref.subview %A_arg0[%mm, %kk] ...
      %2 = memref.cast %1 : memref<4x4xi32, #map0> to memref<?x?xi32, #map0>
      %3 = accel.send %2, %0 : (memref<?x?xi32, #map0>, i32) -> i32

      %5 = memref.subview %B_arg1[%kk, %nn] ] ...
      %6 = memref.cast %5 : memref<4x4xi32, #map1> to memref<?x?xi32, #map1>
      %7 = accel.send %6, %4 : (memref<?x?xi32, #map1>, i32) -> i32

      %10 = memref.subview %C_arg2[%mm, %nn] ...
      %11 = memref.alloca() : memref<4x4xi32>
      %12 = accel.recv %11, %9 : (memref<4x4xi32>, i32) -> i32

      // Accumulation of received C (%12) on the CPU subview of C (%10)
      linalg.generic ... ins(%11 : memref<4x4xi32>) outs(%10 : memref<4x4xi32, #map1>) {...}
    }
  }
}

```

sendAB_recvC

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<  
  sendA = [send_literal(1), send(0)],  
  
  sendB = [send_literal(2), send(1)],  
  
  compute = [send_literal(4)],  
  
  recvC = [send_literal(8), recv(2)],  
  
  sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
```

“Finer grained”
opcodes support
additional dataflows
Introducing:
sendA, sendB,
compute, recvC

```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
```

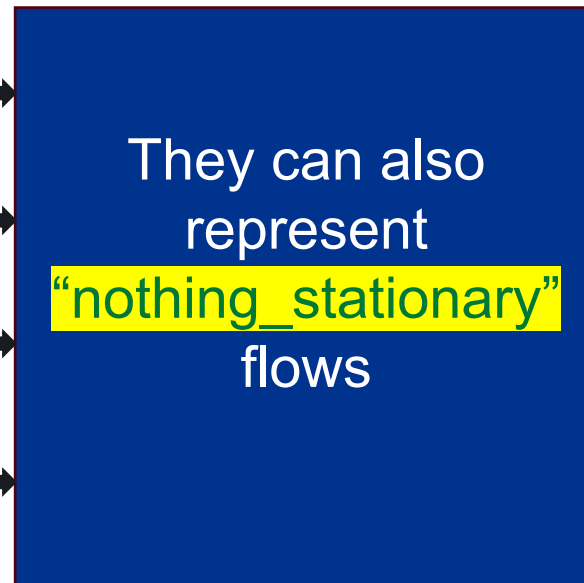
```
sendA = [send_literal(1), send(0)],
```

```
sendB = [send_literal(2), send(1)],
```

```
compute = [send_literal(4)],
```

```
recvC = [send_literal(8), recv(2)],
```

```
sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
```



```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```

```
#nothing_stationary1 = opcode_flow<(sendA sendB compute recvC)>
```

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
```

```
  sendA = [send_literal(1), send(0)],
```

```
  sendB = [send_literal(2), send(1)],
```

```
  compute = [send_literal(4)],
```

```
  recvC = [send_literal(8), recv(2)],
```

```
sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
```

Or the
"A_stationary"
flow

```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```

```
#nothing_stationary1 = opcode_flow<(sendA sendB compute recvC)>
```

```
#A_stationary_option0 = opcode_flow<(sendA (sendB compute recvC))>
```

(sendA (sendB compute recvC))

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
```

```
sendA = [send_literal(1), send(0)],
```

```
sendB = [send_literal(2), send(1)],
```

```
compute = [send_literal(4)],
```

```
recvC = [send_literal(8), recv(2)],
```

```
sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
```

Or the
"A_stationary"
flow

```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```

```
#nothing_stationary1 = opcode_flow<(sendA sendB compute recvC)>
```

```
#A_stationary_option0 = opcode_flow<(sendA (sendB compute recvC))>
```

(sendA (sendB compute recvC))

