Received 13 February 2024, accepted 24 April 2024, date of publication 6 May 2024, date of current version 13 May 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3397195*

RESEARCH ARTICLE

# Case Study: Optimization Methods With TVM Hybrid-OP on RISC-V Packed SIMD

**MENG-SHIUN YU, CHUAN-YUE YUAN, TAI-LIANG CHEN, AND JENQ-KUEN LEE**
Department of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan

Corresponding author: Jenq-Kuen Lee (jklee@cs.nthu.edu.tw)

This work was supported in part by Google, and in part by the National Science and Technology Council (NSTC) of Taiwan under Grant 111-2221-E-007-080-MY3.

**ABSTRACT** In recent years, considerable research has focused on the use of custom hardware to accelerate deep learning on edge devices. However, the end-to-end flow of deep learning includes preprocessing and postprocessing. Deep learning hardware accelerators cannot accelerate these operations, which consequently becomes a performance bottleneck in the execution flow. In this study, we propose optimization methods to improve preprocessing and postprocessing at the edge devices. For this purpose, we adopt Tensor Virtual Machine (TVM), an end-to-end machine learning compiler framework. TVM provides hybrid script, which is a front-end language that allows users to write programs for preprocessing and postprocessing. We propose rewriting strategies to improve the performance of operators written in hybrid script through the RISC-V Packed SIMD extension (P extension). RISC-V is an open instruction set architecture (ISA) that provides base instructions and many extensions for different use cases. The P extension defines specific subword single-instruction multiple-data (SIMD) instructions that allow complex computations to be efficiently performed on edge devices. In this study, we design custom instructions based on the RISC-V P extension for rewriting strategies to accelerate deep learning operations. Experimental results indicate that our methods improve performance by a factor of 1.28 to 15.29.

**INDEX TERMS** TVM, machine learning compiler, RISC-V, custom instruction.

## I. INTRODUCTION

Deep learning has achieved remarkable successes across various fields, particularly in tasks such as object detection in computer vision [1], [2], keyword spotting in speech [3], [4], natural language processing [5], [6] and industrial defect inspection [7], [8], [9]. However, as these models find applications in real-world scenarios, such as on mobile and edge devices, a new set of challenges is emerging. In these contexts, the constraints on computing resources become pronounced, presenting challenges for the effective execution of deep learning algorithms. For instance, consider the domain of real-time object detection on edge devices, in which stringent latency requirements are imposed. Current approaches, such as MobileNet and SqueezeNet, have made progress in reducing computational demands. Nevertheless, the end-to-end execution flow, which includes preprocessing and postprocessing, remains a bottleneck. Preprocessing and postprocessing operations often cannot be accelerated by standard deep learning accelerators, making optimization in these stages critical. In light of these challenges, our motivation stems from the need to address the limitations of existing solutions and to enhance the overall efficiency of deep learning execution on resource-constrained devices. This study focuses on proposing novel methods to optimize both the preprocessing and postprocessing stages by leveraging the rewriting of Tensor Virtual Machine (TVM) [10] hybrid script and the design of custom instructions based on the RISC-V Packed SIMD extension (P extension).

TVM is an end-to-end deep learning compiler for machine learning models, such as object detection models and decision trees [11]. TVM allows tensor operations to be optimized through a tensor-level intermediate representation (TIR) and provides hybrid script to describe how to complete operational calculations. TVM hybrid script is a programming language based on Python syntax. Many

The associate editor coordinating the review of this manuscript and approving it for publication was Alicia Fornés.

operators use hybrid scripts to complete their operations. The hybrid script provides four loop annotations: serial, unrolled, parallel, and vectorized. In this paper, we rewrite these annotations to improve the performance of the preprocessing and postprocessing stages. RISC-V is a free and open instruction set architecture (ISA) based on a reduced instruction set computer (RISC) architecture with various optional extensions, such as the packed SIMD (P) and vector (V) extensions, that provide SIMD instructions with powerful features for parallel computing for different use cases.

