Agostini, N. B., Haris, J., Gibson, P. , Jayaweera, M., Rubin, N., Tumeo, A., Abellán, J. L., Cano Reyes, J. and Kaeli, D. (2024) AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators. In: International Symposium on Code Generation and Optimization (CGO) 2024, Edinburgh, United Kingdom, 02-06 Mar 2024, pp. 143-157. ISBN 9798350395099 (doi: 10.1109/CGO57630.2024.10444801)

This is the author version of the work. There may be differences between this version and the published version. You are advised to consult the published version if you wish to cite from it: https://doi.org/10.1109/CGO57630.2024.10444801

https://eprints.gla.ac.uk/311284/

Deposited on 11 December 2023

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

AXI4MLIR: User-Driven Automatic Host Code Generation fc

Abstract—**This paper addresses the efficient generation of host driver cc AXI-based accelerators targeting linea important workload in various applic learning and scientific computing. V focused on automating accelerator pr has been paid to the host-acceleratoı introduces AXI4MLIR, an extension framework designed to facilitate the host-accelerator driver code. With n transformations, AXI4MLIR empowe erator features (including their instruc patterns and exploit the host memory h AXI4MLIR's versatility across differen problems, showcasing significant CPU ((up to 56%) and up to a 1.65× speed optimized driver code implementations link will be included for the camera r**

Index Terms—**MLIR, AXI, Compile**

I. INTRODUCTI

Given the diminishing performa today's general-purpose computing [1], there has been renewed interest in exploring custom hardware accelerators. Accelerators can support architecture-level optimizations that can increase the performance and efficiency of key applications [2], [3], [4], [5], [6], [7]. One important class of applications that can benefit from accelerators is tensor algebra processing, which is widely used in the domains of machine learning, scientific computing, and data analytics [8], [9], [10]. Tensor operations tend to be computationally intensive and require high memory bandwidth, making them suitable for specialized hardware implementations. Automated tools have been proposed [11], [12], [13], [14] to help explore new classes of custom domain-specific accelerators targeting tensor computations, and are currently the best path available to obtain performance gains in scientific workloads and machine learning applications.

However, designing and fully exploiting custom hardware accelerators for tensor operations is not a trivial task. When co-designing these devices, we need to generate efficient architectures, and we must optimize the communication between the host CPU and the accelerator. In particular, the host-accelerator interaction involves several aspects, including data transfers, synchronization, and the accelerator's control flow. These aspects depend on the characteristics of the host CPU microarchitecture, the host-accelerator interface, the accelerator design, and the application code. Manually rewriting the host driver code for each accelerator and application scenario can be very tedious and error-prone. Furthermore, most of

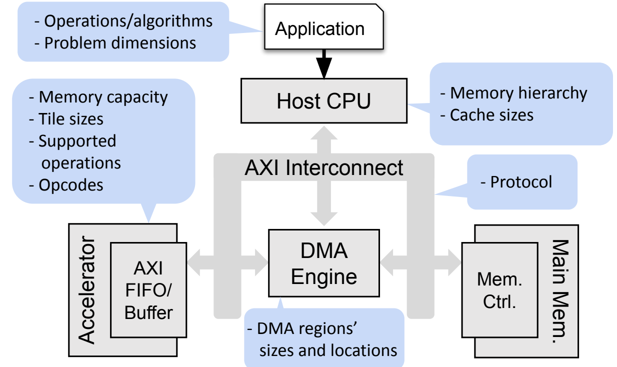


Fig. 1: Typical host-accelerator system design, highlighting (blue color) relevant parameters that should be considered for efficient generation of host-accelerator communication code.

the prior work proposing new accelerators [15], [16], [17], [18], [19] only considers a simple offload model or assumes that the required data is already placed in the accelerator's internal buffers, falling short in providing insights into how host-to-accelerator transfers should be performed or generated. Additionally, complex accelerators, exemplified by Google's TPUs and Nvidia's GPUs, benefit from large teams that can collaboratively engineer dedicated compilers to address some of these issues. However, smaller development teams may lack expertise or available time resources to invest in compilers. Consequently, custom accelerator designers typically implement driver code and instruction streams manually to validate and deploy their designs for a subset of synthetic workloads.

To implement or generate efficient host-to-accelerator communication, we argue that it is necessary to consider all major features of a System-on-Chip (SoC). Figure 1 highlights a typical system using an AXI [20] interconnection between the CPU and a custom accelerator, which is a common choice in many designs [21]. To drive the accelerator effectively, the host-code implementation should exploit features regarding the CPU, the interconnect, and the accelerator (see Figure 1).

To effectively consider each of the key system features described in Figure 1 while also delivering efficient and automated CPU-accelerator driver code generation, we propose **AXI4MLIR**, an extension to the MLIR compiler framework [22] that enables efficient and automated CPU-accelerator driver code generation for accelerators targeting linear algebra applications. AXI4MLIR takes a high-level application description in the MLIR's linear algebra (`linalg`) abstraction [23] as input and introduces custom MLIR attributes to describe the target accelerator capabilities. These attributes provide accelerator-specific information to

custom transformation passes that can effectively specialize and generate accelerator-aware host driver code. Our extensions facilitate hardware-software co-design by allowing developers to automatically generate driver code with varying configurations, more easily explore their design space, and use the designed accelerator in applications that can be compiled with the MLIR framework. The contributions of this work include the following:

- New MLIR attributes that provide a standardized and extensible approach to represent accelerators that can implement a range of linear algebra algorithms supported by the MLIR *linalg* abstraction.
- Automated generation of efficient driver code for custom accelerators leveraging AXI-based interfaces in host-to-accelerator communication.
- The ability to describe and explore accelerator-specific tiling and dataflow strategies for the target linear algebra operation, which can improve computation efficiency within the accelerator and reduce data movement overheads between the accelerator and CPU.
- An analysis of our compiler optimizations on a suite of benchmarks representing key linear algebra applications, demonstrating the effectiveness of our approach in achieving significant performance gains (up to 1.65× speedup and 56% fewer cache references) when compared to optimized manual driver code implementations.

While leveraging the new attributes of AXI4MLIR, our user-directed host code generation is entirely automated by the compiler. This provides a significant advantage in terms of productivity and maintainability.

II. Background

A. MLIR

MLIR is a compiler infrastructure framework that facilitates the creation of domain-specific compilers by providing code generators, translators, optimizers, and the infrastructure to define subsets of operations that expose well-defined language abstractions [22], [24]. Notably, MLIR offers support for compilation from various frontends into its infrastructure, encompassing frameworks such as TensorFlow, PyTorch, and ONNX, as well as languages like Fortran, C, and Mojo. In MLIR, a group of operations modeling an abstraction is called a *dialect*. Dialects are self-contained intermediate representations (IRs) and follow the language rules of MLIR's meta-IR, enabling the framework to have multiple dialects coexisting in the same MLIR file. This approach promotes the reuse of already defined abstractions and associated tools, enabling intra- and inter-dialect transformations.

In support of the underlying algorithms and kernels used by many machine learning frameworks (e.g., TensorFlow and PyTorch), MLIR offers a linear algebra dialect called `linalg` that exposes (named) operations such as convolutions, matrix multiplications, and others. Operations expressed in higher-level dialects can target `linalg` operations and leverage all subsequent transformations supported by `linalg` and lower-level abstractions. Figure 2 presents an MLIR matrix-multiplication (MatMul) implementation in different abstractions.[1] The operation is initially represented using a `linalg.matmul` and subsequently undergoes conversion, transformation, and lowering by the compiler. In Figure 2a-L11 and L17, the `linalg.matmul` is converted into a `linalg.generic`. The `linalg.generic` is a core MLIR operation that can represent most of the `linalg` *named ops*, by careful selection of its *operation trait*[2] - `indexing_maps` (L2), `iterator_types` (L7) -, and kernel (L24 to L27). Finally, the generic operation can be converted into a tiled (4×4×4) implementation of the MatMul (Figure 2b) using the structured control flow (`scf`) dialect. When supporting an accelerator that can process a MatMul_{4x4x4} operation [3], the code in Figure 2b-L11 to L19, has to be replaced by the runtime library calls that drive the accelerator.