```
scf.for %mm = %c0 to %c16 step %c4 {
  scf.for %nn = %c0 to %c8 step %c4 {
    %0 = accel.sendLiteral %c1_i32, %c0_i32 : (i32, i32) -> i32
    %1 = memref.subview %A_arg0[%mm, %kk] ...
    %2 = memref.cast %1 : memref<4x4xi32, #map0> to memref<?x?xi32, #map0>
    %3 = accel.send %2, %0 : (memref<?x?xi32, #map0>, i32) -> i32
    scf.for %kk = %c0 to %c32 step %c4 {
      %???? = accel.sendLiteral %c2_i32, %c0_i32 : (i32, i32) -> i32
      %5 = memref.subview %B_arg1[%kk, %nn] ] ...
      %6 = memref.cast %5 : memref<4x4xi32, #map1> to memref<?x?xi32, #map1>
      %7 = accel.send %6, %4 : (memref<?x?xi32, #map1>, i32) -> i32
      %???? = accel.sendLiteral %c4_i32, %c0_i32 : (i32, i32) -> i32
      %???? = accel.sendLiteral %c8_i32, %c0_i32 : (i32, i32) -> i32
      %10 = memref.subview %C_arg2[%mm, %nn] ...
      %11 = memref.alloca() : memref<4x4xi32>
      %12 = accel.recv %11, %9 : (memref<4x4xi32>, i32) -> i32

      // Accumulation of received C (%12) on the CPU subview of C (%10)
      linalg.generic ... ins(%11 : memref<4x4xi32>) outs(%10 : memref<4x4xi32, #map1>) {...}
    }
  }
}
```

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
```

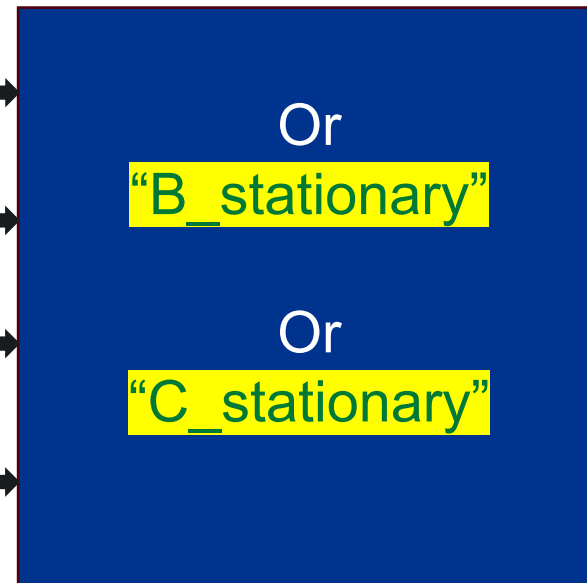
```
sendA = [send_literal(1), send(0)],
```

```
sendB = [send_literal(2), send(1)],
```

```
compute = [send_literal(4)],
```

```
recvC = [send_literal(8), recv(2)],
```

```
sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
```



```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
```

```
#nothing_stationary1 = opcode_flow<(sendA sendB compute recvC)>
```

```
#A_stationary_option0 = opcode_flow<(sendA (sendB compute recvC))>
```

```
#B_stationary_option0 = opcode_flow<(sendB (sendA compute recvC))>
```

(sendB (sendC compute recvC))

```
#C_stationary_option0 = opcode_flow<((sendA sendB compute) recvC)>
```

((sendA sendB compute) recvC)

>

Supporting Accelerators with Many Opcodes

```
#my_opcodes = opcode_map<
  sendA = [send_literal(1), send(0)],
  sendDims = [send_literal(16), send_dim(2,0), send_dim(2,1), send_dim(0,1)],
  sendB = [send_literal(2), send(1)],
  sendAB = [send_literal(3), send(0), send(1)],
  compute = [send_literal(4)],
  sendA_compute = [send_literal(5), send(0),],
  sendB_compute = [send_literal(6), send(1),],
  sendAB_compute = [send_literal(7), send(0), send(1)],
  recvC = [send_literal(8), recv(2)],
  sendA_recvC = [send_literal(9), send(0), recv(2)],
  sendB_recvC = [send_literal(10), send(1), recv(2)],
  sendAB_recvC = [send_literal(11), send(0), send(1), recv(2)],
  compute_recvC = [send_literal(12), recv(2)],
  sendA_compute_recvC = [send_literal(13), send(0), recv(2)],
  sendB_compute_recvC = [send_literal(14), send(1), recv(2)],
  sendAB_compute_recvC = [send_literal(15), send(0), send(1), recv(2)],
>
```

```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
#nothing_stationary1 = opcode_flow<(sendA sendB compute recvC)>

#nothing_stationary2_reset = init_opcodes<(sendDims)>
#nothing_stationary2_flow0 = opcode_flow<(sendAB_recvC)>
#nothing_stationary2_flow1 = opcode_flow<(sendA sendB compute recvC)>

#A_stationary_option0 = opcode_flow<(sendA (sendB compute recvC))>
#A_stationary_option1 = opcode_flow<(sendA (sendB_compute recvC))>
#A_stationary_option2 = opcode_flow<(sendA (sendB_compute_recvC))>

#B_stationary_option0 = opcode_flow<(sendB (sendA compute recvC))>
#B_stationary_option1 = opcode_flow<(sendB (sendA_compute recvC))>
#B_stationary_option2 = opcode_flow<(sendB (sendA_compute_recvC))>

#C_stationary_option0 = opcode_flow<((sendA sendB compute) recvC)>
#C_stationary_option1 = opcode_flow<((sendAB compute) recvC)>
#C_stationary_option2 = opcode_flow<((sendAB_compute) recvC)>
```