This study takes a comprehensive approach to optimizing nonmaximum suppression(NMS) computation, addressing both software and hardware aspects. In the software domain, we propose rewriting strategies for hybrid scripts to enhance their efficiency. Simultaneously, in the hardware domain, we introduce custom instructions based on the RISC-V P extension, aiming to reduce the instruction count and enable simultaneous computations on multiple data elements. Our methodology is specifically tailored for low-power embedded systems, offering a holistic solution to improve NMS performance. To implement these optimizations, we advocate two key methods: the rewriting strategies for hybrid scripts and the design of custom instructions. Additionally, we enable the LLVM code generator to support RISC-V P extension (RVP) instructions within TVM, ensuring the generation of RVP instructions and reinforcing our commitment to an all-encompassing enhancement of NMS computation performance.

To verify our proposed method, we use NMS as an example and the Microsoft COCO [12] dataset as our test data. The NMS algorithm is commonly used in the postprocessing step for object detection models to eliminate redundant or overlapping bounding boxes. We take the original TVM version of NMS as the baseline for comparison with our optimized version, using the number of instructions and accuracy as indicators. Regarding the number of instructions, we use the RISC-V simulator Spike to accumulate the number of instructions required to complete NMS on a test image, and we calculate the accuracy by comparing the detection results with the ground-truth annotation data provided in the dataset. The experimental results indicate that our approach reduces the number of assembly instructions by a factor of 1.28 to 15.29 while the average precision decreases by 0.002 - 0.03. This research makes the following contributions:

1) We propose rewriting strategies for TVM hybrid script to enhance the performance of pre- and postprocessing operations.
2) We enable LLVM code generation for the RISC-V Packed SIMD extension in TVM, allowing the end-to-end deep learning flow to be executed efficiently on low-power devices.
3) We design custom instructions based on the RISC-V Packed SIMD extension to further enhance the performance of NMS.

The remainder of this paper is organized as follows. In Section II, we present the background TVM and RISC-V. In Section III, we describe how to enable the RISC-V P extension in TVM. In Section IV, we introduce our rewriting strategies, and in Section V, we present custom instructions based on the RISC-V P extension, along with a running example. In Section VI, experimental results are reported. Section VII, we present the previous work and compare it with our research. Finally, we summarize our conclusions in Section VIII.