1) MLIR Memory References: Within MLIR, memory buffers exist as N-dimensional (rank=N) memory references, or `memref`s. Our proposed AXI4MLIR DMA runtime library, presented in Section III-A, supports bidirectional data movements between `memrefs` and memory-mapped buffers (raw pointers), while respecting strides, sizes, and dimensions. Ac-

```
#matmul_trait = {
  indexing_maps = [
    affine_map<(m, n, k) -> (m, k)>, // A
    affine_map<(m, n, k) -> (k, n)>, // B
    affine_map<(m, n, k) -> (m, n)>  // C
  ]
  iterator_types = [
    "parallel", "parallel", "reduction"],
}
func.func @matmul_call(...) {
  linalg.generic #matmul_trait
    ins (%A, %B : memref<60x80xf32>, memref<80x72xf32>)
    outs(%C : memref<60x72xf32>) {
    ^bb0(%a: f32, %b: f32, %c: f32):
      %0 = arith.mulf %a, %b : f32
      %1 = arith.addf %c, %0 : f32
      linalg.yield %1 : f32 }
  return }
```

(a) Linalg Abstraction with generic operation.

```
func.func @matmul_call(...) {
  // Declare constants %c0 %c1 %c4 %c60 %c72 %c80 ...
  scf.for %m = %c0 to %c60 step %c4 { // Tiling by 4,4,4
    scf.for %n = %c0 to %c72 step %c4 {
      scf.for %k = %c0 to %c80 step %c4 {
        // Grab handle for the sub-tiles:
        %sA = memref.subview %A[%m, %k] [4, 4] [1, 1] : ...
        %sB = memref.subview %B[%k, %n] [4, 4] [1, 1] : ...
        %sC = memref.subview %C[%m, %n] [4, 4] [1, 1] : ...
        // Matmul computation of a 4x4x4 tile:
        scf.for %mm = %c0 to %c4 step %c1 {
          scf.for %nn = %c0 to %c4 step %c1 {
            scf.for %kk = %c0 to %c4 step %c1 {
              %3 = memref.load %sA[%mm, %kk] : !mr4x4_0
              %4 = memref.load %sB[%kk, %nn] : !mr4x4_1
              %5 = memref.load %sC[%mm, %nn] : !mr4x4_1
              %6 = arith.mulf %3, %4 : f32
              %7 = arith.addf %5, %6 : f32
              memref.store %7, %sC[%mm, %nn] : !mr4x4_1
  } } } } } }
  return }
```

(b) Structured Control Flow (SCF) Abstraction with tiling.

Fig. 2: MLIR representations of a Matrix-Matrix Multiplication Operation in different abstractions.[1]

[1] We intentionally omit some MLIR code, such as constant declarations in the form of `%cX=arith.constant X:i32`, for the sake of brevity.

[2] See *linalg.generic* in https://mlir.llvm.org/docs/Dialects/Linalg

[3] A 2D MatMul operation is MatMul_{MxNxK}: C(M,N) = A(M,K) x B(K,N)

cessing the elements of an MLIR `memref` requires accessing the values in the equivalent C struct of Figure 3. Specializing the code for specific sizes and strides is an important proposed optimization to leverage spatial locality and minimize control-flow instructions, as we will observe in Section IV.

```
typedef struct {
  float *allocated; // For deallocation
  float *aligned;   // Base address
  size_t offset;    // Offset in # of elements
  size_t size[N];   // One size per dim
  size_t stride[N]; // One stride per dim
}
```

Fig. 3: The underlying data structure of a rank==N MLIR `memref` buffer.

B. AXI Interface

Efficiently using the interconnect between the CPU and the accelerator can significantly impact the overall system performance. As part of our framework, we consider a widely adopted bus interface in digital electronics design deployed on SoC and Field-Programmable Gate Array (FPGA) designs, namely Advanced eXtensible Interface (AXI) [20]. AXI provides a flexible and scalable solution for integrating custom accelerators into a system.

The AXI interface provides a simple mechanism to enable data transfers between the CPU cores and other devices. Using AXI, the AXI-Stream (AXI-S) interface allows the developer to quickly transfer [25] a variable-size burst of data to and from the accelerator in a FIFO-like manner, enabling the accelerator to consume/store the data as needed, in a streaming manner. Within SoCs, the CPU host code controls either a single or multiple Direct Memory Access (DMA) engines (see Figure 1). These engines are responsible for initiating and handling data movement requests between the main memory and the accelerator. Additionally, the data regions in the main memory need to be accessible to the accelerator via the AXI-S interface. Therefore, the host code needs to allocate input and output memory buffers using the *mmap* function, which guarantees that only the current process has access to the specific regions of memory. The host code is also required to prepare/pack the input data into the data format that the accelerator requires (e.g., row-major, interleaved data elements, etc.). Our approach within AXI4MLIR is to *use MLIR - the compiler - to generate the host code to interface with the accelerator*, while taking advantage of the full capabilities of the target accelerator.

III. AXI4MLIR

To support efficient host code generation for AXI-based custom accelerators, we extended the MLIR framework with the added capabilities presented in Figure 4. After the custom accelerator is designed and the host CPU system is selected, the user creates a configuration file with the host CPU system details (e.g., number and size of the caches), and with a high-level description of the accelerator capabilities (i.e., supported operations and dimensions), the available opcodes, and possible opcode flows ①. This information is parsed ② by the compiler, and used to find ③ suitable `linalg.generic`

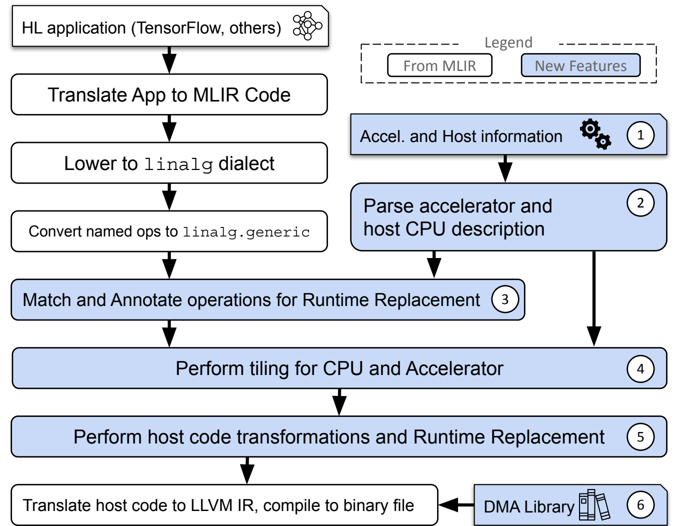


Fig. 4: AXI4MLIR Compiler Flow. The numbered elements are the contributions of this work.

operations with the desired operation traits (algorithm implemented, previously shown in Figure 2a-L1 to L9), that can be executed on the accelerator. These operations will require host-accelerator driver code generation. Subsequently, with user-provided information on the total size of the CPU caches, the compiler transforms the code to efficiently exploit the CPU memory hierarchy and the accelerator size ④, performing the appropriate set of tiling transformations to leverage temporal locality in the CPU caches and to map the problem on the accelerator. In the final step, the compiler generates the runtime calls ⑤ that leverage the accelerator features based on user-directed dataflow description (e.g., avoiding redundant host-accelerator data transfers when the algorithm and accelerator functionality allows).

The following sections discuss the class of supported accelerators and the key features of our AXI4MLIR DMA library. We provide details on how to describe new accelerators, introducing `linalg.generic` trait extensions, a new MLIR dialect that provides support for runtime call replacement of opcodes and data transfers, and some key optimizations that can be performed (depending on the available features of the host system and the custom accelerator).

A. The Custom AXI DMA Library

The AXI4MLIR *DMA library* ⑥ (Figure 4) exposes low-level DMA calls working at privileged level to enable data movement between the main memory and the accelerator. We designed this library to be lightweight (55 bytes in size for our target ARM SoC), so that it can be deployed on both resource-constrained and non-constrained systems. It can also be executed by bare-metal systems. During the compilation process, the AXI runtime issues calls to initialize the DMA engine(s) before entering the computation kernel of the workload. First, a library call initializes the DMA engine, mapping memory for the input and output buffers which act as temporary staging buffers between the CPU and the accelerator.

After DMA initialization, the accelerator is accessible via AXI-based data transfers. Any data that needs to be transferred to the accelerator during workload execution is first copied to a DMA input buffer. This staging copy acts as a packing optimization (similar to [26]), contributing to an increased cache-hit ratio during communication. Then, the AXI "send" function call requests the DMA engine to start the data transfer and waits for it to finish. Note that the data that is sent to the accelerator can be either accelerator instructions or raw input data that needs to be processed. Similarly, AXI4MLIR generates "recv" function calls to wait for computation completion and to obtain output data from the DMA output buffer.

In Section III-C, Figure 9 presents the lowering of different high-level operations into our DMA library calls. `copy_to_dma_region(...)` implements data movement from a `memref` to the DMA-accessible memory region intended for transmission to the accelerator. The `offset` argument allows for efficient batching of different data transfers after computing the total length and executing a single "send" operation. Appropriate offset values prevent overwriting existing data in the DMA region. `dma_start_send(...)` instructs the DMA engine to transmit a `size` of X bytes to the connected accelerator, commencing from a specified `offset` within the DMA space allocated. `dma_wait_send_completion(...)` instructs the CPU to wait for the DMA's signal informing the transaction's completion. When receiving data from the accelerator, we first have to wait for the data to be placed in the DMA-accessible memory so it can be copied back into a `memref`.