Check AXI4MLIR Guides for More Examples

AXI4MLIR library

Here are some of the key files where AXI4MLIR is defined within the llvm-project:

[Accel Dialect](#)

[New Attributes](#)

[DMA Library](#)

[Linalg to Accel transformation pass](#)

[Accel to AXI4MLIR DMA library transformation pass](#)

[Current Function Prototypes](#)

[Example](#)

[Available Options \(end of the file\)](#)

Tests [1](#) & [2](#)

Guides

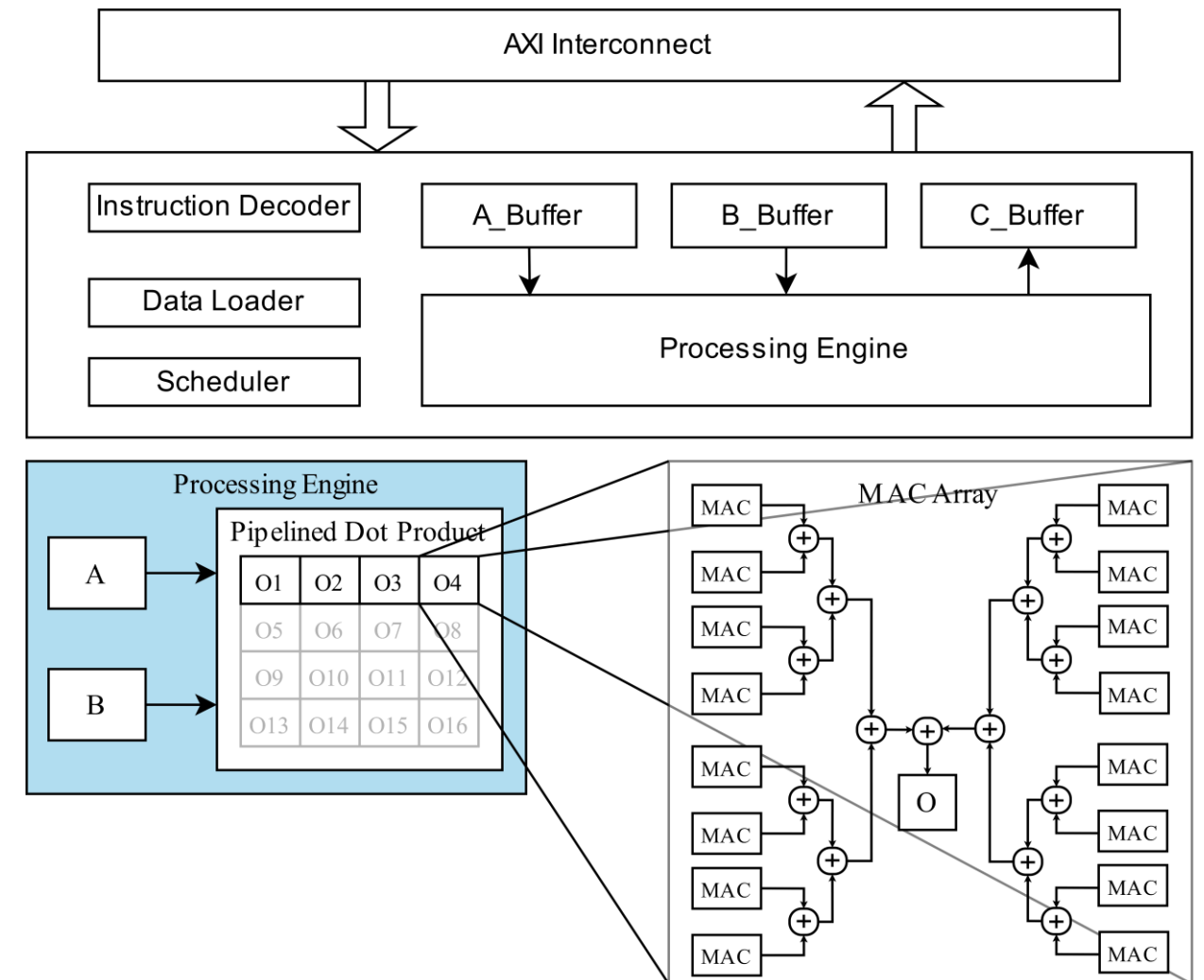
Here are some relevant guides to get started with AXI4MLIR:

- [Enabling a new accelerator compatible with AXI4MLIR](#)
- [Create opcode maps and flows for custom accelerators](#)

https://github.com/AXI4MLIR/axi4mlir/tree/cgo2024_artifact

Accelerator System Level Design

- For our MatMul experiments, we developed simple, scalable MatMul accelerators
- Supports different **dataflows** via instructions (**Instruction Decoder**)
- Input data streamed via **AXI interconnect** stored into global **A/B buffers** (via the **Data Loader**)
- **Processing Engine** contains local A/B cache to compute the required dot product within a MAC array



MMv4₁₆

Manual vs. AXI4MLIR Generate Host Code

Type	Possible Reuse	Opcode(s)	Configurations (Size, OPs/Cycle)
$v2_{size}$	Inputs	sA, sB, cCrC	(4, 10)
$v3_{size}$	Ins/Out	sA, sB, cC, rC	(8, 60)
$v4_{size}$	Ins/Out (flex size)	sA, sB, cC, rC	(16, 112)



```
#nothing_stationary0 = opcode_flow<(sendAB_recvC)>
#nothing_stationary1 = opcode_flow<(sendA sendB compute recvC)>

#nothing_stationary2_reset = init_opcodes<(sendDims)>
#nothing_stationary2_flow0 = opcode_flow<(sendAB_recvC)>
#nothing_stationary2_flow1 = opcode_flow<(sendA sendB compute recvC)>
```



```
#A_stationary_option0 = opcode_flow<(sendA (sendB compute recvC))>
#A_stationary_option1 = opcode_flow<(sendA (sendB_compute recvC))>
#A_stationary_option2 = opcode_flow<(sendA (sendB_compute_recvC))>

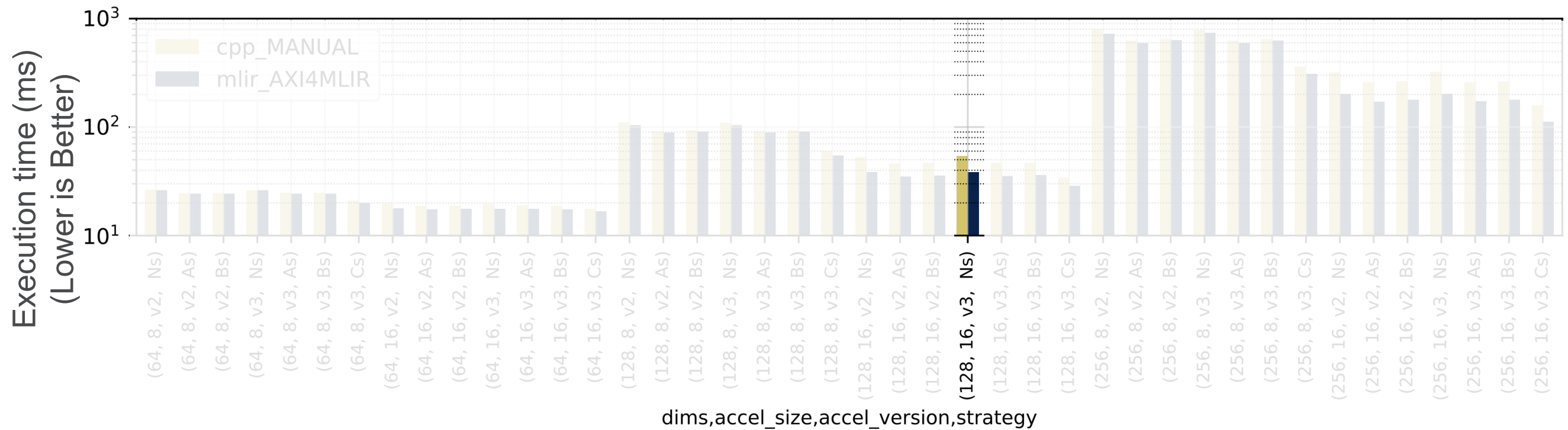
#B_stationary_option0 = opcode_flow<(sendB (sendA compute recvC))>
#B_stationary_option1 = opcode_flow<(sendB (sendA_compute recvC))>
#B_stationary_option2 = opcode_flow<(sendB (sendA_compute_recvC))>
```



```
#C_stationary_option0 = opcode_flow<((sendA sendB compute) recvC)>
#C_stationary_option1 = opcode_flow<((sendAB compute) recvC)>
#C_stationary_option2 = opcode_flow<((sendAB_compute) recvC)>
```

