An End-to-End Tool Flow with Intrinsic-Level Kernel Optimization on Versal ACAP

Liyang Dou[1], Zhe Lin[2†], Kai Shi[1], Xinya Luan[1], and Kang Zhao[1†],
[1]Beijing University of Posts and Telecommunications [2]Sun Yat-sen University
{leondou, shikai, luanxinya, zhaokang}@bupt.edu.cn, linzh235@mail.sysu.edu.cn

***Abstract*—To meet the growing demand for machine learning workloads, AMD introduced the Versal Adaptive Compute Acceleration Platform (ACAP). Although this platform delivers high throughput, fully exploiting the single-instruction-multiple-data (SIMD) and very-long-instruction-word (VLIW) parallelism of the individual AI Engine (AIE) poses significant challenges to developer productivity and hardware utilization. To address these issues, this paper proposes an end-to-end compilation tool flow that automatically generates optimized AIE kernels from C++ code. It is based on the MLIR framework, eliminating the need for manual SIMD and VLIW programming. To further enhance this tool flow, a ping-pong chaining mechanism is proposed to reduce the memory-register interaction and improve the register reuse. Our work currently supports 1D operators (e.g., element-wise multiplication) and 2D operators (e.g., convolution, matrix multiplication). Compared to the MLIR-AIE baseline, our approach achieves up to 1.33× speedup for small-scale convolution workloads and consistently achieves a 1.08× speedup for medium and large-scale cases.**

***Index Terms*—Versal ACAP, AIE Architecture, SIMD, Compilation Tool Flow, MLIR**

I. Introduction

The rapid advancement of AI technologies [1] has led to increasing demand for hardware systems that support massive parallelism and high memory throughput. Traditional monolithic architectures fail to satisfy these demands [2], driving a shift toward heterogeneous computing [3], [4] and domain-specific hardware designs [5], [6]. To address these challenges, AMD introduced the Versal ACAP platform [7], which improves edge-side computational acceleration while improving energy efficiency. Architecturally, the Versal ACAP platform integrates AI Engines (AIEs) for high-performance computing, programmable logic (PL) for architectural flexibility, and a processing system (PS) for control and coordination, collectively enabling both flexibility and performance [8].

Application development on Versal ACAP follows a kernel-centric model, where performance-critical kernels are written using AIE-specific intrinsics and compiled with AMD's Vitis toolchain. The intrinsic API exposes low-level hardware details of the AIE single core, which features a 2D SIMD [9] and VLIW [10] architecture. This requires developers to explicitly manage SIMD vectorization, register allocation, data selection, and instruction scheduling to fully exploit hardware capabilities. This fine-grained control introduces two primary challenges.

†Corresponding author.

Challenge 1: Programming Complexity. Using low-level intrinsics that expose architectural details can lead to a steep learning curve and limit developer productivity.

Challenge 2: Hardware Utilization. Using the 2D SIMD and VLIW features effectively requires careful manual tuning of parallelism and instruction scheduling, making performance optimization difficult.

Several efforts have attempted to address these challenges. AMD's early project, Vyasa [11], is a single-core compiler for the AI Engine (AIE) that focuses on convolution operations. Based on the Halide framework, Vyasa provides users with a Domain-Specific Language (DSL) and incorporates a set of optimization passes tailored for convolution workloads. However, Vyasa has two main limitations: (1) it is limited to convolution operations only; (2) since AMD has now adopted the MLIR ecosystem for its Versal toolchain, Vyasa's Halide-based design is difficult to integrate into multi-AIE compilation workflows. A more recent effort, MLIR-AIE [12], extends the MLIR framework to model the AIE computation, incorporating Vyasa's optimizations. While offering greater extensibility, MLIR-AIE also faces challenges: (1) it lacks a fully end-to-end compilation flow, preventing high-level programming of AIEs; and (2) like Vyasa, it currently supports only convolution workloads, with optimizations limited to automatic vectorization and simple register optimization, resulting in suboptimal hardware resource utilization.

To reduce programming complexity, we propose an end-to-end compilation tool flow based on MLIR-AIE that automatically translates C++ code into AIE intrinsics. To enhance hardware utilization, we propose pattern-specific optimization passes for different workloads. For the sliding-window pattern in convolution, a ping-pong chaining mechanism is proposed, which, when combined with loop unrolling, reduces memory-register interaction overhead. For scalar-vector multiplication in vectorized matrix multiplication, we apply a memory coalescing optimization to minimize redundant data movement.

The contributions of this paper are summarized as follows:

- We propose an end-to-end compilation framework from C++ code to AIE intrinsics by integrating Polygeist, MLIR, and MLIR-AIE. The framework is extended to support 2D matrix multiplication, and the complete flow now covers a variety of operators, ranging from 1D element-wise to 2D convolution and matrix multiplication.
- We propose a register reuse optimization based on a ping-pong chaining mechanism, which enhances vectorized

int16 convolutions. When combined with loop unrolling, this method can effectively leverage the AIE 2D-SIMD architecture.

- We perform a performance evaluation and parameter study across multiple convolution sizes, showing that combining loop unrolling with our register reuse optimization yields consistent performance improvements. Compared to the MLIR-AIE workflow, our method reduces latency by up to 1.33× for small-scale conv2D workloads and achieves a consistent 1.08× speedup for medium and large-scale inputs.

II. Background

In this section, we provide a brief overview of the AIE architecture and its intrinsic-based programming model.

A. AMD AIE Architecture

AMD's Versal ACAP integrates three architectures: AI Engine (AIE), programmable logic (PL), and processing system (PS). Our work focuses on the AIE core, a VLIW and SIMD-based processor optimized for high-throughput computing tasks. The AIE architecture is shown in Figure 1.

The AIE core consists of two main components: the memory subsystem and the function units. The memory subsystem includes 16 KB of instruction memory and 32 KB of data memory, with three access ports (two for loads and one for stores). It also features a register file with 32 scalar registers, 16 256-bit vector registers, and 8 384-bit accumulator registers. The function units consist of a 32-bit RISC scalar core and two SIMD vector units, optimized for fixed-point and floating-point arithmetic, respectively. Fixed-point operations take advantage of 2D SIMD parallelism, while floating-point operations use a conventional SIMD structure. The remainder of this section focuses on the 2D SIMD datapath and the VLIW execution model, both of which are central to the optimizations proposed in this work.