B. Supported Accelerators

In matrix-multiplication and similar algorithms, the term *stationary* refers to a slice of data that can be reused across many iterations of an algorithm's computation. A *stationary* strategy attempts to maximize data reuse and minimize data movement, which can greatly benefit accelerators that require efficient memory accesses. We want to enable the programmer to easily control accelerators that support *stationary* flows.

Next, we discuss the types of accelerators that AXI4MLIR can support. Then we propose a standardized approach to concisely define the class of supported accelerators in a configuration file. Finally, we show how the AXI4MLIR parser is able to take user-defined configurations, extract essential attributes of the target accelerator, and populate a trait specification to guide our MLIR compiler transformations.

1) Accelerator Designs: The AXI4MLIR compiler transformations support linear algebra kernels implemented as accelerators using the AXI interconnect. In addition, the AXI-S data transfers within AXI4MLIR facilitate support for accelerators that use a micro-ISA (Instruction Set Architecture) with opcodes, which consist of instructions that the host-CPU sends to the accelerator. Generally, the following three actions are used to categorize the actions within an instruction: *send*, *compute*, and *receive*. Any accelerator's instructions that require external communication (i.e., data transfers or activation/reset/configuration of accelerator compute modules) can be completed by issuing a combination of these three actions. In addition, each action can have additional meta-data (e.g., opcode literal, data, length, dimensions, and indexes), which is used to guide compiler transformations during accelerator host code generation. Further, specific traits of the accelerator - such as internal buffer space (or accelerator tile sizes), and data types - are supported and must be defined within the accelerator configuration file.

```
1  {"cpu" = { "cache-levels": [32K,512K],
2            "cache-types": [data,shared] }
3   "accelerators" = [
4     { "name": ..., "version": x.x, "description": ...,
5       "dma_config" : {...}, "kernel": "linalg.matmul",
6       "accel_size": [4,4,4], "data_type": int32,
7       "dims": ["m", "n", "k"],
8       "data": { "A": [m,k], "B": [k,n], "C": [m,n]},
9       "opcode_map" : "<opcode_map string - see IV-D>",
10      "opcode_flow_map" : { "flowID01" :
11            "<opcode_flow string - see IV-D>", ...},
12      "selected_flow" : "flowID01" }]}
```

Fig. 5: Accelerator and CPU configuration file.

2) Accelerator Configuration File: Once an AXI-based accelerator is fully designed, the accelerator developer can quickly integrate it with our AXI4MLIR compiler transformations by providing *Accelerator and Host information* ① (Figure 4) through a configuration file for the new accelerator and the target host system. Figure 5 shows a sample configuration file defined in the standard JSON format. For the accelerator, the developer must specify the accelerator's architectural features, e.g., supported tile sizes, data type, and input and output data with related dimensions. Additionally, the developer should describe any micro-ISA that the accelerator can execute. The developer should define "opcode IDs", captured by the "opcode_map string", which are comprised of actions to describe the memory operations and related data transfers. Finally the developer should define the possible "opcode flow IDs" and select the desired flow for the particular operation. The configuration file does not capture the internal behavior of the accelerator, which has been the focus of other works [12], [15]; instead, we seek to optimize the communication with the accelerator. Thus the configuration file contains information about the I/O interface for sending data and instructions to the accelerator. Similar to the accelerator information, the CPU information, shown in Figure 5-L1 to L2, needs to contain basic architectural details such as the number and size of caches.

3) Configuration Parsing: The parser implemented in ② (Figure 4) is responsible for providing the information from the configuration file to the MLIR IR and the AXI4MLIR transformation passes. To this end, the *kernel* and *cache information*, paired with a simple heuristic that identifies the dimensions of the target MLIR operation, are used to schedule tiling transformations (Figure 4 - ④) that leverage the CPU memory hierarchy sizes and increase temporal locality of the memory accesses. Additionally, the parser validates the `opcode_map` and the user selected `opcode_flow`, which are then translated into new MLIR attributes to the target

```
#matmul_accel_trait = {
  dma_init_config = {        id = 0x0,
    inputAddress = 0x42,     inputBufferSize = 0xFF00,
    outputAddress = 0xFF42,  outputBufferSize = 0xFF00 },

  // Opcodes sent once. Tokens defined in opcode_map.
  init_opcodes = init_opcodes < (reset) >,

  accel_dim = map<(m, n, k) -> (4, 4, 4)>, // Tiling

  // Permutation and who can be stationary.
  permutation_map = affine_map<(m, n, k) -> (m, k, n)>,

  opcode_map = opcode_map < // Valid Opcodes
    sA       = [send_literal(0x22), send(0)],
    sB       = [send_literal(0x23), send(1)],
    cC       = [send_literal(0xF0)],
    rC       = [send_literal(0x24), recv(2)],
    sBcCrC   = [send_literal(0x25), send(1), recv(2)],
    reset    = [send_literal(0xFF)] >,

  // Flow to implement. Tokens defined in opcode_map.
  opcode_flow = opcode_flow < (sA (sBcCrC)) > // As
  // Example of other     < ((sA sB cC) rC) > // Cs
  //       valid flows    < (sB sA cC rC) >   // Ns
}
```

(a) New Attributes for Accelerator Description.

```
func.func @matmul_call(...) {
  // Declare constants (loop bounds and literals): %cX, ...
  accel.dma_init(%c0,%c66,%c65280,%c65346,%c65280) : ...
  accel.sendLiteral(%c0xFF, %c0) : i32,i32->i32 // reset
  // Tiling by 4,4,4
  scf.for %m = %c0 to %c60 step %c4 { // first loop
    scf.for %k = %c0 to %c80 step %c4 { // second loop
      %sA = memref.subview %A[%m, %k] [4, 4] [1, 1] : ...
      %offset0 = accel.sendLiteral(%c0x22,%c0):i32,i32->i32
      accel.send(%sA, %offset0) : !mr4x4_0, i32 -> i32
      scf.for %n = %c0 to %c72 step %c4 { // innermost
        %sB = memref.subview %B[%k, %n] [4, 4] [1, 1] :
        %sC = memref.subview %C[%m, %n] [4, 4] [1, 1] :
        %offset1 = accel.sendLiteral(%0x25,%c0): ...
        %offset2 = accel.send(%sB, %offset1) :
                              !mr4x4_0, i32 -> i32
        accel.recv {mode="accumulate"}(%sC, %c0) :
                              !mr4x4_0, i32 -> i32
  } } } } } }
  return }
```

(b) IR to drive the MatMul accelerator with an A-stationary flow.

Fig. 6: Information added to the linalg.generic traits to capture accelerator behavior in MLIR and IR with accel operations.

`linalg.generic` operation trait. Their syntax and functionality are described in Section III-C.

4) Supported Systems: Our work is focused on SoCs with accelerators connected to ARM CPUs via an AXI-S interconnect. AXI4MLIR seamlessly integrates with a diverse set of Xilinx platforms, though we also anticipate similar applicability to other FPGA-SoC devices. Changing the cross-compiler would allow support for other processors. Adapting our DMA library implementation to other standards would be required to support other types of interconnects. AXI4MLIR currently supports AXI-Stream accelerators, which do not communicate via direct memory requests. Thus, AXI4MLIR does not require support for host-accelerator coherence protocols, since the host manages the DMA engine transfers.

C. MLIR extensions and optimizations

To implement *match and annotate operations for runtime replacement* (3) (Figure 4), and to offload the computation onto the accelerator, we implemented passes to identify the target algorithms supported by the accelerator and extended the `linalg.generic` operation `trait` with additional information, as shown in Figure 6a. In particular, we introduced

```
opcode_dict   ::=
    `opcode_map` `<` opcode_entry (`,` opcode_entry)* `>`
opcode_entry ::= (bare_id | string_literal) `=` opcode_list
opcode_list  ::= `[` opcode_expr (`,` opcode_expr)* `]`
opcode_expr  ::= `send` `(` bare_id `)`
              | `send_literal` `(` integer_literal `)`
              | `send_dim` `(` bare_id `)`
              | `send_idx` `(` bare_id `)`
              | `recv` `(` bare_id `)`
```

Fig. 7: Opcode Map Syntax. A dictionary for accelerator opcodes and actions.

```
opcode_flow_entry ::= `opcode_flow` `<` flow_expr >
flow_expr ::= `(` flow_expr `)` | bare_id (` ` bare_id)*
```

Fig. 8: Opcode Flow Syntax. The sequence of opcodes to implement a specific dataflow of host-accelerator communication.

two new types of attributes to MLIR, `opcode_map` and `opcode_flow`, which follow the syntax described in Figure 7 and Figure 8, respectively. We elaborate more on each attribute in the operation `trait` below.