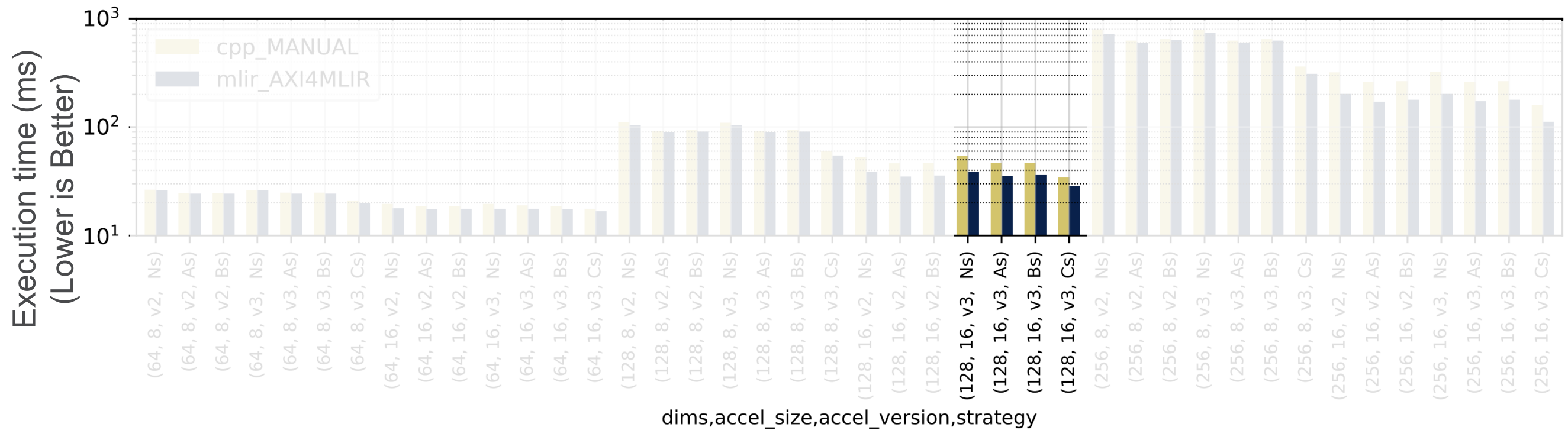
Manual vs. AXI4MLIR Generate Host Code

Type	Possible Reuse	Opcode(s)	Configurations (Size, OPs/Cycle)
$v2_{size}$	Inputs	sA, sB, cCrC	(4, 10)
$v3_{size}$	Ins/Out	sA, sB, cC, rC	(8, 60)