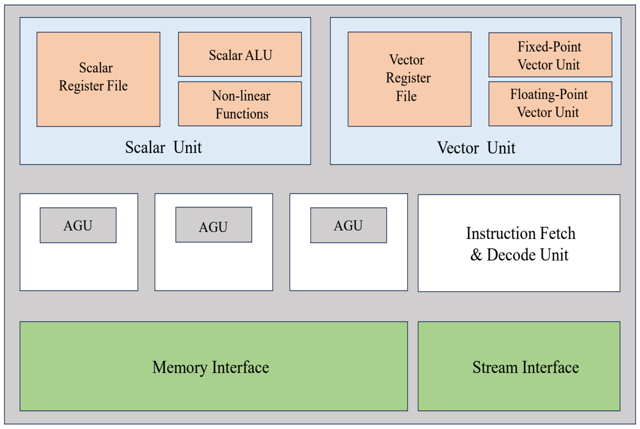


Fig. 1: Architecture of an AIE tile.

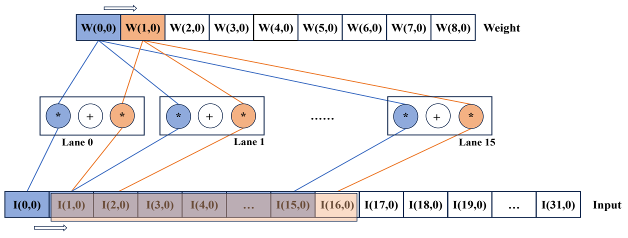


Fig. 2: Lane column abstraction in 2D-SIMD architecture.

1) 2D SIMD Architecture: The AIE features a specialized 2D SIMD architecture for fixed-point arithmetic, comprising two main features: parallel execution units arranged in lanes and columns, and a configurable shuffle network that links the register file to the computation buffer.

The first key feature is the abstraction of lanes and columns, which facilitates parallel computation along two dimensions. Lanes are the parallel execution units, each processing one output vector element, with the number of lanes determining the output vector's width. Columns represent the number of independent multiplication operations within each lane, where results are combined through reductions. This structure enables efficient SIMD operation fusion [9]. The way this lanes columns abstraction works is illustrated in Figure 2. The configuration of lanes and columns depends on the data type. For 32-bit operations, the engine uses 8 lanes and a single column with no internal reduction; for 16-bit operations, either 16 lanes with 2 columns or 8 lanes with 4 columns are utilized [13].

The second key feature is the shuffle network. This programmable interconnect enables fine-grained control over data selection. Through AIE intrinsics, developers can leverage the shuffle network for flexible data routing, enabling efficient computation patterns.

2) VLIW Architecture: The AIE adopts a VLIW architecture, enabling parallel execution across multiple functional units. Each instruction word can encode up to seven operations using variable-length encoding, offering high instruction-level parallelism. To simplify programming and improve scheduling efficiency, the Vitis AI compiler applies automatic software pipelining to innermost loops. With simple loop annotations, developers can exploit parallelism without manually managing low-level instruction sequencing [13].

B. AIE Intrinsic programming model

AMD AIE intrinsics are low-level constructs that provide direct control over the AIE's vector processing units, memory units, and vector registers. Although powerful, this low-level model requires a deep understanding of the hardware, making development complex and error-prone. Table I lists common AIE intrinsics.

III. Design Methodology

In this section, we first provide an overview of the end-to-end tool flow. We then explain the proposed ping-pong

TABLE I: Common AIE Intrinsics

Intrinsic	Description
`upd_v`	Updates a 128-bit segment of a vector with 128-bit data.
`upd_w`	Updates a 256-bit segment of a vector with 256-bit data.
`mul16`	Performs 16-bit multiply operations on vectorized data.
`mac16`	Performs MAC operations on 16-bit vector registers.
`lmac8`	Performs MAC operations on 8-bit vector registers.
`fpmul`	Single precision real times real floating point vectors.
`concat`	Combines two vectors into a single larger vector.
`ups`	Convert integer into a 48-bit accumulator.
`lups`	Convert integer into a 80-bit accumulator.
`srs`	Shifts and rounds the results of vector operations.

chaining mechanism to improve convolution performance. Finally, we present our extension to matrix multiplication vectorization, which includes a memory-coalescing optimization.

A. Overview

We propose an end-to-end tool flow for generating single-core operators from high-level C++ code. The workflow is organized into three main stages: front-end integration, middle-end optimization, and code generation, as illustrated in Figure 3.

In the front-end stage, we utilize Polygeist [14] to lower C++ programs into the MLIR affine dialect, producing a loop-structured intermediate representation (IR). This integration ensures seamless interoperability with the MLIR ecosystem [15] and allows for leveraging its existing optimization passes and dialect extension mechanisms.

The middle stage employs a two-stage vectorization process. In the hardware-agnostic phase, affine loops are transformed into virtual vector abstractions through the `mlir-opt` tool. This phase introduces vectorization that is independent of any hardware, allowing general optimizations such as loop unrolling and fusion [16]. In the hardware-specific phase, these virtual vectors are lowered to the AIE-specific `aievec` dialect. This phase focuses on mapping the virtual vector operations to the target hardware architecture by applying optimizations such as 2D SIMD instruction fusion, data path configuration, and register allocation. Our pattern-specific optimization pipelines are also applied during this stage.

Finally, the code generation stage converts the `aievec` IR into C++ code incorporating AIE intrinsics through a direct, one-to-one mapping.

B. Ping-pong Chaining Mechanism

In this section, we first outline the principles of vectorized convolution. Next, we describe how vectorized convolution is expressed using AIE intrinsics, emphasizing the mapping between high-level operations and low-level hardware instructions. We then identify optimization patterns and present our ping-pong chaining mechanism, discussing its effectiveness. Finally, we explain how this optimization is implemented as a compiler pass within our tool flow.