Extensions to linalg.generic traits:

– `dma_init_config`: defines the parameter values used to configure a DMA engine associated with a specific accelerator. If multiple or different accelerators are present, they would have different values in this field. Figure 6a-L2 to L4 show the available parameters. The code generated for the DMA initialization is executed by the CPU only once per application.

– `init_opcodes`: defines a flow of opcodes that should be sent to initialize or reset the accelerator for a new kernel execution. During application runtime, these opcodes are sent N times, where N is the number of kernels in an application that can be mapped onto the custom accelerator. In Figure 6a-L7, we define that the reset opcode must be included to support the described accelerator. The opcode's functionality is derived from the `opcode_map` parameter below.

– `accel_dim`: defines the size of the accelerator for each dimension of the implemented algorithm. Figure 6a-L9 shows an example, specifying that the *accelerator* supports a tiled MatMul_{4x4x4} version of the implemented algorithm.

– `permutation_map`: defines the order in which nested loops execute. In Figure 6a-L12, we switch the order of the two innermost loops, potentially enabling the data structure that uses *[m,k]* indices to be stationary, as the other data structures are streamed in/out of the accelerator. In our MatMul example (Figure 2b), this enables an `A` stationary dataflow (Figure 6b).

– `opcode_map`: describes accelerator opcodes as key-value pairs. Following the syntax scheme shown in Figure 7, the key, or *opcode_entry*, is an identifier that maps to a list of actions, or *opcode_list*, which represents sequential memory operations that have to be performed to drive the accelerator. Each action, or *opcode_expr* (`send`, `send_literal`, `send_dim`, `send_idx`, `recv`), implements different types of copies to/from the DMA memory-mapped region. The `send` and `recv` actions take an input. The input is a number that is used to represent one of the arguments to the `linalg.generic` operation, e.g., 0, 1, or 2 would map to A, B, or C, respectively, in the MatMul example (Figure 2a-L12-13). During code generation, this information is used to copy the needed tile to the memory-mapped region. For

Fig. 9: Semantics and lowering of `accel` operations.

example, Figure 6a-L15 shows an opcode with identifier *"sA"* that issues copies *to* the accelerator for the *literal* value `0x22` and then for the *data* associated with the tile of argument 0. Furthermore, `send_dim` and `send_idx` can be used to send tile dimensions or tile indices, which could be used to drive more complex accelerators. Subsequent text will refer to an *opcode_entry*, such as *"sA"*, simply as *opcode.*

– `opcode_flow`: represents valid opcode/data transfer *flows* and respects the syntax scheme shown in Figure 8. Figure 6a-L23 shows an example, which defines an *input A stationary* (associated with argument 0) valid flow implemented with two opcodes, using the identifiers defined in the `opcode_map`. Additional valid examples for *output C stationary* and *nothing stationary* flows are shown in lines 24 and 25 of Figure 6a. The information in `opcode_flow` is parsed and the set of parentheses is understood as a proxy to specify multiple scopes for sequential or nested *for* loops in the algorithm. Following this flow, logic related to *"sA"* would be transmitted inside of the second loop (Figure 6b-L8 to L10), and logic related to *"sBcCrC"* would appear in the innermost loop (Figure 6b-L12 to L18). Suppose the user decides to forego the opportunity to specify *input A* as stationary, then the opcode flow could become *"(sA sB cC rC)"*, and all communication driver logic would be generated in the innermost loop.

The `accel` dialect: Before generating function calls for *runtime replacement* to the DMA runtime library (described in Section III-A), we perform *host code transformations* ⑤ (Figure 4) by lowering the `linalg.generic` operation, with the proposed `trait`, to standard MLIR dialects (`scf`, `arith`, `memref`) and a new dialect that we call `accel`. Operations in the `accel` dialect abstract host-accelerator transactions, such as initialization, memory transfers, and synchronization. Figure 9 presents the core `accel` operations and their semantics, providing examples of how these operations map onto our custom AXI DMA library calls. Additionally, Figure 6b shows how the `accel` operations are used in our MatMul example.

Note that it is easier to perform analysis and transformations of operations when they are expressed in our `accel` dialect, as opposed to using a lower-level abstraction. With lower-level abstractions such as `llvm`, function calls and additional logic have already been exposed: additional instructions must be present in the IR to implement buffer slicing, size/offset calculations, and function calls to copy data to/from the DMA regions. Performing analysis and transformations in the `llvm` abstraction is more challenging, as traversal of control flow blocks and LLVM instructions are necessary. Instead, operations in the intermediate `accel` dialect encode the relevant information, and are easily relocated during transformation passes, respecting dependencies without requiring complex compiler analysis. This approach facilitates implementing communication flows that consider one of the data structures to be stationary by simply hoisting the `accel` operations up to the right loop nest level, while considering the flow patterns. Finally, the `accel` dialect provides an intermediate step before runtime call replacement. In this work we target our AXI DMA runtime library described in Section III-A, but further extensions could implement the transformation of `accel` operations into other runtime libraries such as OpenCL [27] or SYCL [28], which are commonly used to interface with SoC FPGA accelerators.

IV. Experiments and Results

To evaluate AXI4MLIR, we use a PYNQ-Z2 board that includes a Zynq-7000 SoC with a dual-core ARM Cortex-A9 CPU (650 MHz), and a library of tile-based accelerators derived from SECDA-TFLite [29] implemented with AXI-S interface and opcodes with a micro-ISA. For workloads, we target a suite of kernels covering a range of dimensions, as well as an end-to-end machine learning application. We leverage hand-written baselines, which we discuss in Section IV-A. Section IV-B evaluates accelerators implementing MatMul, comparing inference performance against a hand-written baseline, identifying potential bottlenecks, and showcasing the benefits of our optimized dataflows. Section IV-C highlights the value of AXI4MLIR by demonstrating how to handle accelerators with configurable parameters such as tile sizes and dataflows. We showcase how to use AXI4MLIR with a convolution-based accelerator in Section IV-D. Finally, Section IV-E shows how AXI4MLIR can work in the context of a complete application, evaluating the TinyBERT model [30].

A. Hand-written Baselines

The next experiments employ hand-written optimized driver code derived from the SECDA-TFLite accelerator toolkit [29] to establish performance baselines. SECDA-TFLite presents a state-of-the-art toolchain and methodology for HW/SW co-design of embedded machine learning accelerators targeting FPGA SoC devices. With host-driver code written in C++, these manual baselines will be labeled as *cpp_MANUAL*. All baselines are implemented with various tiling strategies, with no additional data transfer overheads and with the fewest number of data transfer calls for the selected dataflow.

B. Matrix-Multiplication Experiments

The tile-based accelerators used here resemble vector MAC engines [31], [32], [33], [34] implementing MatMul algorithms. They vary in input/output buffer size and supported dataflow. From the CPU-host perspective, some of them can support varying degrees of data reuse when the appropriate opcode stream drives the accelerator. Table I presents a short summary of their functionality, where *size* stands for the supported tile size of the accelerator. For example, $\mathrm{v1}_4$ is a MatMul_{4x4x4} accelerator that does not support data reuse and only supports $tM, tN, tK == 4, 4, 4$ tiles. For $\mathrm{v1}_4$, AXI4MLIR will tile the algorithm's loops in the host code, taking into account the accelerator size of 4 and all the data movement will happen in the innermost loop - "`opcode_flow <(sA sB cCrC)>`". For $\mathrm{v2}_8$, AXI4MLIR will tile the computation by 8 and generate code to maximize the reuse of one of the inputs. In v2, a stationary (As) is implemented with `opcode_flow <(sA (sB cCrC))>`.

Accelerators v3 and v4 can also reuse their output data structures. Accelerator v4, marked with *flex size*, supports computations of non-square tiles, i.e., $\mathrm{v4}_{16}$ can process a MatMul of $tM, tN, tK = 32, 16, 64$, as long as tM, tN, tK are divisible by 16 and fit in the accelerator's memory. All accelerators were implemented using HLS pipelining and unrolling to maximize the number of internal processing elements instantiated and their arithmetic throughput. The last column of Table I reports throughput (OPs/cycle) for each accelerator, highlighting that many arithmetic operations are executed in parallel at each cycle. Different types of accelerators with the same size have the same throughput, and accelerators with bigger sizes provide higher throughput. All bar graphs presented in this section represent the average of 5 independent runs with the same configuration.

Accelerator relevance. In order to evaluate the performance of the accelerators defined in Table I, we conducted experiments to compare the runtime of the CPU execution (*mlir_CPU*) against the manual C++ implementation (referred to as *cpp* for short) of the driver code using the accelerators. The task clock was used as a metric to measure the execution time of the benchmarks. We present the results of the experiments in Figure 10, which plots the task clock on the y-axis (smaller is better) and only includes the "Nothing Stationary flow", which means that the data transfers happen in the innermost loop.