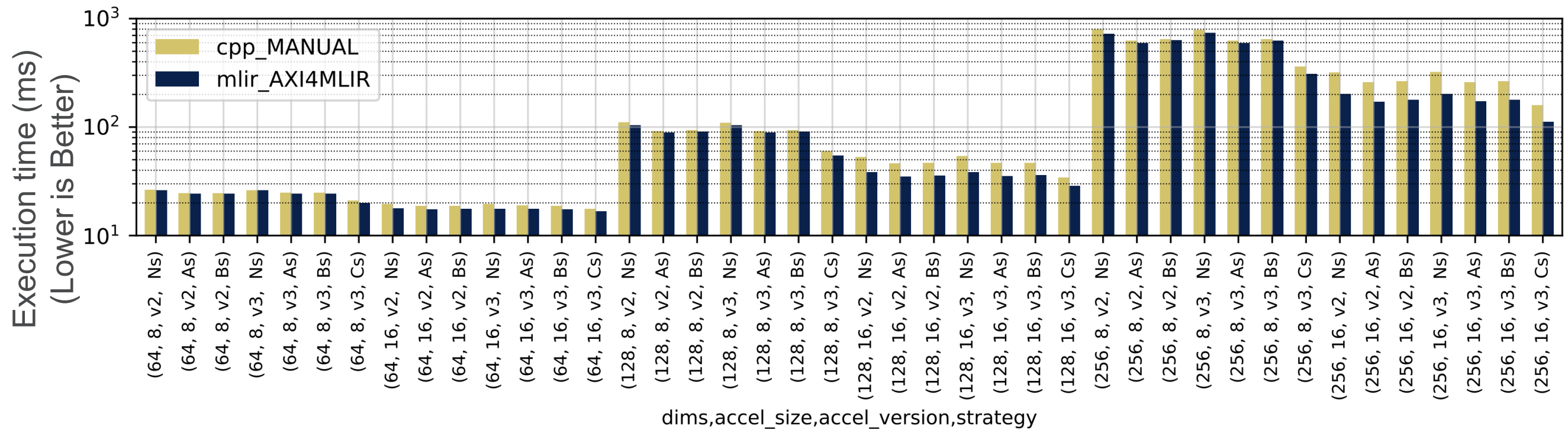
Manual vs. AXI4MLIR Generate Host Code

Type	Possible Reuse	Opcode(s)	Configurations (Size, OPs/Cycle)
$v2_{size}$	Inputs	sA, sB, cCrC	(4, 10)
$v3_{size}$	Ins/Out	sA, sB, cC, rC	(8, 60)



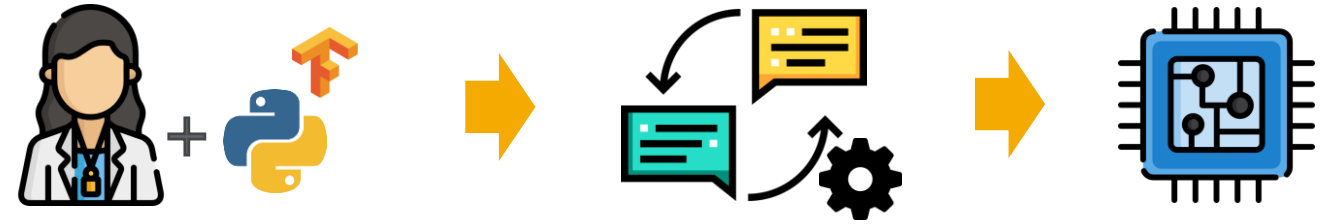
Manual vs. AXI4MLIR Generate Host Code

Type	Possible Reuse	Opcode(s)	Configurations (Size, OPs/Cycle)
$v2_{size}$	Inputs	sA, sB, cCrC	(4, 10)
$v3_{size}$	Ins/Out	sA, sB, cC, rC	(8, 60)