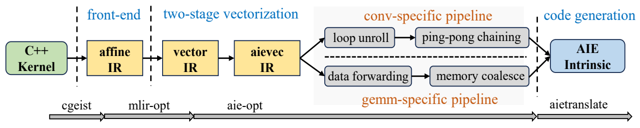


Fig. 3: End-to-end single kernel generator tool flow.

1) Principles of Convolution Vectorization: Convolution is a fundamental operation in image processing and deep learning [17]–[19], widely used for feature extraction. Given an input image $I \in \mathbb{R}^{H\times W}$ and a filter $K \in \mathbb{R}^{m\times n}$, the 2D convolution produces an output $O \in \mathbb{R}^{(H-m+1)\times(W-n+1)}$, where each output element is computed as:

$$O(i,j) = \sum_{u=0}^{m-1}\sum_{v=0}^{n-1} I(i+u, j+v)\cdot K(u,v) \qquad (1)$$

To enable parallel execution, the vectorized convolution kernel adopts a sliding window strategy. This approach structures the input data into horizontal regions (windows), allowing multiple output elements to be computed simultaneously using SIMD instructions. Fig. 4 illustrates the vectorized convolution kernel computation, where each color represents a sliding region.

2) AIEVEC-Based Realization of Vectorized Convolution: Building on the sliding-window-based vectorization strategy, our tool flow lowers high-level convolutions into AIE-specific vector instructions using the `aievec` dialect, exposing hardware intrinsics such as vector loads, multiply-accumulate (MAC) operations, and fixed-point scaling. Fig. 5 illustrates the generated `aievec` IR for a 3×3 convolution on a 32×32 output tile with `int16` precision, which clearly shows four steps.

Step1: Data Loading. The entire filter is loaded into a vector register with a single `aievec.upd` instruction, enabling weight reuse across the tile by broadcasting during computation. For input features, a block of 32 `int16` elements is loaded into a `v32int16` register, providing the necessary data to compute a single `v16int16` output vector.

Step2: Data Path Configuration. The `xoffset` and `zoffset` attributes in the `mac` and `mul` instructions control the data flow from registers to function units. This mechanism is key to the register-level optimizations presented later.

Step3: Elementwise Computation. The `mul` and `mac` instructions perform elementwise multiplication and accumu-

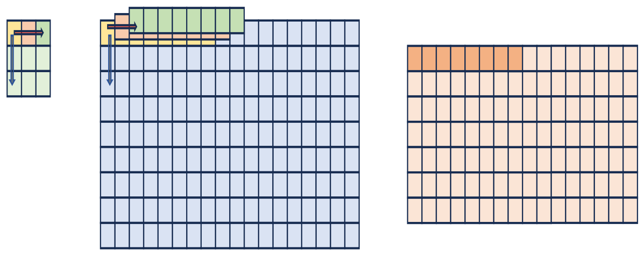
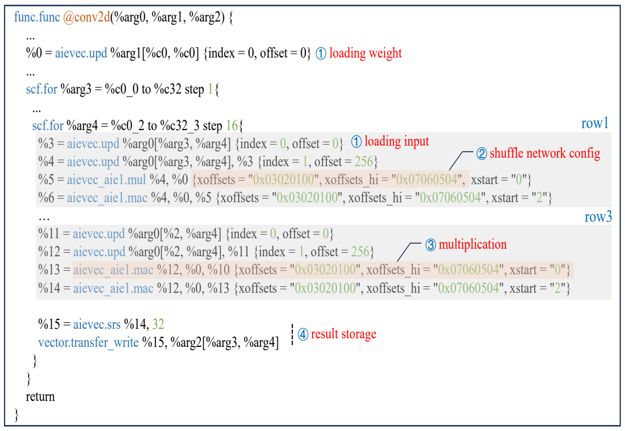


Fig. 4: Sliding window pattern in convolution kernel computation.



```
func.func @conv2d(%arg0, %arg1, %arg2) {
  ...
  %0 = aievec.upd %arg1[%c0, %c0] {index = 0, offset = 0}
  ...
  scf.for %arg3 = %c0_0 to %c32 step 1{
   ...
   scf.for %arg4 = %c0_2 to %c32_3 step 16{
    %3 = aievec.upd %arg0[%arg3, %arg4] {index = 0, offset = 0}
    %4 = aievec.upd %arg0[%arg3, %arg4], %3 {index = 1, offset = 256}
    %5 = aievec_aie1.mul %4, %0 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504",  xstart = "0"}
    %6 = aievec_aie1.mac %4, %0, %5 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504", xstart = "2"}
    …
    %11 = aievec.upd %arg0[%2, %arg4] {index = 0, offset = 0}
    %12 = aievec.upd %arg0[%2, %arg4], %11 {index = 1, offset = 256}
    %13 = aievec_aie1.mac %12, %0, %10 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504", xstart = "0"}
    %14 = aievec_aie1.mac %12, %0, %13 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504", xstart = "2"}

    %15 = aievec.srs %14, 32
    vector.transfer_write %15, %arg2[%arg3, %arg4]
   }
  }
  return
}
```

Fig. 5: aievec IR

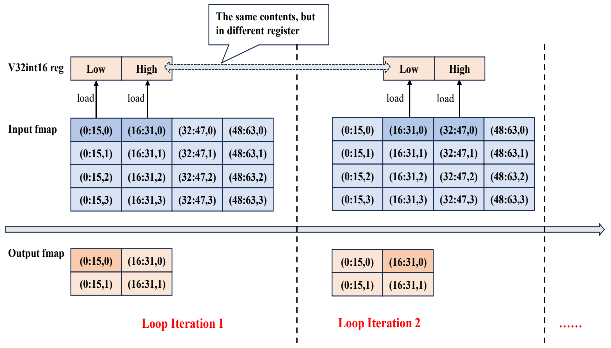


Fig. 6: Register reuse pattern across loop iteration.

lation. Using AIE's 2D SIMD architecture, two overlapping instructions can be bundled into a single MAC operation.

Step4: Result Storage. The results are stored in the `v48int16` accumulator register. The `srs` operation converts this into the final `v16int16` register, which is then written back to memory.