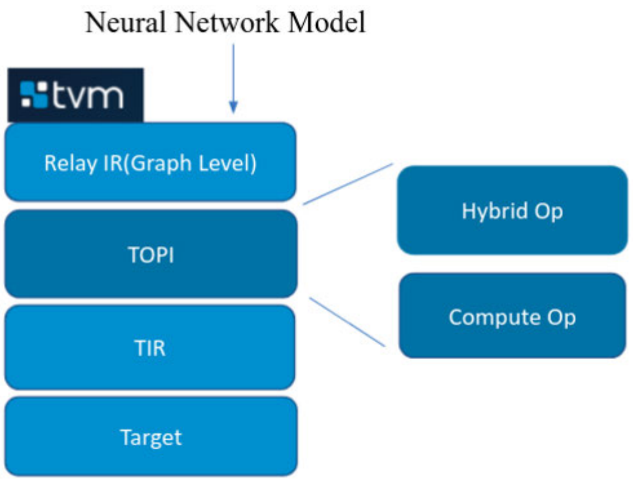


**FIGURE 1.** Overview of TVM software architecture, from the neural network model to the hardware target.

## II. BACKGROUND

### A. TVM AND HYBRID SCRIPT

TVM [10] is an end-to-end machine learning compiler framework for CPUs, GPUs, and accelerators. It is an intermediary platform that can integrate various applications and systems, including blockly applications [13], runtime support for Android NNAPI [14], compiler optimizations across many machine learning computers [15], [16] and underlying integration with a GPU [17], [18]. Fig. 1 illustrates the entire software architecture. The top layer is the Relay intermediate representation (IR), which describes the contents of the directed acyclic graph (DAG) used to define many Ops. There are many optimization methods for DAG. For example, [19] decomposed a task into a decomposition-coordination DAG (DC-DAG) to improve performance further. The optimizations provided by TVM for the DAG of artificial intelligence (AI) models include dead node elimination, constant folding, and operator fusion, among others. The second layer is the TVM Operator Inventory (TOPI), which implements each Op defined in Relay. TOPI describes two types of implementations: one consists of computations for tensor operations, and the other is hybrid script, designed using a subset of the Python language, which can be used to implement algorithms other than tensor operations. The third layer is TIR, which provides a middle-level expression describing low-level programs, including loops, tensor data load/store operations, and built-in intrinsic functions. It is typically used to present how the operator in a model completes their calculation. The fourth layer is the target and includes code generators for different backends, such as LLVM [20] (for CPU), C code, OpenCL, and CUDA.

There are two methods for describing how an Op completes its calculation in TVM. The first one is the tensor expression, which is suitable for tensor operations such as convolution and elementwise addition and subtraction. The other method is hybrid script, which is based on partial Python syntax; this method offers a more flexible programming model than the compute expression method and is suitable for describing preprocessing or postprocessing operations. In addition, hybrid script provides loop annotations (unroll, parallel, vectorize, and bind), loop manipulations (split and fuse), and reordering, allowing developers to schedule the execution order at the source level and write more efficient programs. TVM provides an autotuning module called AutoTVM, which can generate more efficient TIR but does not support the more flexible Python-based hybrid script. Taking vector addition as an example, Listing III-A and Listing 2 use the tensor express and hybrid script, respectively, to describe how to complete the operation of adding two vectors. In line 3 of Listing 1, the te.compute function takes the output shape (dshape) and the lambda function as arguments. The number of arguments of the lambda function depends on the shape of the output tensor. Because the output tensor shape in this example is one-dimensional, the lambda function takes only one argument, denoted by i. The lambda function returns the value to be input into the output tensor by the arguments and the value of the axis. In Listing 2, an output tensor is first created based on the data type and shape of the left-hand side input value in lines 2 and 3. Following the conventions of hybrid script, a loop is used to iterate over the data and add values to the output in lines 6 and 7. Finally, the output tensor is returned. Hybrid script is similar to the C and Python languages, and the TVM computations are based on the output tensor shape for programming.

```
1 def compute_vector_add_1d(lv, rv):
2     dshape = lv.shape
3     return tvm.te.compute(dshape, lambda i: lv[i
      ] + rv[i])
```

**Listing 1.** Tensor expression description of vector addition.

```
1 @hybrid.script
2 def hybrid_vector_add_1d(lv, rv):
3     length = lv.shape[0]
4     output = output_tensor((length,), lv.dtype)
5 
6     for i in range(length):
7         output[i] = lv[i] + rv[i]
8 
9     return output
```

**Listing 2.** Hybrid script description of vector addition.

| Inst[4:2] / Inst[6:5] | 000 | 001 | 010 | 011 |
|---|---|---|---|---|
| 00 | LOAD | LOAD-FP | custom-0 | MISC-MEM |
| 01 | STORE | STORE-FP | custom-1 | AMO |
| 10 | MADD | MSUB | NMSUB | NMADD |
| 11 | BRANCH | JALR | reserved | JAL |

**FIGURE 2.** RISC-V instruction format related to custom instructions.

**TABLE 1.** Examples of SIMD instructions in the RISC-V P extension.

| Instructions | Description |
|---|---|
| KADD8 | SIMD 8-bit signed saturating addition |
| KSUB8 | SIMD 8-bit signed saturating subtraction |
| KHM8 | SIMD 8-bit signed saturating Q7 multiply |
| SMAX8 | SIMD 8-bit signed maximum |
| SMIN8 | SIMD 8-bit signed minimum |
| UMIN8 | SIMD 8-bit unsigned minimum |
| KSLL8 | SIMD 8-bit saturating shift left logical |
| KSLRA8 | SIMD 8-bit shift left logical with saturation or shift right arithmetic |
| SCMPLT8 | SIMD 8-bit signed compare less than |
| CMPEQ8 | SIMD 8-bit integer compare equal |
| SCMPLE8 | SIMD 8-bit signed compare less than equal |