Up to 1.65x speedup
and 56% cache references

End-to-End ResNet18

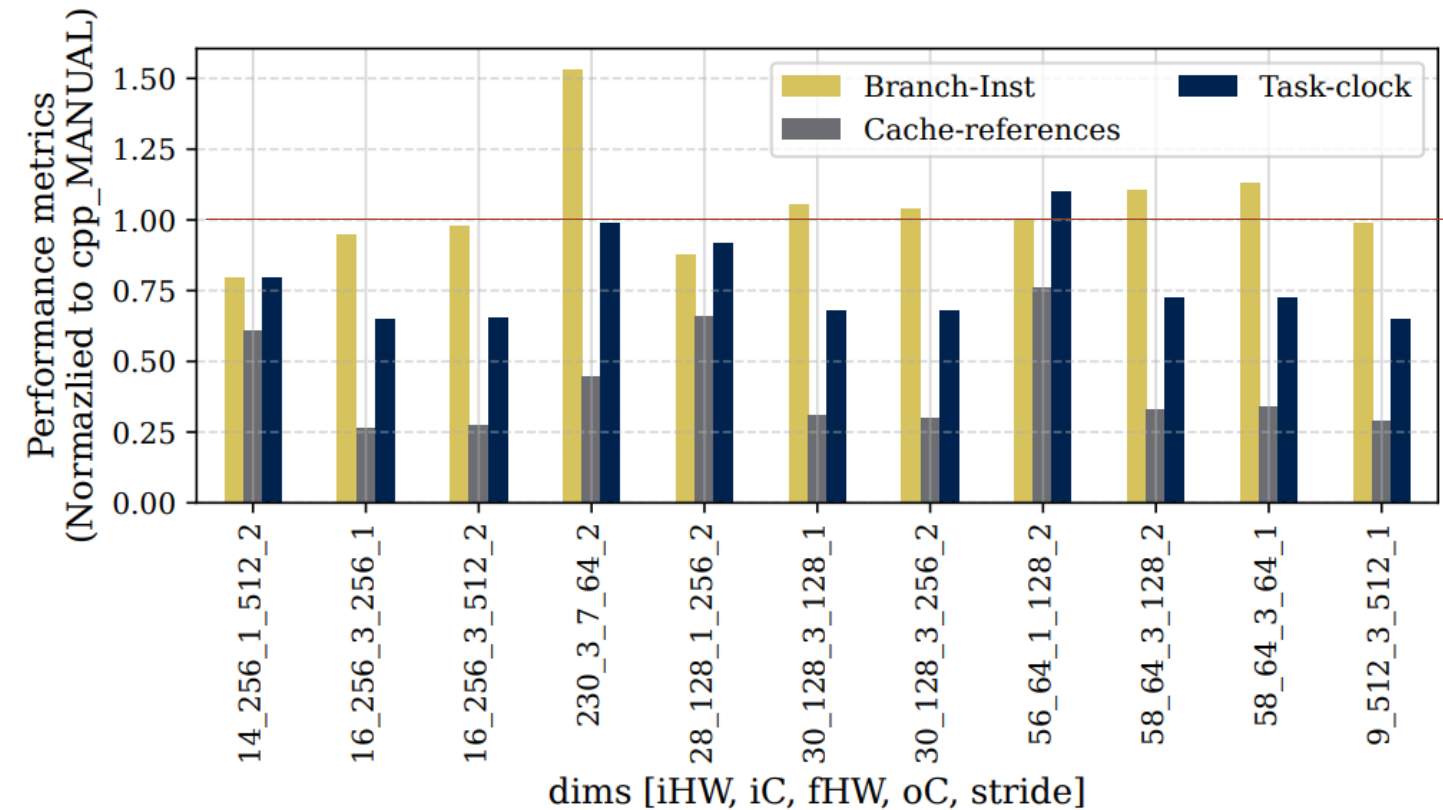


- Accelerating Convolutional Layers of ResNet18 image classification model
- `init_opcodes` attribute is used to send the dimensions of the computation before every convolution

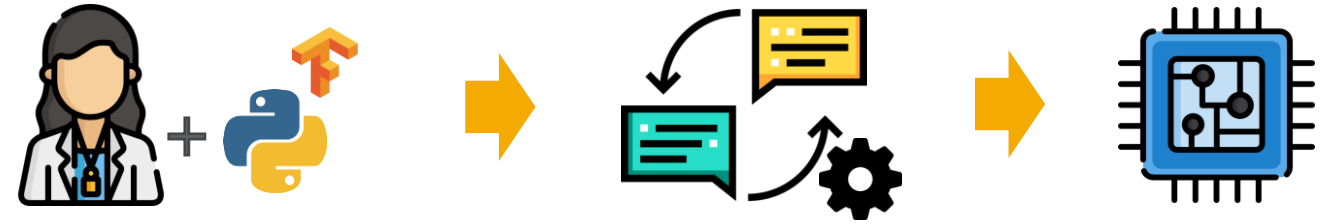
```

1 accel_dim = map<(B,H,W, iC,oC,fH,fW) ->
2             (0,0,0,256, 1, 3, 3)>, // Tiling
3 opcode_map<
4   sIcO=[send_literal(70), send(0)], // send 3D input window
5                                     // and compute
6   sF=[send_literal(1), send(1)],    // send 3D filter
7   rO=[send_literal(8), rcv(2)],    // rcv 2D output slice
8   rst=[send_literal(32), send_dim(1,3), // set filter size
9        send_literal(16), send_dim(0,1)]> // set iC size
10 opcode_flow <(sF (sIcO) rO)> // filter+output stationary
11 init_opcodes <(rst)>

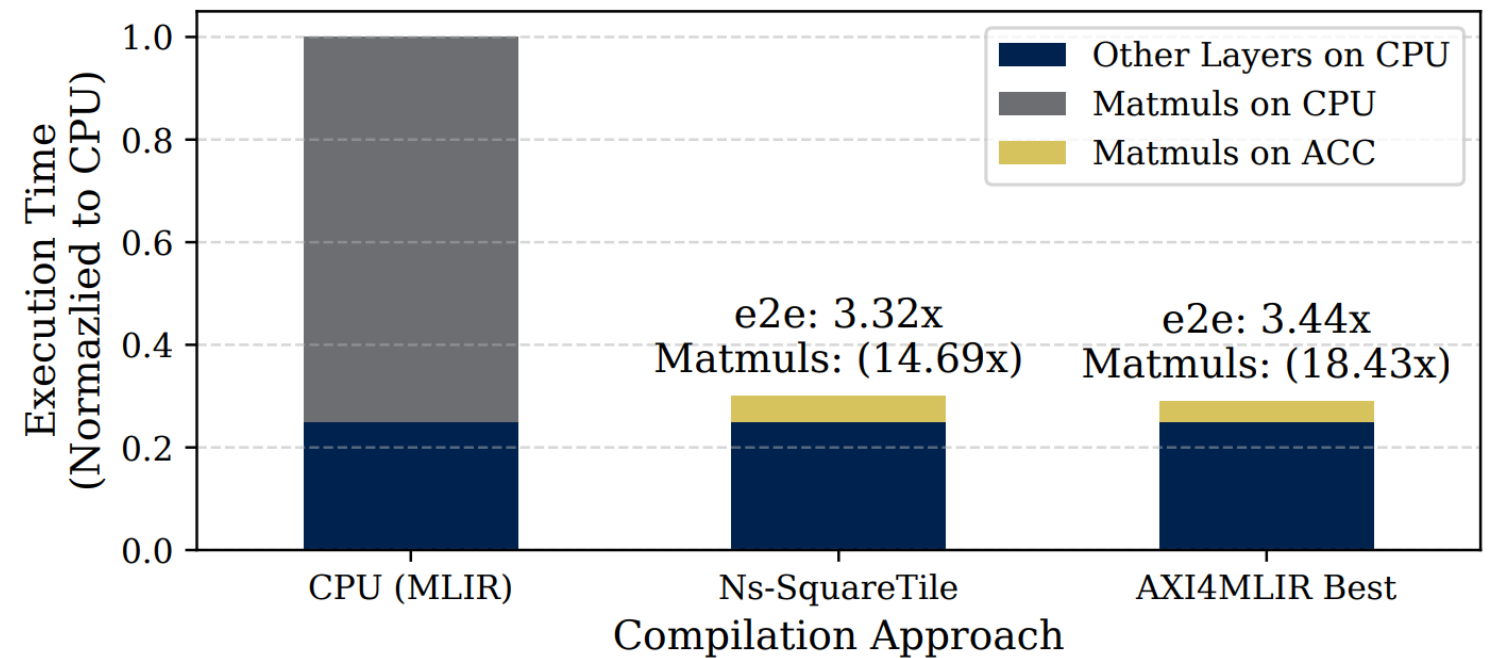
```