3) Ping-Pong Chaining Mechanism: Analyzing the generated `aievec` IR reveals a clear opportunity for register reuse across adjacent loop iterations. Specifically, in the innermost loop, the output vector advances by 16 elements along the x-axis each iteration, causing the input tile to shift accordingly—from (0:31) to (16:47). This implies that the two `aievec.upd` instructions constructing a `vector<32xi16>` in the next iteration can partially reuse register contents from the previous iteration. Fig. 6 shows this potential reuse opportunity.

To exploit this reuse opportunity, our mechanism leverages two key features of the AIE architecture.

Feature1: Vector Register Construction. The AIE's smallest register is the `v16int16`, and the `v32int16` register is constructed by concatenating two `v16int16` registers. This allows independent access to the lower and upper halves of the `v32int16` register. Specifically, in consecutive iterations, we can retain the upper half of the `v32int16` register and only load the new data segment (elements 32:47) into the lower half to form the input vector for the next iteration.

Feature2: Data Selection in Shuffle Network. Since the new iteration places elements 16:31 in the high half and 32:47 in the low half of the register, the shuffle network that routes data from the register file to the MAC/MUL computation units must be reconfigured to maintain correct data alignment. To address this, the compiler swaps the `xoffsets` and `xoffsets_hi` parameters in the `mac` and `mul` intrinsics, effectively adjusting the data selection within the shuffle network and ensuring the computation uses the correct input segments.

Through theoretical analysis, in the current IR, each iteration consists of three blocks, with each block containing two `upd` operations and two `mac/mul` operations, corresponding to the three rows of the 3×3 filter kernel. By applying this optimization, we can eliminate one `upd` operation and one register allocation per block, resulting in a potential saving of up to three `upd` operations and three registers over two iterations.

The previous analysis focuses solely on potential optimizations between two adjacent loop iterations. Our register reuse mechanism can propagate like a chain across multiple iterations, thus further expanding the optimization space. We also need to continually flip-flop the `xoffsets` attribute for shuffle network configuration. These two key aspects are exactly why we refer to our mechanism as the ”ping-pong chaining mechanism.” Fig. 7 illustrates the complete chained register reuse pattern, which represents the final form of our optimization pass.

Before delving into the implementation of our optimization pass, it's important to acknowledge the gap between theory and practice. A larger unroll factor does not always lead to better optimization, especially in a VLIW architecture. Excessive unrolling increases register pressure and issue slot conflicts, which reduces instruction bundling efficiency and limits optimization effectiveness. Thus, there is a trade-off between unroll depth and instruction scheduling performance. The impact of different unroll factors is evaluated in the Experiments section.

4) Implementation of the Register Reuse Optimization: Our register reuse optimization is realized through two complementary transformation passes: a loop unrolling pass and a ping-pong register reuse pass. The loop unrolling pass expands the inner loop to reveal more opportunities for register reuse across consecutive iterations, enabling a chained reuse of vector registers. Building on this, the second pass performs targeted pattern analysis and transformations to realize the reuse strategy.

The core optimization contains two main stages. The first stage is an interval analysis, where clusters of `upd` operations accessing overlapping memory regions are identified. Algorithm. 1 outlines this process. This is achieved by computing the flattened linear access ranges of each `UPDOp` via affine

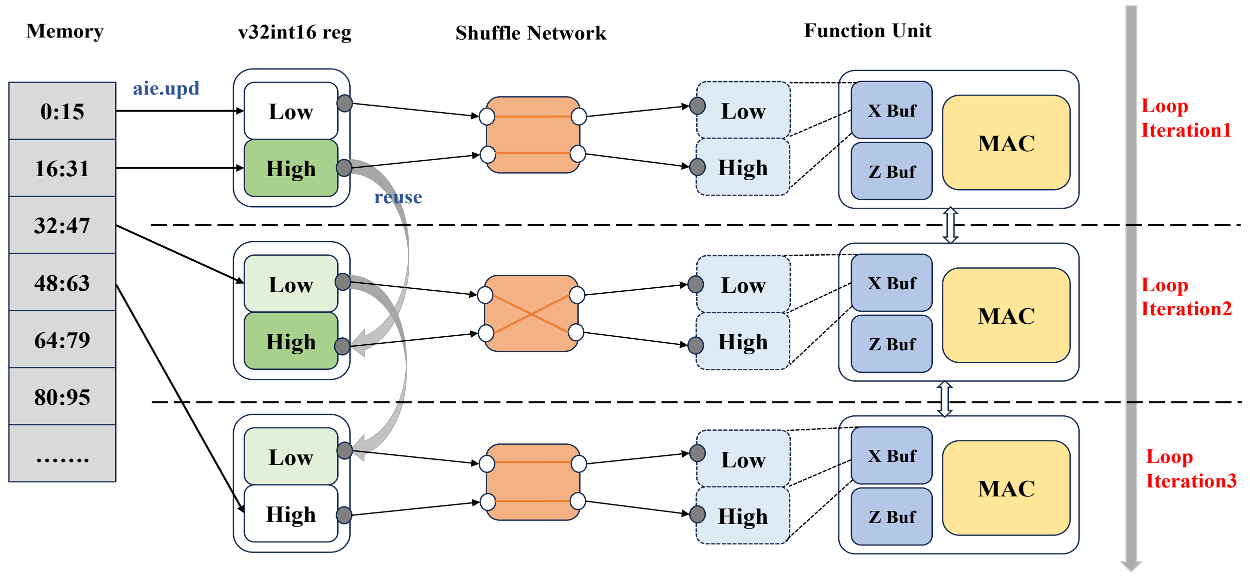


Fig. 7: Ping-pong chaining mechanism.

expressions on the loop indices and offset attributes. Any pair of `UPDOps` with matching base memrefs and overlapping access intervals is grouped into an *interval*, forming candidates for reuse.

Algorithm 1 Interval Analysis for `UPDOp` Clustering

```
1:  Input: List of UPDOps from aievec IR
2:  Output: Interval chains of overlapping UPDOps
3:  function INTERVAL_ANALYSIS(UPDOpList)
4:      Initialize hash table, IntervalMap ← ∅
5:      for all upd ∈ UPDOpList do
6:          Extract memory base and affine expr
7:          offset ← $\sum_{i=1}^{N}$ index$_i$ × stride$_i$ + offset
8:          key ← (base, addr)
9:          Insert upd into IntervalMap[key]
10:     end for
11:     return IntervalMap
12: end function
```

In the second stage, transformations are applied to each interval by selecting the first `UPDOp` as the canonical load and reusing its vector register for subsequent operations. Redundant `UPDOps` are removed and their consumers—usually `mul` or `mac`—are updated to reference the canonical register with correct shuffle network configuration as needed. This transformation is subsequently propagated to other intervals to maintain consistency along the update chains.