Looking at Figure 10, we can see that the accelerator offload only becomes relevant (i.e., executes faster than the CPU) for problems with $dims \geq 64$, where $dims = M = N = K$. For problems with smaller dimensions, CPU execution will be faster than the accelerator. In addition, the results in Figure 10 suggest that accelerators only become relevant if $accel_size = tM = tN = tK \geq 8$. For smaller accelerator sizes, the CPU execution is faster than the accelerator.

These observations suggest that the performance benefits of using the accelerators are limited for ranges of problem sizes and accelerator sizes. Therefore, it is important to carefully

TABLE I: Accelerators used in the experiments. Synthesized with AMD/Xilinx Vitis at 200MHz.

Type	Possible Reuse	Opcode(s)
$\mathrm{v1}_{\mathrm{size}}$	Nothing	sAsBcCrC
$\mathrm{v2}_{\mathrm{size}}$	Inputs	sA, sB, cCrC
$\mathrm{v3}_{\mathrm{size}}$	Ins/Out	sA, sB, cC, rC
$\mathrm{v4}_{\mathrm{size}}$	Ins/Out (flex size)	sA, sB, cC, rC

Configurations (Size, OPs/Cycle)
(4, 10)
(8, 60)
(16, 112)

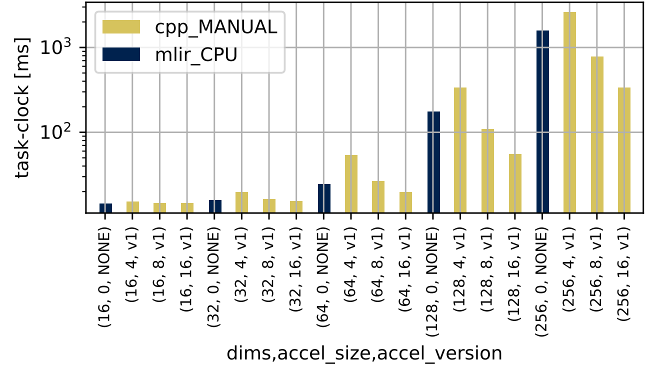


Fig. 10: Runtime characterization CPU vs. Accelerator execution for Matrix Multiplication problems. Note how an accelerator only becomes relevant for problems with $dims \geq 64$ and $accel_size \geq 8$.

choose the appropriate accelerator configuration for a given problem to achieve the best performance. Consequently, for the next experiments we will limit our focus to problems with $dims \geq 64$ and accelerators with $accel_size \geq 8$.

AXI4MLIR generated vs. Manual implementation.

AXI4MLIR provides several benefits. First, our passes automatically tile data mapped to the CPU memory hierarchy, leveraging spatial and temporal locality. The second benefit is the ability to automatically generate specific flows, such as the Nothing Stationary (Ns) flow, which can be tedious and error-prone when done manually. Additionally, AXI4MLIR provides an efficient path to flow strategies that can potentially improve performance, such as input A or B stationary (As, Bs) flows. Figure 11 presents these results.

First, we compare the differences in execution time between a *manual implementation* (see Section IV-A) of an Ns flow strategy and an *AXI4MLIR generated* Ns flow strategy, represented by the first two bars in each group of bars in Figure 11. The remaining bars in each group of bars show results for automatically generated flow strategies, with As and Bs for v2 accelerators and As, Bs, and Cs for v3 accelerators. Looking at Figure 11 we see that some flows, especially Cs, provide improvements. To achieve this, the user simply has to encode the information for Cs (or other flows) during compilation. For example, we can encode Cs using the opcode_flow previously presented in Figure 6a-L25 in the the operation's trait.

Next, in Figure 11, we focus on the results with the v3 accelerator. Here, we see that AXI4MLIR generated Cs performs better than the manually generated Ns, although the other flows are not performing as expected. First, we would expect the performance of AXI4MLIR generated Ns to have similar/closer task clock performance than manual Ns. And second, we would also expect As and Bs flows to always outperform Ns due to the degree of reuse, as they copy less

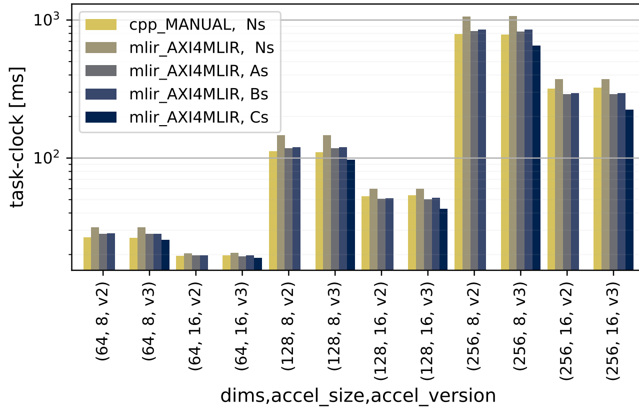
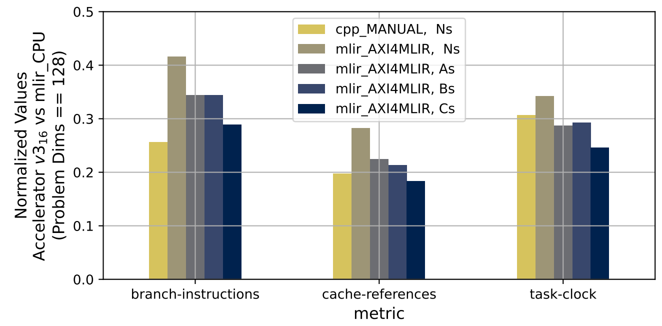
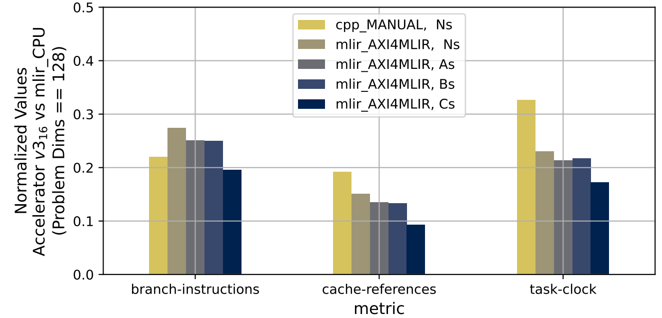


Fig. 11: Runtime results on Matrix Multiplication kernels. Manual implementation of Ns flow vs. AXI4MLIR generated driver code for different flow strategies, Ns, As, Bs, Cs. All bar groups follow similar trends. Ns, As, and Bs **bottlenecks are analysed and addressed** in following experiments.



(a) Without the MemRef-DMA buffer copy optimization. Generated host-accelerator code has overheads if not specialized.



(b) With MemRef-DMA buffer copy optimization. AXI4MLIR improves accelerator performance over manual Ns implementation.

Fig. 12: Cache, branch, and runtime metrics of different tools and strategies using $v3_{16}$ accelerator with problem size ($dims == 128$). Normalized values against CPU (without accelerator) executions of same problem size.

data and can keep the accelerator better utilized. Hence, this first implementation has room for improvement and, in the following experiment, we *identified and fixed the bottlenecks* by analyzing performance counters and implementing optimizations that specialize memory copies.

Identifying bottlenecks & improving AXI4MLIR codegen. Next, we identify performance bottlenecks in AXI4MLIR generated copies and improve upon them to enhance the performance of the workloads when using the custom hardware accelerators. Specifically, the experiment compares the performance of manually implemented host-accelerator driver code with AXI4MLIR generated code for Ns, As, Bs, and Cs flows in terms of branch-instructions, cache reference counters, and the task clock. These metrics were obtained using the `perf` tool [35] to profile the application and retrieve counters for CPU perf_events over 5 runs.

Figure 12a shows branch instructions, cache reference counters, and the task clock for $dims == 128$, for the $v3_{16}$ accelerator that supports input and output stationary flows. The trends are similar to other problem and accelerator sizes. Our results are normalized to the same counters collected on a CPU-only execution of the same problem size. In each group we show results for AXI4MLIR automatically generated code for Ns, As, Bs, Cs flows, and compare against manual implementations (first bar of a group) for copying the necessary data through the DMA memory-mapped region. MLIR applications have to consider MLIR memory references (presented in Section II-A1), but manual implementations use bare C-arrays. To support generality, MLIR copies between MemRef and the raw array (DMA buffer region) are implemented with a recursive call, loading and storing one element at a time. This is necessary to support $rank = N$ MemRefs, where strides in all dimensions are different from 1.