End-to-End TinyBERT

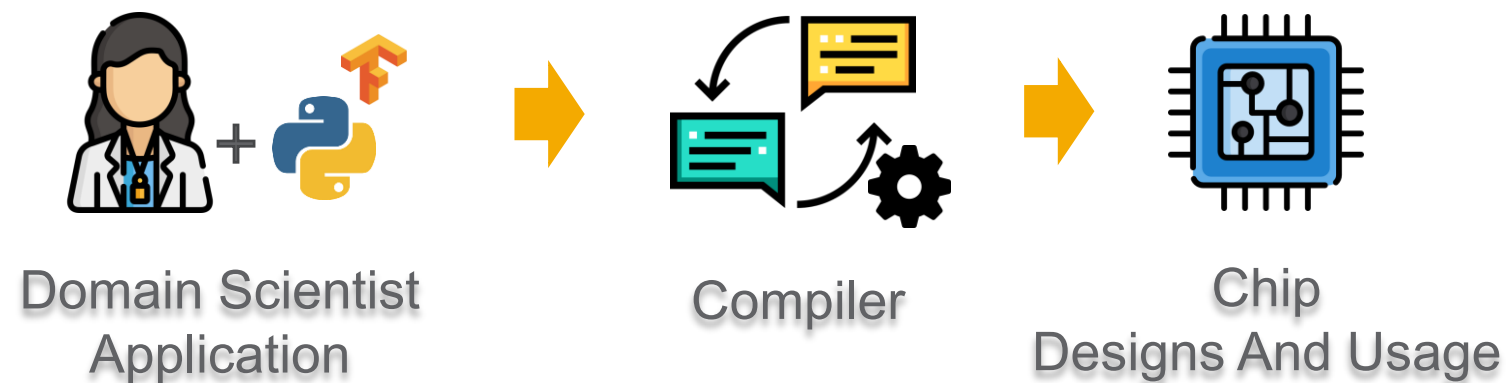


- Accelerating Matrix-Multiplication Layers of TinyBERT language model
- Presenting results for
 - CPU execution
 - Worst case (Nothing stationary)
 - Best case (oracle selection of best dataflow)



Contributions

- Standardized and extensible approach to **represent accelerators with an instruction** set architecture in MLIR
- Ability to describe and **explore** accelerator-specific **tiling and dataflow** strategies
- AXI4MLIR generated code **outperforms** manual implementations by up to **1.65x**, with average **56% fewer cache references**



“Democratizing System-Level Design, Automatic Generation, and Use of Optimized Custom Domain Specific Accelerators”

Thank you