C. Extension for Vectorized Matrix Multiplication

To cover a wider range of operators, we also extended MLIR-AIE to support vectorization for matrix multiplication. Our approach follows a common pattern [17], where vectorization is applied along the j-dimension, as illustrated in Fig. 8.

Upon analyzing this matrix multiplication pattern, we observed that the scalar elements of the matrix A accessed in the innermost loop are contiguous in memory along the k-dimension. This observation motivates an optimization to coalesce multiple scalar accesses into a single vector register load. By utilizing the AIE's shuffle network, we can broadcast the scalar values stored in the vector register during each loop iteration. In the case where the vectorization factor is 8, this optimization effectively transforms 8 scalar accesses

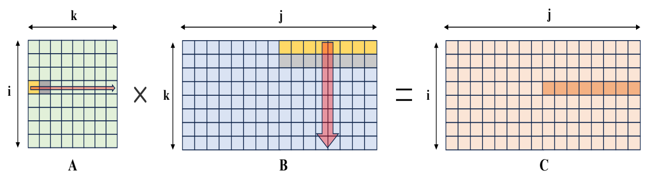


Fig. 8: Vectorization in matrix multiplication workload.

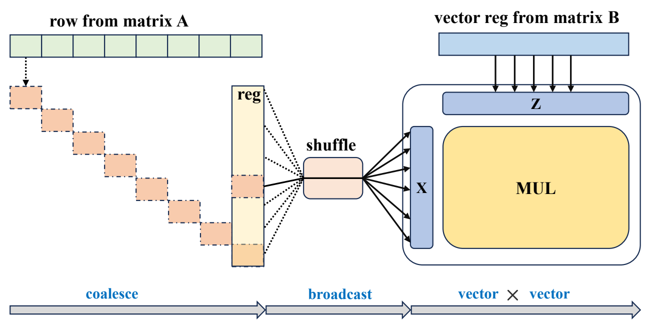


Fig. 9: Memory coalescing optimization for vectorized matrix multiplication.

into a single vector register access, thus improving memory efficiency and reducing access overhead. Fig. 9 illustrates this optimization.

IV. Experiments

In the experiment section, we evaluate the performance gains from our register-level optimizations. To guide the selection of an optimal unroll factor, we also conduct a detailed study analyzing its impact on performance. Finally, we walk through a simple yet representative convolution task to demonstrate the ease of use and the optimization process of our approach.

A. Experimental Setup

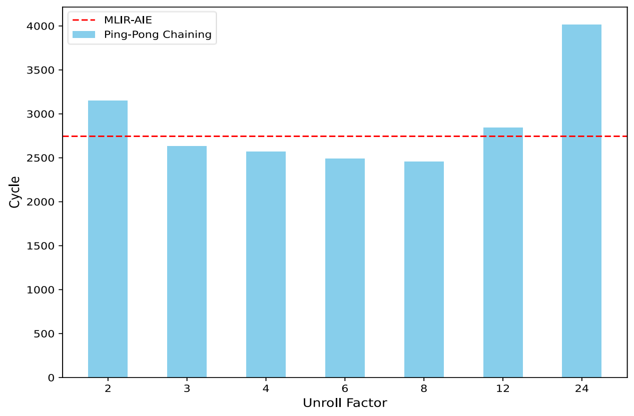
All experiments are conducted on the Versal ACAP platform using the VCK190 toolkit, with AIEs operating at 1.0GHz in the Vitis 2021.1 environment. The Vitis AIE toolchain (version 2021.1) is used for kernel compilation and simulation. Specifically, the `xchessmk` tool compiles single-core kernels with customized testbenches, and the `xca_udm_dbg` simulator performs cycle-accurate simulation and performance profiling. Profiling data are collected in the `prf` files for further analysis.

B. Evaluation of Performance

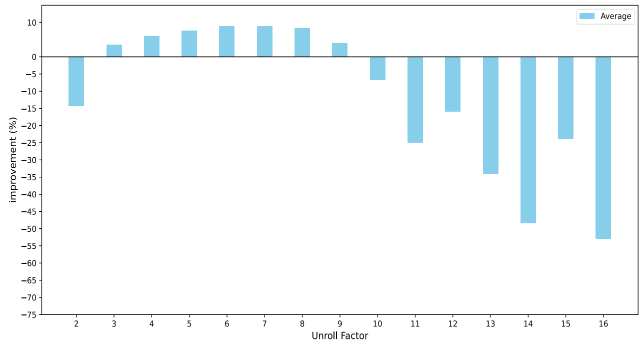
Table II compares the performance of our tool flow with the MLIR-AIE baseline. Our method integrates the proposed ping-pong chaining mechanism with loop unrolling, while the baseline applies basic vectorization and simple register optimizations. The unroll factor is selected based on three observations: (1) The unroll factor is determined by the column size of the convolution output. For example, with a column size of 64 and 16-vectorization, the loop range is 4, thus the unroll factor is typically constrained to its divisors, such as 2 or 4; (2) For small convolutions, unrolling has limited overhead due to low register pressure and ease of scheduling; (3) For medium to large convolutions, the performance is more sensitive to unrolling, with optimal values typically falling between 3 and 9.

TABLE II: Performance Evaluation

Scale	Output Size	MLIR-AIE Baseline	Our Method		Speedup
			Unroll Factor	Register Reuse	
small	16×16	428	2	234	1.45×
	32×32	828	2	442	1.46×
	64×32	1628	2	858	1.53×
	16×64	505	4	426	1.16×
	32×64	985	4	826	1.16×
	64×64	1943	4	1626	1.17×
medium	16×160	1177	5	1098	1.07×
	16×192	1403	6	1274	1.09×
	16×224	1625	7	1482	1.09×
	16×256	1849	4	1738	1.06×
	16×288	2073	6	1882	1.09×
	16×320	2297	5	2122	1.08×
large	16×448	3193	7	2890	1.09×
	16×480	3419	6	3100	1.09×
	16×512	3643	8	3260	1.11×



(a) Unroll Factor Impact on 16x384 convolution size.