### B. RISC-V AND THE PACKED SIMD EXTENSION

RISC-V is an open ISA based on RISC architecture. The instruction format is illustrated in Fig. 2. Different combinations of opcode fields indicate different functions for each set of instructions. The red words in Fig. 2 indicate fields that RISC-V reserves for developers who want to design custom instructions; we use these two fields in this work to design and implement our custom instructions. RISC-V has open-source licenses with many extensions for different use cases [21], [22], and allows the addition of specific custom instructions, as in [23]. Since 2010, five basic instruction sets (RVWMO, RV32I, RV32E, RV64I, and RV128I) and 21 different feature extensions, including the V extension [24] and the draft cryptography extension [25], have been released. In addition to setting standard specifications, RISC-V International and volunteers from various fields maintain the software tools required for RISC-V development, such as the compiler (LLVM/GNU toolchain) and simulators (Spike and QEMU). In this study, we use these frameworks as the basis for developing custom instructions and performance enhancements.

The RISC-V P extension was contributed to RISC-V International by Andes Technology in 2019.[1] The architecture of the P extension instruction set defines many subword SIMD instructions, including add, subtract, bit shift, compare, and multiply. In contrast to the V extension, which is also a SIMD instruction set, the P extension uses general-purpose registers to complete operations without additional vector registers. The P extension supports saturation and rounding. The

[1]http://www.andestech.com/en/2019/12/31/a-look-back-at-the-achievements-andes-made-in-2019/

possible data types for operations include integers and fixed-point numbers, and the supported element sizes are 8, 16, and 32 bits. The latest version is V0.9.11 (20211209), which defines 112 instructions.[2] This extension enables relatively low-level devices to efficiently complete complex operations, such as audio/speech decoding and processing and Internet of Things (IoT) sensor data fusion. Table 1 lists examples of instructions from the RISC-V P extension that are used in this study. In the case of these SIMD instructions, each element contains eight bits. KADD8, KSUB8, KHM8 perform saturating arithmetic operations; SLL8 and KSLRA8 perform saturating logical operations; and SCMPLT8, CMPEQ8, and SCMPLE8 perform comparison operations.

### C. LLVM

LLVM [20] is a modular compiler infrastructure that can compile multiple high-level programming languages into its LLVM IR. There are many IR transformations between different IRs; the transformation from TVM TIR to LLVM IR is one example. In practice, LLVM optimization functions [20] can be further utilized for compiler optimization with advanced architectures [26], [27]. After optimizing the IR, users can choose different backend targets to generate target-dependent code. Our work extends the LLVM with RISC-V P extension and custom instructions for conducting experiments.

## III. OUR ENHANCED COMPILATION FLOW IN TVM BASED ON THE RISC-V P SIMD EXTENSION

### A. ENABLING THE RISC-V P SIMD EXTENSION IN TVM

The process of generating assembly code from Ops written in the TVM hybrid script is divided into four phases, as shown in Fig. 3, where we consider the maximum Op as an example. In the hybrid script shown in Fig. 3, the value of the output tensor is based on the maximum value between the two input tensors. Once the behavior of this Op has been described in hybrid script, it is converted into TIR by TVM, which then generates corresponding code by the backend specified by the user. Because we have chosen the RISC-V 64-bit architecture as our backend, in this example, TVM uses the TIR to generate an LLVM IR for the RISC-V backend. When generating the LLVM IR code, TVM provides a well-designed interface that allows users to choose whether to generate the default LLVM IR code or to implement a specific class for code generation.