In order to address this issue, we implemented an optimization for when $strides[N-1] == 1$ (i.e., elements in $N-1$ dimension are adjacent to each other in memory) and specialized MemRef copies for some known rank sizes, such as $rank == 2$. For this scenario, we leverage the spatial locality and implement the copy not with individual load and store instructions, but by calling `std::memcpy(src, dst, size)`. When compiling this function for our platform, the compiler will inline the assembly, implementing a vectorized copy that improves the performance of the copy operation. The implications of this optimization are twofold. First, it reduces the number of branch references because there is no need for branching to handle non-unitary strides or to iterate over an arbitrary number of dimensions, resulting in better control flow and branch prediction. Second, the vectorized code reduces the number of cache references because the data is accessed sequentially in memory. Therefore, there will only be two cache reference to fetch the cache line containing the requested data, and subsequent loads within the same cache line will not require additional cache references as they are read from the vector VFP registers [36]. The results for this optimization are presented in Figure 12b.

After incorporating this optimization, the AXI4MLIR generated driver code executed faster on all accelerators as compared to their respective manual implementations. In Figure 13, we compare AXI4MLIR against manual implementations for Ns, As, Bs, and Cs, and found that the compiled generated driver code provided by AXI4MLIR is consistently faster (1.18× average speedup and 1.65× max speedup), thanks to its ability to leverage proper tiling for the CPU's memory hierarchy, resulting in a 10% average and 56% max reduction in cache references.

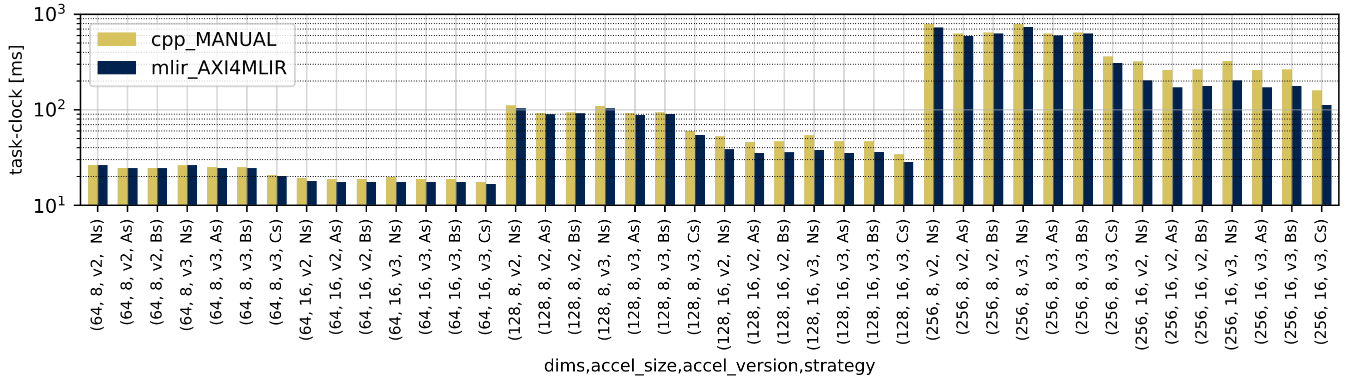


Fig. 13: Runtime comparison of manual implementation of driver code and AXIMLIR generated. Each set of two bars have a matching Accelerator Type, Accelerator Size, and Flow Strategy (Ns, As, Bs, Cs). AXI4MLIR is better in all cases.

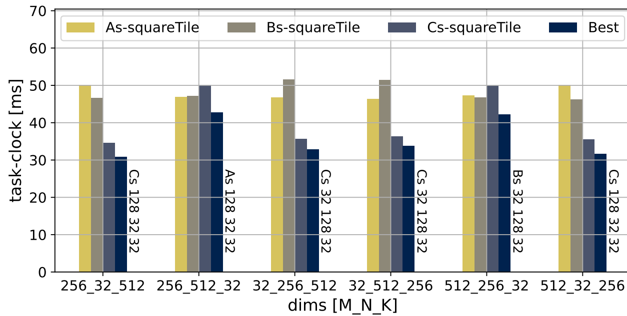


Fig. 14: MatMul problem permutations (v4 accelerator) for different strategies. For the "Best" strategies we annotate the chosen flow and tiling values.

C. *Matrix-Multiplication with flexible sizes*

Runtime configurable accelerators allow for fine-grained hardware tuning for specific problems. With AXI4MLIR, we can generate host code to configure and optimize flexible accelerators for the target problem. To demonstrate this capability, we evaluate multiple permutations of a MatMul problem on the $\text{v}4$ accelerator. The $\text{v}4$ accelerator supports multiple dataflow strategies and adjustable tile sizes for its tM, tN, and tK dimensions. The intuition is that scientific and machine learning workloads present problem sizes with different values for each dimension, sometimes resulting in tall/skinny matrices during execution. Tiling the problem in the accelerator with different dimensions for tM, tN, and tK, and selecting the appropriate flow strategy can be beneficial for the application.

When using AXI4MLIR, a developer is *not* limited to one configuration of an accelerator. Based on the user's knowledge of the application, AXI4MLIR can automatically generate the driver for accelerators with adjustable dimensions. This flexibility allows for a more thorough exploration of the design space, enabling the developer to find the best sizes for tM, tN, tK, and the best flow strategy for each problem instance.

In Figure 14, we compare four different heuristics and use them to choose the best tiling and dataflow configuration for a MatMul problem. We evaluate performance in terms of execution time. We profile the problem with M,N, and K dimensions permuted from the following values: $[32, 256, 512]$. Hence, the theoretical minimum number of multiply-accumulate operations required for all permutations is the same. Here, the *As-squareTile*, *Bs-squareTile*, and *Cs-squareTile* heuristics try to find the best configuration to reduce the total memory access count given the constraint of tiling the MatMul with square tiles (i.e., $tM = tN = tK = T$), with A, B, and C stationary dataflow, respectively. The fourth heuristic, *Best*, chooses between all dataflows and flexible tiling options, only sharing the choice of the accelerator. In Figure 14 we annotate the "*Best*" configuration found for each problem.

Square tiling. We observe that as we change the problem permutation, the best flow between *As-squareTile*, *Bs-squareTile*, and *Cs-squareTile* tiling strategies changes. The best flow depends on the problem shape, the size, and the available accelerator buffer space. $T = 32$ was selected for all square flows because it is the biggest value, so the tiles fit inside the accelerator's internal memory.

Flexible tiling. The *Best* heuristic, selected from non-square strategies, outperforms square tiling by leveraging flexible tiling sizes. AXI4MLIR can generate code to utilize larger tile sizes in various dimensions, taking advantage of the $\text{v}4$ accelerator's unrestricted tiling factors and improving the accelerator's internal memory utilization.

Configurations. Manually implementing all configurations' driver code for even a simple accelerator such as $\text{v}4$ is very time-consuming. AXI4MLIR can quickly generate hostcode for configurable accelerators easily, enabling the developer to specify an accelerator configuration per problem instance.

D. *Convolution*

We show the flexibility of AXI4MLIR by generating driver code for a convolution-based accelerator executing different problems sizes. This accelerator supports varying input channel (*iC*) and filter (*fHW*) sizes, computing one output slice (all elements in one output channel - *oC*) per iteration. To orchestrate the execution, multiple instructions have to be sent to the accelerator. This orchestration is achieved by compiling the driver code derived from the MLIR `accel` code (Figure 15b). The `accel` code is generated after a transformation pass takes into account the attributes shown in Figure 15a and MLIR's `linalg.conv_2d_nchw_fchw` operations. Note that if the convolution operation has *iC*, *fH*, *fW* dimensions that are smaller than the dimensions in `accel_dim`, no tiling

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

(a) Opcode Map and Flow for Conv2D accelerator.

```
func.func @conv_call(...) {
  // With %I: !mrI_1_256_7_7; %W: !mrW_64_256_3_3
  // and  %O: !mrO_1_64_5_5
  // Declare constants (loop bounds and literals): %cX, ...
  accel.dma_init(%c0,%c66,%c65280,%c65346,%c65280) : ...
  accel.sendLiteral(%c32, %c0) : i32,i32->i32 // send inst
  accel.sendDim(%W,%c3,%c0) : !mrW,i32,i32->i32 // send %fH
  accel.sendLiteral(%c16, %c0) : i32,i32->i32 // send inst
  accel.sendDim(%I,%c1,%c0) : !mrI,i32,i32->i32 // send %iC

  // Tile dims by (B,H,W,iC,oC,fH,fW) -> (-,-,-,256,1,3,3)
  scf.for %b = %c0 to %c1 step %c1 { // B loop
    scf.for %oc = %c0 to %c64 step %c1 { // OC loop
      %sW = memref.subview %W[%oc,0,0,0][1,256,3,3] ...
      %offset0 = accel.sendLiteral(%c1, %c0) : i32,i32->i32
      %offset1 = accel.send(%sW, %offset0) :
                                     !mrSubWx256x3x3, i32 -> i32
      scf.for %oh = %c0 to %c5 step %c1 { // OH loop
        scf.for %ow = %c0 to %c5 step %c1 { // OW loop
          %xoffset = ... // index calculation
          %yoffset = ... // index calculation
          %sI = memref.subview %I[0,0,%xoffset,%yoffset]
                                     [1,256,3,3] ...
          %offset2 = accel.sendLiteral(%c70, %c0) : ...
          %offset3 = accel.send(%sI, %offset2) :
                                     !mrSubIx256x3x3, i32 -> i32
          // inner product of sW and sI computed in HW
      } }
      %sO = memref.subview %O[0,%oc,0,0] [1,1,5,5] ...
      %offset4 = accel.sendLiteral(%c8, %c0) : ...
      accel.recv {mode="accumulate"}(%sO, %c0) :
                                     !mrSubO_5x5_0, i32 -> i32
} } } } return }
```

(b) IR to drive the Conv2D accelerator with an output-stationary flow.