(b) Average Unroll Factor Impact.

Fig. 10: Unroll Factor's impact on performance

Our approach shows improvements across most convolution sizes. For smaller convolutions, such as 16×32 and 16×64, we observe average cycle reductions of 46% and 17%, respectively. However, for medium and large convolutions, improvements are more modest, averaging around 8%.

This behavior is closely tied to the size of the output matrix. We observe that the number of *rows* correlates linearly with execution time, while the number of *columns* determines how effectively our ping-pong chaining mechanism can reduce latency, as it operates along the column dimension. For smaller column sizes (e.g. 32 or 64), register-level reuse across iterations is highly effective, leading to significant performance gains. In contrast, for larger convolutions (e.g., 16×256), excessive unrolling results in larger inner-loop bodies, which hinder VLIW instruction optimization and increase register pressure. This behavior can be confirmed in the assembly codes, where we observe more register spills and additional nop insertion, leading to performance degradation compared to smaller convolutions.

Thus, the appropriate selection of unroll factor is crucial, particularly for large-scale convolutions.

C. Parameter Study

To demonstrate how the unroll factor affects performance, we use a 16×384 convolution output as our example case. Fig. 10a shows the performance impact of various unroll factors. The ping-pong chaining optimization is effective from an unroll factor from 4 to 8, with continuous performance improvement. However, an unroll factor of 2 results in poor performance due to the lack of register reuse chaining opportunity. At 24, performance drops sharply, as assembly analysis reveals increased register pressure and reduced VLIW optimization effectiveness.

To confirm our observation, we analyze 16 convolution sizes, ranging from column size 32 to 512, with unroll factors from 2 to 16. Fig. 10b shows that unroll factors between 3 and 9 improve performance. Based on this insight, our tool provides both manual and automatic unroll factor selection options, enabling developers to choose the optimal factor for their specific task.

D. Case Study

To illustrate the overall flow of our work, we walk through a commonly used convolution task with an output size of 16×256 and a 3×3 filter kernel, specifically targeting an int16 computation. Fig. 11 shows how high-level code is transformed into AIE Intrinsic code step by step, with AIE hardware optimizations applied.

1) Frontend: To generate correct AIE Intrinsic code, the user needs to provide two C++ files: one for kernel description and another for testing. The kernel file defines a simple C++ function. The test file prepares test data (with manual alignment for int16), launches the kernel, and verifies the results. The first step of our tool flow is to integrate with the MLIR ecosystem. We leverage cgeist tool from Polygeist project, which automatically generates the MLIR affine dialect. This results in a nearly one-to-one mapping between the C++ input and the generated affine IR(Fig. 11①).

2) Two Stage Vectorization: The core of our middle-end optimization lies in a two-stage vectorization process, which enables the transformation from high-level affine IR to AIE-specific vectorized code.

The first stage, hardware-agnostic vectorization, applies the `supervectorize` pass from upstream MLIR to lift scalar operations into 16-wide vector operations, as shown in Fig. 11②. In this stage, loop steps are widened, memory access is replaced with `vector.transfer` to load vectorized data, and scalar operands are promoted to vector types(Fig. 11ⓐⓑⓒ). This introduces a virtual vector abstraction, flexible in shape and dimension but not yet aligned with the hardware constraints of AIE, motivating a second stage to lower these generic vector operations into AIE-specific intrinsics.

The second stage, hardware-specific vectorization, uses the MLIR-AIE vectorization pass to lower the generic vector dialect into the AIE-specific `aievec` dialect(Fig. 11③). This includes mapping vector loads to the `aievec.upd` operation(Fig. 11ⓓ), which encodes the register access pattern using `index` and `offset` attributes. Vector multiplication is lowered to `aievec.mul`, with hardware-specific attributes such as `xoffsets` to configure data selection for the MAC unit(Fig. 11ⓔ). The `srs` instruction is also inserted to adjust the bitwidths of the accumulator(Fig. 11ⓕ).

3) Pattern-specific Passes: To further enhance performance, we provide two pattern-specific optimization pipelines: ping-pong chaining for convolution workloads and memory coalescing for matrix multiplication. In this case study, we apply ping-pong chaining to optimize the sliding window pattern. Users can enable this optimization via the `--aie-register-reuse` option and control the unrolling factor with the `--unroll-factor`. We choose `--unroll-factor` = 4 to better show our optimization effects.

Fig. 11④ illustrates the transformation process, which consists of two key phases: loop unrolling and register reuse via ping-pong chaining. Our focus is the latter. First, the `aievec.upd` instructions reveal that each `v32int16` register is loaded in two halves; our reuse mechanism enables one half (high or low) to be retained across iterations, reducing redundant memory accesses(Fig. 11ⓖ). Second, due to the alternate fill direction of the registers, a shuffle network is leveraged every two iterations to flip the register layout, ensuring correctness in `mac` or `mul` operations. This behavior is reflected in the alternating `xoffset_hi` and `xoffset` attributes across instruction groups(Fig. 11ⓗ). Together with a sufficiently large unroll factor(Fig. 11ⓘ), this creates a chained reuse pattern across iterations.

4) Code Generation: Code generation is quite simple, as aievec ir is already very close to AIE Intrinsics and a direct one-to-one mapping works.