In our research, we have developed the RISC-V CodeGen class, which is utilized in the transformation of a TIR MaxNode into an LLVM IR intrinsic function. We illustrate this process in Algorithm 1. Here, we outline the key steps involved in Algorithm 1 that enable the lowering of TVM into LLVM with the RISC-V P extension. When the TIR visitor encounters a TIR MaxNode, the VisitMaxNode function is invoked in line 7. To generate the LLVM IR intrinsic, we first check whether the MaxNode corresponds to the SIMD type, which consists of eight lanes and eight bits in a 64-bit RISC-V target (line 8). If it is a SIMD-type MaxNode, we convert it into a TIR CallNode by utilizing the CreateVectorMax function (line 9), as defined in line 1. This function employs the MaxNode to obtain the LLVM intrinsic ID. Subsequently, the TIR CallNode is created with the right-hand side (RHS) and left-hand side (LHS) parameters (lines 3-5). Finally, the CreateLLVMIntrinsic function (line 10) generates the LLVM IR intrinsic. Additionally, we need to implement the CodeGen rules for generating RISC-V instructions from LLVM IR in the target assembly language in LLVM. This involves defining the LLVM IR intrinsics and employing LLVM's TableGen to define the RISC-V instructions. In our study, each phase described above incorporates a specific amount of implementation code, making model execution from TVM to RISC-V feasible.

**Algorithm 1** CodeGen: RVP LLVM IR From TVM SIMD TIR

```
    Input  : SIMD TIR
    Output: RVP LLVM IR
 1: procedure RISCV::CreateVectorMax(MaxOp)
 2:     ID = ::llvm::Intrinsic::riscv_simd_smax
 3:     LHSExpr = MaxOp.lhs()
 4:     RHSExpr = MaxOp.rhs()
 5:     return tir::Call(ID, LHSExpr, RHSExpr)
 6: end procedure
 7: procedure RISCV::VisitMaxNode(MaxOp)
 8:     if IsSIMDType(MaxOp) then
 9:         CallNode = CreateVectorMax(MaxOp)
10:         return tir::CreateLLVMIntrisic(CallNode)
11:     else
12:         return tvm::llvm::VisitMaxNode(MaxOp)
13:     end if
14: end procedure
```