Fig. 15: Information added to the linalg.generic traits to capture convolution accelerator behavior in MLIR and IR with accel operations.

will be performed across these dimensions. In the convolution example (Figure 15), upon accelerator reset, we use `send_dim(1,3)` to send to the accelerator the filter size as the dimension '3' of data structure '1' (i.e., the filter), and we use `send_dim(0,1)` to send the input channel size as the dimension '1' of the data structure '0' (i.e., the input).

We evaluate the performance of AXI4MLIR during the execution of all convolution layers of ResNet18 [37]. Figure 16 presents performance metrics normalized to the runtime of layer-specific manual C++ driver code. The results observed here present similar trends as observed in the MatMul experiments. Only one layer (*56_64_1_128_2*) presented a 10% slowdown, contrary to previous trends. The slowdown happened because *fHW (1)* and *iC (64)* were too small, and the overhead of dealing with small MemRefs was not overcome since we could not leverage the *strided copy optimization* presented in Section IV-B. Smaller AXI4MLIR speedups are observed every time that *fHW*== **1**. That said, AXI4MLIR achieves better runtime performance on 10 out of 11 ResNet18 layers, with 1.28× and 1.54× average and max speedup, respectively, thanks to the improved CPU cache performance.

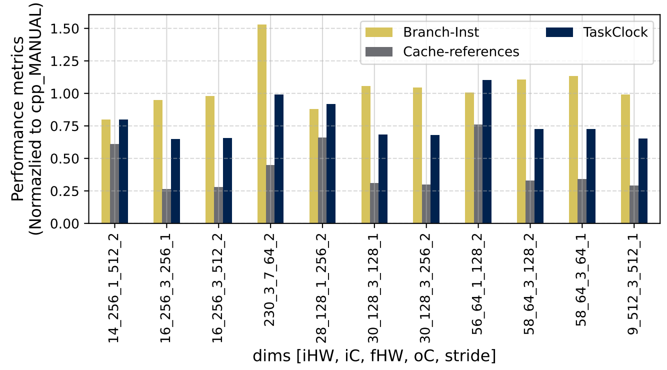


Fig. 16: ResNet18 convolution layers: AXI4MLIR vs. Manual.

E. End-To-End Analysis

Finally, we evaluate AXI4MLIR when compiling a natural language processing model to co-execute on both the CPU and the $\mathrm{v4}_{16}$ accelerator. We benchmark TinyBERT [30], a compact transformer [38] model for Masked Language Modeling and Next Sentence Prediction targeted at mobile and embedded devices. We translate TinyBERT to MLIR IR using Torch-MLIR [39] and compare the inference performance of CPU execution (using -O3 during compilation) against co-execution using the *"Ns"* offloading approach and the *"Best"* approach, which employs the heuristics presented in Section IV-C.

As we can see in Figure 17, AXI4MLIR achieves a 3.4× speedup in end-to-end execution, with an 18.4× speedup in the accelerated MatMul layers that represent 75% of the original CPU runtime. This experiment showcases how AXI4MLIR can be used during evaluation and optimization of natural language processing models on embedded devices. Our study highlights that developers can easily co-design the accelerators when targeting full workloads, which enables efficient exploration and utilization of both CPU and accelerator resources.

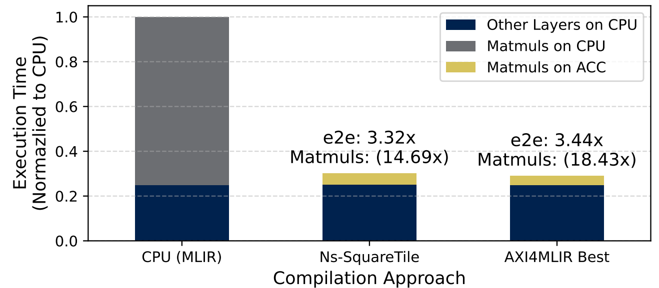


Fig. 17: Execution time of the TinyBERT model with `batch_size` == 2. Each bar represents a compilation strategy. Speedups for end-to-end ($e2e$) and for accelerated MatMul layers are shown as annotations.

V. Related Work

Prior studies [40], [17], [11], [16], [13], [41], [42], [43] have proposed new accelerator designs or presented new methodologies to generate flexible accelerators for a wide range of algorithms. However, these approaches fall short in providing insights into how host-to-accelerator transfers should be performed. Most of these tools assume that the required data

is already placed in the accelerator's internal buffers. There have also been efforts to support hardware/software co-design of an accelerator for an application [18], [19], [29]. However, these implementations adopt a simple offload model, where execution of the kernel code is simply *replaced* by runtime calls that copy the data to-and-from the accelerator, *without considering the host memory hierarchy or accelerator features*, which would require manual driver code modifications.

HeteroFlow [44], an FPGA accelerator programming model, decouples algorithm specification from data placement optimization using a new primitive "`.to()`". This approach exposes data placement specification at various granularities, achieving efficient code generation while matching optimized manual HLS designs. HeteroFlow does not support arbitrary custom accelerators, as it is limited to accelerators co-designed with its framework (extended HeteroCL [45]). It also requires the new primitive to be used while describing the algorithm in Python, imposing manual application modification. Unlike HeteroFlow, AXI4MLIR utilizes MLIR to target languages employing `linalg.generic` operations during compilation, eliminating the need for manual transformation.

Several other studies have addressed the challenge of efficiently mapping algorithms and their loops onto accelerators through operation scheduling. Notably, Interstellar [46], DMazeRunner [47], and PolySA [48] delve into more versatile loop structures by adopting diverse loop representations for DNN layers. CoSA [49] and Vaidya et al. [50] tackle the generation of execution schedules for DNNs in a time-efficient manner, leveraging constrained optimization solvers. Self-tuning algorithms have also been employed in addressing the scheduling problem. Approaches like ConfuciuX [51], FlexTensor [52], AutoTVM [53], and Ansor [54] utilize machine learning algorithms. Furthermore, Flexer [55] employs an out-of-order scheduling technique, unbound by loop order, which orchestrates operations based on a comprehensive analysis of the data-flow graph of a given layer. Some of these works are tailored to a specific type of accelerator or algorithm. In addition, these works primarily focus on scheduling aspects, which AXI4MLIR currently lacks as a component. Nonetheless, these scheduling techniques could potentially complement AXI4MLIR's attributes and transformations to enhance the overall accelerator integration process.

The Pattern Description Language (PDL) and Transform MLIR dialects [56] offer productive ways for expressing IR transformations and could be leveraged to implement similar functionality as provided by AXI4MLIR. However, PDL cannot currently identify patterns in nested MLIR regions. Additionally, the transform dialect focuses on scheduling linear algebra transformations but requires extensions for runtime call generation targeting accelerators and dataflows. In contrast, AXI4MLIR's `opcode_map` and `opcode_flow` extensions enable flexible automatic driver code generation for custom accelerators. Future work may involve integrating AXI4MLIR passes as Transform operations and using PDL to identify operation sequences for transformation, potentially supporting fusing operations for custom accelerator execution.

Host code generation transforms accel operations into DMA library calls. To facilitate further optimizations leveraging the MLIR infrastructure, users can modify these transformation passes while applying optimizations such as double buffering, building on our infrastructure supporting non-blocking transfers and transfer completion checks. Our ongoing work will introduce a new attribute to select inputs/outputs for double buffering. This aligns with MLIR's capability to modify and add passes to the transformation pipeline. For further efficiency, coalescing transfer requests are essential; future work will implement a transformation that consolidates multiple `start_send` calls into a single call after data preparation, thus reducing the need for multiple `wait_send` calls, which incur higher CPU accelerator-DMA synchronization costs.

VI. Conclusion

In this paper we presented AXI4MLIR, an extension to the MLIR compiler framework to describe AXI-based accelerators with a range of features including accelerator opcodes. We implemented attribute extensions and compiler transformations to describe and automatically generate host code that can leverage different flows of flexible accelerators, allowing us to break away from simple offload HW/SW co-design models. After implementing data staging and accessing optimizations during communication, our results show that AXI4MLIR is effective in generating host code that efficiently uses CPU resources and accelerator features. This allows for measurable runtime improvements versus manual implementations for all tested accelerators, while providing automation and convenience during the co-design cycle. Finally, our user-driven host code generation is entirely automated, providing a significant advantage in terms of productivity and maintainability, specially during the early stages of the co-design process.