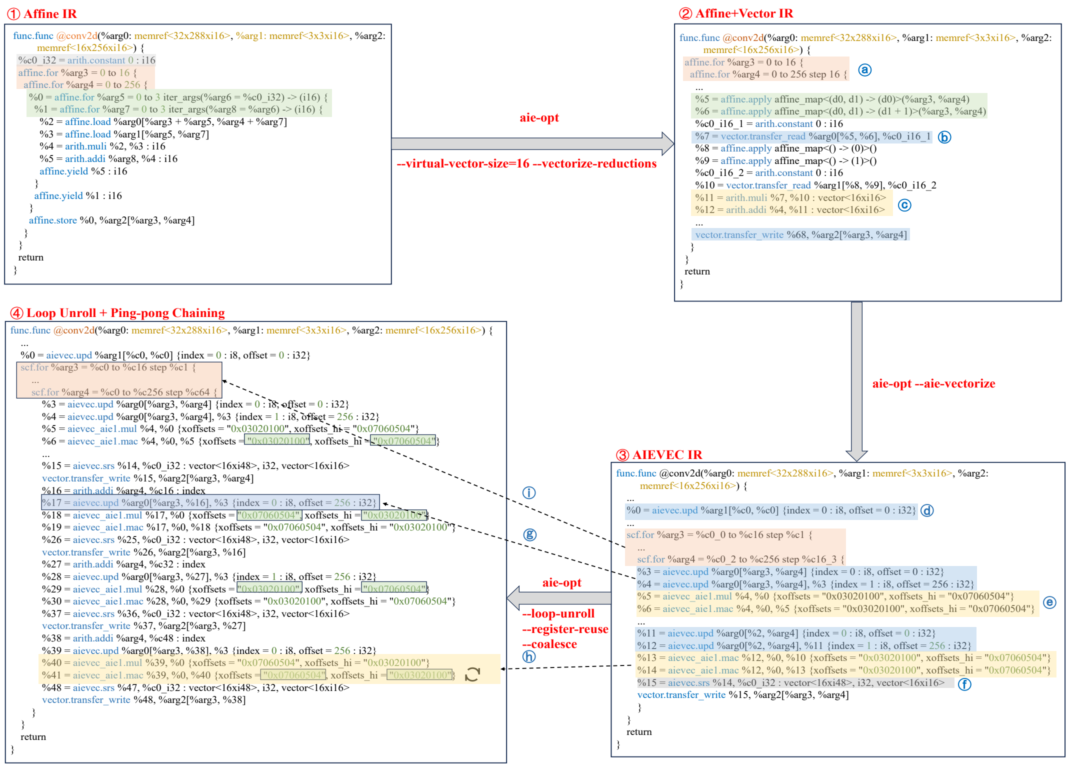
Through these four stages, the optimized AIE intrinsic code is generated. The entire transformation pipeline is transparent, requiring the developer only to provide a C++ implementation and select an appropriate unroll factor.

V. Conclusion

In this work, we propose an MLIR-based tool flow that automatically generates optimized AIE intrinsic code for the AI Engine from C++ input. The tool flow supports operators such as 1D element-wise operations, 2D convolutions, and matrix multiplication. We introduce a ping-pong chaining mechanism for the sliding-window pattern in convolution tasks, reducing memory-register interaction and improving register reuse. Our approach achieves up to 1.33× speedup for small-scale convolutions, 1.08× for medium and large-scale workloads compared to the MLIR-AIE baseline workflow.

Acknowledgment

This work is supported by the National Key R&D Program of China (2022YFB2901100), the National Natural Science Foundation of China (No. 62404257), and the Guangdong Basic and Applied Basic Research Foundation (2023A1515110769).



```
func.func @conv2d(%arg0: memref<32x288xi16>, %arg1: memref<3x3xi16>, %arg2:
        memref<16x256xi16>) {
  %c0_i32 = arith.constant 0 : i16
  affine.for %arg3 = 0 to 16 {
   affine.for %arg4 = 0 to 256 {
    %0 = affine.for %arg5 = 0 to 3 iter_args(%arg6 = %c0_i32) -> (i16) {
     %1 = affine.for %arg7 = 0 to 3 iter_args(%arg8 = %arg6) -> (i16) {
       %2 = affine.load %arg0[%arg3 + %arg5, %arg4 + %arg7]
       %3 = affine.load %arg1[%arg5, %arg7]
       %4 = arith.muli %2, %3 : i16
       %5 = arith.addi %arg8, %4 : i16
       affine.yield %5 : i16
      }
     affine.yield %1 : i16
    }
    affine.store %0, %arg2[%arg3, %arg4]
   }
  }
  return
}
```

aie-opt
--virtual-vector-size=16 --vectorize-reductions

```
func.func @conv2d(%arg0: memref<32x288xi16>, %arg1: memref<3x3xi16>, %arg2:
        memref<16x256xi16>) {
  affine.for %arg3 = 0 to 16 {
   affine.for %arg4 = 0 to 256 step 16 {
     ...
     %5 = affine.apply affine_map<(d0, d1) -> (d0)>(%arg3, %arg4)
     %6 = affine.apply affine_map<(d0, d1) -> (d1 + 1)>(%arg3, %arg4)
     %c0_i16_1 = arith.constant 0 : i16
     %7 = vector.transfer_read %arg0[%5, %6], %c0_i16_1
     %8 = affine.apply affine_map<() -> (0)>()
     %9 = affine.apply affine_map<() -> (1)>()
     %c0_i16_2 = arith.constant 0 : i16
     %10 = vector.transfer_read %arg1[%8, %9], %c0_i16_2
     %11 = arith.muli %7, %10 : vector<16xi16>
     %12 = arith.addi %4, %11 : vector<16xi16>
     ...
     vector.transfer_write %68, %arg2[%arg3, %arg4]
   }
  }
  return
}
```

aie-opt --aie-vectorize

```
func.func @conv2d(%arg0: memref<32x288xi16>, %arg1: memref<3x3xi16>, %arg2:
        memref<16x256xi16>) {
  ...
  %0 = aievec.upd %arg1[%c0, %c0] {index = 0 : i8, offset = 0 : i32}
  ...
  scf.for %arg3 = %c0_0 to %c16 step %c1 {
    ...
    scf.for %arg4 = %c0_2 to %c256 step %c16_3 {
    %3 = aievec.upd %arg0[%arg3, %arg4] {index = 0 : i8, offset = 0 : i32}
    %4 = aievec.upd %arg0[%arg3, %arg4], %3 {index = 1 : i8, offset = 256 : i32}
    %5 = aievec_aie1.mul %4, %0 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
    %6 = aievec_aie1.mac %4, %0, %5 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
    ...
    %11 = aievec.upd %arg0[%2, %arg4] {index = 0 : i8, offset = 0 : i32}
    %12 = aievec.upd %arg0[%2, %arg4], %11 {index = 1 : i8, offset = 256 : i32}
    %13 = aievec_aie1.mac %12, %0, %10 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
    %14 = aievec_aie1.mac %12, %0, %13 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
    %15 = aievec.srs %14, %c0_i32 : vector<16xi48>, i32, vector<16xi16>
    vector.transfer_write %15, %arg2[%arg3, %arg4]
    }
  }
  return
}
```