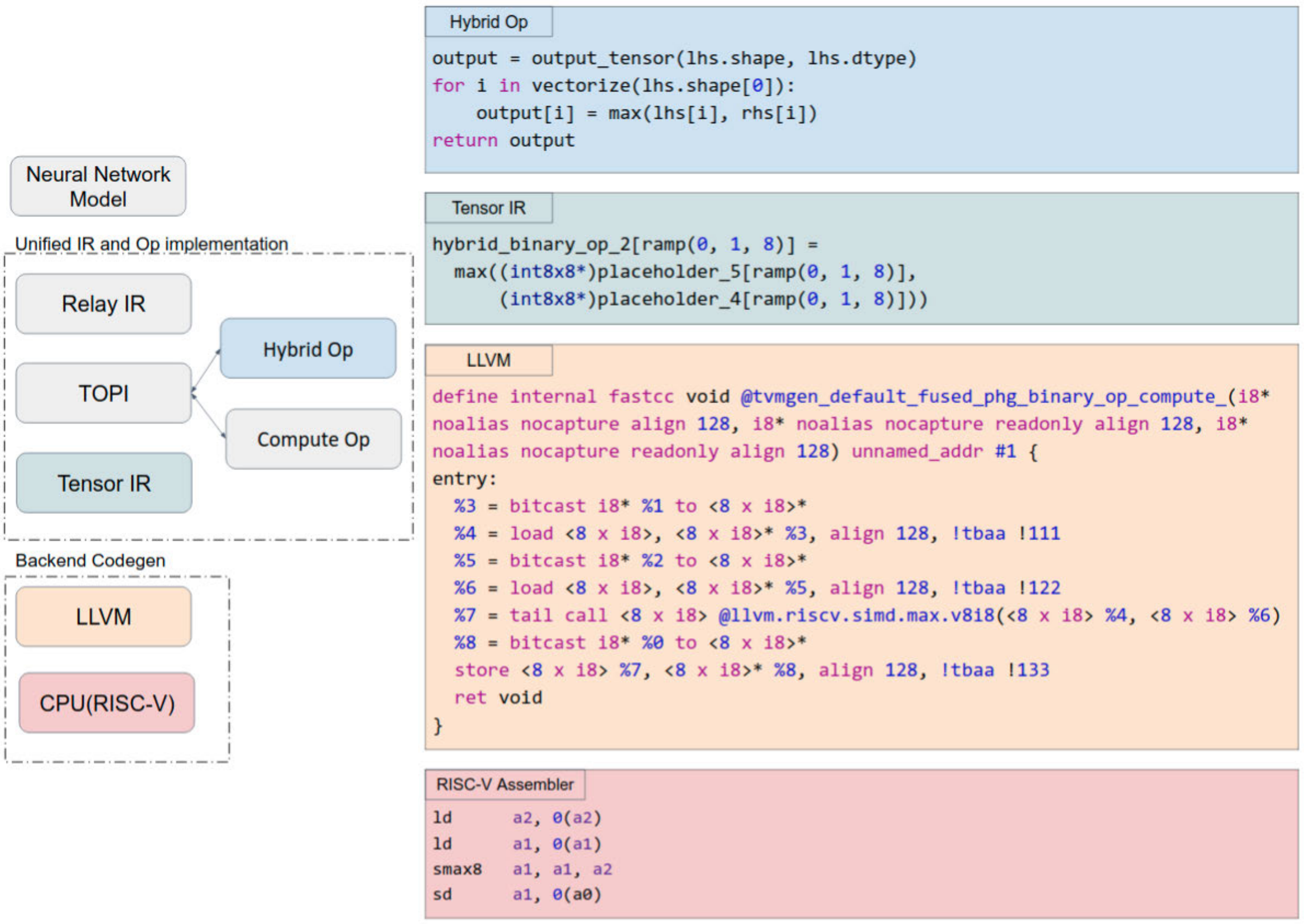
### B. ADDING CUSTOM INSTRUCTIONS TO THE RISC-V GNU TOOLCHAIN

To add custom instructions in the RISC-V GNU toolchain for the RISC-V architecture, certain modifications are necessary. First, the format of the custom instructions needs to be defined in the "riscv-opc.h"[3] file, specifying the opcode, encoding, and other relevant information. Second, corresponding modifications are needed in the "riscv-opc.c"[4] file, which handles the decoding and encoding of RISC-V instructions. These modifications ensure that the custom instructions will be properly recognized and processed by the RISC-V GCC compiler during compilation. Additionally, adjustments are required in the instruction selection logic, instruction selection table, and target description file of GCC to fully support and generate code for the custom instructions. With these modifications, GCC can be used to effectively

[2]https://github.com/riscv/riscv-p-spec/commit/5a12c90b2c206c501a448 9eb79e5d4d46afa1014

[3]riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h

[4]riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c



```
output = output_tensor(lhs.shape, lhs.dtype)
for i in vectorize(lhs.shape[0]):
    output[i] = max(lhs[i], rhs[i])
return output
```

```
hybrid_binary_op_2[ramp(0, 1, 8)] =
  max((int8x8*)placeholder_5[ramp(0, 1, 8)],
      (int8x8*)placeholder_4[ramp(0, 1, 8)]))
```

```
define internal fastcc void @tvmgen_default_fused_phg_binary_op_compute_(i8*
noalias nocapture align 128, i8* noalias nocapture readonly align 128, i8*
noalias nocapture readonly align 128) unnamed_addr #1 {
entry:
  %3 = bitcast i8* %1 to <8 x i8>*
  %4 = load <8 x i8>, <8 x i8>* %3, align 128, !tbaa !111
  %5 = bitcast i8* %2 to <8 x i8>*
  %6 = load <8 x i8>, <8 x i8>* %5, align 128, !tbaa !122
  %7 = tail call <8 x i8> @llvm.riscv.simd.max.v8i8(<8 x i8> %4, <8 x i8> %6)
  %8 = bitcast i8* %0 to <8 x i8>*
  store <8 x i8> %7, <8 x i8>* %8, align 128, !tbaa !133
  ret void
}
```

```
ld      a2, 0(a2)
ld      a1, 0(a1)
smax8   a1, a1, a2
sd      a1, 0(a0)
```

**FIGURE 3.** **TVM compilation flow: taking the maximum Op as an example, from hybrid script to RISC-V assembly.**

incorporate and optimize custom instructions within the RISC-V architecture.