References

[1] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. Los Angeles, CA, USA: IEEE, 2018, pp. 27–29.

[2] H. Shabani, A. Singh, B. Youhana, and X. Guo, "Hirac: A hierarchical accelerator with sorting-based packing for spgemms in dnn applications," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Montreal, QC, Canada: IEEE, 2023, pp. 247–258.

[3] B. Kim, S. Li, and H. Li, "Inca: Input-stationary dataflow at outside-the-box thinking about deep learning accelerators," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Montreal, QC, Canada: IEEE, 2023, pp. 29–41.

[4] J. Zhao, Y. Yang, Y. Zhang, X. Liao, L. Gu, L. He, B. He, H. Jin, H. Liu, X. Jiang, and H. Yu, "Tdgraph: A topology-driven accelerator for high-performance streaming graph processing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 116–129. [Online]. Available: https://doi.org/10.1145/3470496.3527409

[5] S. Hsia, U. Gupta, B. Acun, N. Ardalani, P. Zhong, G.-Y. Wei, D. Brooks, and C.-J. Wu, "Mp-rec: Hardware-software co-design to enable multi-path recommendation," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 449–465. [Online]. Available: https://doi.org/10.1145/3582016.3582068

880 [6] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. Liang, "Amos: Enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 874–887. [Online]. Available: https://doi.org/10.1145/3470496.3527440

887 [7] F. Muñoz Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, "Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 252–265. [Online]. Available: https://doi.org/10.1145/3582016.3582069

895 [8] M. Abolhasani and E. Kumacheva, "The rise of self-driving labs in chemical and materials sciences," *Nature Synthesis*, vol. 0, no. 0, pp. 1–10, 2023.

898 [9] Q. Rao and J. Frtunikj, "Deep learning for self-driving cars: Chances and challenges," in *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*, ser. SEFAIS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 35–38. [Online]. Available: https://doi.org/10.1145/3194085.3194087

903 [10] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

907 [11] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W. Hwu, and D. Chen, "DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator," in *ICCAD*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–9.

911 [12] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin, "Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: Association for Computing Machinery, 2020, p. 40–50. [Online]. Available: https://doi.org/10.1145/3373087.3375306

918 [13] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybriddnn: A framework for high-performance hybrid DNN accelerator design and implementation," in *DAC*. San Francisco, CA, USA: IEEE, 2020, pp. 1–6.

922 [14] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

926 [15] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

930 [16] TVM Developers, "VTA: Deep learning accelerator stack," 2020. [Online]. Available: docs.tvm.ai/vta

932 [17] J. Ngadiuba, V. Loncar, M. Pierini, S. Summers, G. Di Guglielmo, J. Duarte, P. Harris, D. Rankin, S. Jindariani, M. Liu *et al.*, "Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml," *ML: Science and Technology*, vol. 2, no. 1, p. 015001, 2020.

936 [18] S. Skalicky, J. Monson, A. Schmidt, and M. French, "Hot & Spicy: Improving Productivity with Python and HLS for FPGAs," in *FCCM*. Boulder, CO, USA: IEEE, 2018, pp. 85–92.

939 [19] N. Bohm Agostini, S. Dong, E. Karimi, M. T. Lapuerta, J. Cano, J. L. Abellán, and D. Kaeli, "Design space exploration of accelerators and end-to-end dnn evaluation with tflite-soc," in *SBAC-PAD*. Porto, Portugal: IEEE, 2020, pp. 10–19.

943 [20] ARM Developers, "AMBA AXI and ACE Protocol Specification," 2020. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification

946 [21] S. Liu, H. Fan, X. Niu, H.-c. Ng, Y. Chu, and W. LUK, "Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, dec 2018. [Online]. Available: https://doi.org/10.1145/3242900

951 [22] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *CGO*. Seoul, Korea (South): IEEE, 2021, pp. 2–14.

955 [23] M. Developers, "'linalg' Dialect," 2020, online accessed on 11-04-2023. [Online]. Available: https://mlir.llvm.org/docs/Dialects/Linalg/

957 [24] T. D. Le, G.-T. Bercea, T. Chen, A. E. Eichenberger, H. Imai, T. Jin, K. Kawachiya, Y. Negishi, and K. O'Brien, "Compiling ONNX Neural Network Models Using MLIR," *ArXiv*, vol. 0, no. 0, pp. 1–8, 2020.

960 [25] Xilinx, "AXI Reference Guide," 2012. [Online]. Available: https://docs.xilinx.com/v/u/14.1-English/ug761_axi_reference_guide

962 [26] C. Salvador Rohwedder, N. Henderson, J. a. P. L. De Carvalho, Y. Chen, and J. N. Amaral, "To pack or not to pack: A generalized packing analysis and transformation," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 14–27.

968 [27] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engg.*, vol. 12, no. 3, p. 66–73, may 2010.

971 [28] R. Reyes, G. Brown, R. Burns, and M. Wong, "Sycl 2020: More than meets the eye," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020.

975 [29] J. Haris, P. Gibson, J. Cano, N. Bohm Agostini, and D. Kaeli, "SECDA-TFLite: A toolkit for efficient development of FPGA-based DNN accelerators for edge inference," *Journal of Parallel and Distributed Computing*, vol. 173, pp. 140–151, 2023.

979 [30] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv preprint arXiv:1909.10351*, vol. 0, no. 0, pp. 1–12, 2019.

982 [31] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

986 [32] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE. Taipei, Taiwan: IEEE, 2016, pp. 1–12.

990 [33] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE. Cambridge, UK: IEEE, 2014, pp. 609–622.

994 [34] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 161–170.

999 [35] The Linux Perf Team, "Perf wiki," n.d., accessed on April 13, 2023. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

1001 [36] A. Developer, "Neon registers," 2023. [Online]. Available: https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/NEON-architecture-overview/NEON-registers

1004 [37] F. Wang, M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang, and X. Tang, "Residual attention network for image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, Honolulu, HI, USA, 2017, pp. 3156–3164.

1008 [38] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

1017 [39] Torch-MLIR Developers, "The Torch-MLIR Project," 2021. [Online]. Available: https://github.com/llvm/torch-mlir

1019 [40] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A Unified Backend for Targeting FPGAs from DSLs," in *ASAP*. Milan, Italy: IEEE, 2018, pp. 1–8.

1022 [41] N. Bohm Agostini, S. Curzel, J. Zhang, A. Limaye, C. Tan, V. Amatya, M. Minutoli, V. G. Castellana, J. Manzano, D. Brooks, G.-Y. Wei, and A. Tumeo, "Bridging python to silicon: The soda toolchain," *IEEE Micro*, vol. 42, no. 5, 2022.

1026 [42] N. Bohm Agostini, S. Curzel, V. Amatya, C. Tan, M. Minutoli, V. G. Castellana, J. Manzano, D. Kaeli, and A. Tumeo, "An mlir-based compiler flow for system-level design and hardware acceleration," in *ICCAD*. New York, NY, USA: Association for Computing Machinery, 2022.

1031 [43] A. Stjerngren, P. Gibson, and J. Cano, "Bifrost: End-to-End Evaluation and optimization of Reconfigurable DNN Accelerators," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Singapore: IEEE, May 2022, pp. 288–299.

1035 [44] S. Xiang, Y.-H. Lai, Y. Zhou, H. Chen, N. Zhang, D. Pal, and Z. Zhang, "Heteroflow: An accelerator programming model with decoupled data placement for software-defined fpgas," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 78–88.

1041 [45] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 242–251.

1047 [46] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 369–383.

1054 [47] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "Dmazerunner: Executing perfectly nested loops on dataflow accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, oct 2019.

1057 [48] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. San Diego, CA, USA: IEEE, 2018, pp. 1–8.

1061 [49] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, 2021, pp. 554–566.

1066 [50] M. Vaidya, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Comprehensive accelerator-dataflow co-design optimization for convolutional neural networks," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Seoul, South Korea: Association for Computing Machinery, 2022, pp. 325–335.

1071 [51] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Athens, Greece: IEEE, 2020, pp. 622–636.

1075 [52] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 859–873.

1082 [53] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/8b5700012be65c9da25f49408d959ca0-Paper.pdf

1089 [54] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating High-Performance tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

1093 [55] H. Min, J. Kwon, and B. Egger, "Flexer: Out-of-order scheduling for multi-npus," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 212–223.

1098 [56] M. Developers, "Transform Dialect: Fine-grain transformation control dialect," 2022, online accessed on 11-04-2023. [Online]. Available: https://mlir.llvm.org/docs/Dialects/Transform/ 1100