aie-opt
--loop-unroll
--register-reuse
--coalesce

```
func.func @conv2d(%arg0: memref<32x288xi16>, %arg1: memref<3x3xi16>, %arg2: memref<16x256xi16>) {
  ...
  %0 = aievec.upd %arg1[%c0, %c0] {index = 0 : i8, offset = 0 : i32}
  scf.for %arg3 = %c0 to %c16 step %c1 {
    ...
    scf.for %arg4 = %c0 to %c256 step %c64 {
      %3 = aievec.upd %arg0[%arg3, %arg4] {index = 0 : i8, offset = 0 : i32}
      %4 = aievec.upd %arg0[%arg3, %arg4], %3 {index = 1 : i8, offset = 256 : i32}
      %5 = aievec_aie1.mul %4, %0 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
      %6 = aievec_aie1.mac %4, %0, %5 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
      ...
      %15 = aievec.srs %14, %c0_i32 : vector<16xi48>, i32, vector<16xi16>
      vector.transfer_write %15, %arg2[%arg3, %arg4]
      %16 = arith.addi %arg4, %c16 : index
      %17 = aievec.upd %arg0[%arg3, %16], %3 {index = 0 : i8, offset = 256 : i32}
      %18 = aievec_aie1.mul %17, %0 {xoffsets = "0x07060504", xoffsets_hi = "0x03020100"}
      %19 = aievec_aie1.mac %17, %0, %18 {xoffsets = "0x07060504", xoffsets_hi = "0x03020100"}
      %26 = aievec.srs %25, %c0_i32 : vector<16xi48>, i32, vector<16xi16>
      vector.transfer_write %26, %arg2[%arg3, %16]
      %27 = arith.addi %arg4, %c32 : index
      %28 = aievec.upd %arg0[%arg3, %27], %3 {index = 1 : i8, offset = 256 : i32}
      %29 = aievec_aie1.mul %28, %0 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
      %30 = aievec_aie1.mac %28, %0, %29 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504"}
      %37 = aievec.srs %36, %c0_i32 : vector<16xi48>, i32, vector<16xi16>
      vector.transfer_write %37, %arg2[%arg3, %27]
      %38 = arith.addi %arg4, %c48 : index
      %39 = aievec.upd %arg0[%arg3, %38], %3 {index = 0 : i8, offset = 256 : i32}
      %40 = aievec_aie1.mul %39, %0 {xoffsets = "0x07060504", xoffsets_hi = "0x03020100"}
      %41 = aievec_aie1.mac %39, %0, %40 {xoffsets = "0x07060504", xoffsets_hi = "0x03020100"}
      %48 = aievec.srs %47, %c0_i32 : vector<16xi48>, i32, vector<16xi16>
      vector.transfer_write %48, %arg2[%arg3, %38]
    }
  }
  return
}
```

Fig. 11: Step by Step code transformation

References

[1] Y. Annepaka and P. Pakray, "Large language models: a survey of their development, capabilities, and applications," Knowledge and Information Systems, vol. 67, no. 3, pp. 2967–3022, Mar. 2025.

[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," Commun. ACM, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[3] S. Kang, H. J. Choi, C. H. Kim, S. W. Chung, D. Kwon, and J. C. Na. 2011. Exploration of CPU/GPU co-execution: From the perspective of performance, energy, and temperature. In Proceedings of the 2011 ACM Symposium on Research in Applied Computation, RACS '11, pp. 38–43.

[4] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno. May 2005. CODESSEAL: Compiler/FPGA approach to secure applications. In Proceedings of the IEEE International Conference on Intelligence and Security Informatics, pp. 530–535.

[5] N. P. Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," Apr. 16, 2017, arXiv: arXiv:1704.04760.

[6] D. Abts et al., "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads, " in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain: IEEE, May 2020, pp. 145–158.

[7] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: VersalTM architecture," in Proceedings of the 2019 ACM/SIGDA International Symposium on FieldProgrammable Gate Arrays, 2019, pp. 84–93.

[8] Versal: The First Adaptive Compute Acceleration Platform (ACAP), Xilinx, 9 2019, v1.0.1.

[9] Y. Ben-Asher, D. Egozi, and A. Schuster, "2-D SIMD algorithms in the perfect shuffle networks," SIGARCH Comput. Archit. News, vol. 17, no. 3, pp. 88–95, Jun. 1989.

[10] J. A. Fisher, "Very Long Instruction Word architectures and the ELI-512," in Proceedings of the 10th annual international symposium on Computer architecture - ISCA '83, Stockholm, Sweden: ACM Press, 1983, pp. 140–150.

[11] P. Chatarasi, S. Neuendorffer, S. Bayliss, K. Vissers, and V. Sarkar, "Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AIE," in 2020 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA: IEEE, Sep. 2020, pp. 1–10.

[12] AMD. MLIR-AIE: An MLIR-based AIE toolchain. https://xilinx.github.io/mlir-aie/. Accessed: 2024-09-15.

[13] AMD. AIE Kernel and Graph Programming Guide, UG1079, 2023.

[14] W. S. Moses, "Polygeist: Affine C in MLIR," 2021.

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

[16] U. Bondhugula, "High Performance Code Generation in MLIR: An Early Case Study with GEMM," Mar. 01, 2020, arXiv: arXiv:2003.00532. Accessed: Nov. 04, 2024. [Online]. Available: http://arxiv.org/abs/2003.00532

[17] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," ACM Transactions on Architecture and Code Optimization (TACO), vol. 14, no. 3, pp. 1–25, 2017.

[18] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein et al., "Imagenet large scale visual recognition challenge," International Journal of Computer Vision, vol. 115, no. 3, pp. 211–252, 2015.

[19] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in Conference on Computer Vision and Pattern Recognition (CVPR), 2015.