## IV. STRATEGIES FOR REWRITING TVM HYBRID SCRIPT

This section presents the rewriting strategies proposed in this study. As shown in Fig. 1, hybrid script is a Python-based subgrammar in the TVM architecture that is a description language suitable for optimizing preprocessing and postprocessing operations. This hybrid script supports a flexible programming model and provides various optimizations for application programming interfaces (APIs), including loop annotations (unroll, parallel, vectorize, and bind), loop manipulations (split and fuse), and reordering. This allows developers to schedule execution sequences at the source level and write more efficient programs.

### A. SUBWORD SIMD VECTORIZATION

In AI and deep learning computations, for loop is frequently utilized to execute identical operations on each element of a tensor. However, conventional for loop is typically designed to process one data element at a time, resulting in suboptimal computational efficiency. To address this issue, we propose an approach that enhances performance by reconfiguring a for loop using hybrid script and generating RISC-V subword SIMD instructions. For example, consider the operation of vector addition. To optimize its execution, we split the for loop into inner and outer loops. The inner loop uses the vectorize syntax and the vector size of the subword SIMD instruction, and the outer loop controls the number of times the inner loop is executed. In the subsequent compilation

```
# Original
for i in range(len):
    output[i] = lv[i] + rv[i]


# After Rewrite
XLEN = 8 #v8i8
for i in range(len/XLEN):
  for j in vectorize(XLEN):
    output[i * XLEN + j] = lv[i * XLEN + j] +
    rv[i * XLEN + j]
```

**Listing 3.** **Rewriting strategy with subword SIMD vectorization: if the data type of input and output data is int8 or int16, then we can tile the for loop and use the subword SIMD instruction KADD8 or KADD16 to improve performance.**

process, the TIR will be converted into an LLVM IR. During this stage, different subword SIMD instructions will be chosen based on the data type. For signed 8-bit data, the KADD8 instruction will be used; for unsigned 8-bit data, the UKADD8 instruction will be used. Similarly, for signed or unsigned 16-bit data, the TIR description will be converted into KADD16 or UKADD16 instructions, respectively.

For a for loop to be amenable to vectorization, the following conditions need to be met. 1. The body of the for loop must not contain other nested loops. 2. The number of iterations of the loop should exceed the vector size. 3. Each operation within the loop body must have a corresponding SIMD instruction. 4. Each iteration must have no dependency on any other iteration. If the conditions for loop vectorization are satisfied, then the for loop can be split into outer and inner loops. The body of the inner loop is represented using "for j in vectorize(XLEN)" notation, enabling acceleration through SIMD instructions. Here, XLEN represents the number of data elements that can be processed in a single SIMD instruction. Conversely, the outer loop is a conventional sequentially executed for loop. The difference is that the iteration count of this loop is divided by XLEN. This is expressed as "for i in range(len/XLEN)" to ensure optimal utilization of hardware resources.

Listing 3 shows an example of this strategy. The original version has a loop structure that contains descriptions of the initialization, conditions, and advancement to the next iteration. After applying the proposed strategy, an additional loop is added as the outer loop, and the iteration count is divided by XLEN. The inner loop runs over the index value from zero to XLEN - 1, and we can vectorize it with the KADD8 SIMD instruction. Thus, rewriting the for loop in this way allows us to improve the performance.

## B. TERNARY OPERATION

In general, if-else statements make conditional decisions based on specific criteria. However, this approach may introduce branching, which can adversely affect performance due to potential pipeline stalls and mispredictions. To solve this problem, we propose replacing the if-else construction with a ternary operation. This strategy is accomplished by combining the select function of TIR in TVM with the instructions of the RISC-V P extension. Two instructions are used to complete the ternary operation. The first is a comparison instruction CMPEQ8, which performs elementwise equality comparisons of 8-bit integers in parallel for use in if statements. The other is the BPICK instruction, which selects from two source operands by a bitmask in the third operand for the expression after if or else statements are found to be true.

For this strategy to be applicable, the following conditions must be met: 1. The conditions evaluated by the if statements must be equality, greater-than, or less-than comparisons. Such operations are well-suited for vectorized execution. 2. The data types of the variables being compared in the if-else statements and being assigned should all be int8. This aligns with the capabilities of the RISC-V P extension for efficient parallel integer operations. 3. Both the if and else branches should declare and assign values to the same variable. This is essential for ensuring consistent results when utilizing ternary operations. If the conditions for the ternary operation strategy are satisfied, then we can apply the following form:

$$\text{result} \;=\; \text{true value} \;\;\textbf{if}\;\; \text{conditions} \;\;\textbf{else}\;\; \text{false value}$$

Listing 4 illustrates an example of this strategy. The original version presents a program description using if-else statements. Upon applying the proposed strategy, the program description transforms into a ternary operation. This operation involves three operands: the expression executed if the condition is true (is_valid $=$ 1), followed by another expression specifying the condition to be checked (if area $>$ 0), and finally, a value determining the alternative expression (is_valid $=$ 0) to be executed if the condition is false.

```
# Original
    if area > 0:
        is_valid = 1;
    else:
        is_valid = 0;


# After Rewrite
    is_valid = 1 if area > 0 else 0
```

**Listing 4.** Rewriting Strategy Based on Ternary Operation: If the sole operations within an if-else construct are assignment operations, the construct can be refactored using ternary syntax. Additionally, performance improvement can be achieved by leveraging subword SIMD instructions, specifically CMPEQ8 and BPICK.

```
# Original
area = area_a + area_b - area_intersection

# After Rewrite
area = area_b - area_intersection
area = area + area_a
```

**Listing 5.** Rewriting Strategy Considering Saturation Arithmetic: Due to the utilization of subword SIMD instructions, there is a reduction in the value range. Consequently, the order of operations needs to be adjusted to prevent saturation without compromising the integrity of the results.

## C. SATURATION ARITHMETIC

Quantization is a common technique for optimizing the computations of AI models. Performance can be improved by reducing the bit width of the data to minimize the amount of data transferred and the number of computing cycles needed. However, reducing the bit width also reduces the range of representable values. For example, for an int8 variable (an 8-bit integer), the representable value range is $-128$ to $+127$. Saturation occurs when the result of an arithmetic operation exceeds this range, causing apparent incorrect. For example, if we use KADD8 to add 200 $+$ 100 since the result exceeds the representable range, the final result will be 127 instead of 300. To avoid this situation, we can rearrange the order of operations to keep the results within the representable range. For this rewriting strategy, to be applicable the following conditions need to be met: 1. Consistency of data types: all variables involved in the operations, including the input and outputs, should share the same data type. 2. Use of test data to determine value ranges: test data should be used to evaluate the possibility of saturation or underflow.

If the above conditions are met, we can rearrange the order of operations to avoid saturation or underflow. If the analysis indicates that some operations may be saturated, the data range can be reduced by performing subtraction or division first. This helps prevent saturation. Conversely, if the analysis indicates that there is a risk of underflow, addition or multiplication can be performed first to expand the range of the data.

Listing 5 shows an example of this strategy, where the data type is int8. The original version is a program describing the process of adding area_a and area_b and then subtracting area_intersection. After the proposed strategy is applied, area_intersection is first subtracted from area_b, and area_a is then added to obtain the final value. This strategy can reduce the occurrence of saturation beyond the numerical range during runtime execution.

### D. QUANTIZED INTEGER DIVISION

Among general arithmetic operations, the division operation involves floating-point calculations. When both the input and output values are int8 type, the typical process is to first convert the inputs into floating-point numbers for division and then convert the result back to an integer; however, this process incurs additional computational overhead due to these conversions and the floating-point division operation itself. Python provides two forms of division, namely, standard floating-point division '/' and floor division '//', where the latter returns the integer quotient. Therefore, when all variables involved in an operation are integers, leveraging floor division can be advantageous. By combining this operation with low-level code generation to utilize hardware instructions specifically designed for integer division, the need for intermediate floating-point conversions can be mitigated.

For this rewriting strategy to be applicable, only one condition needs to be satisfied: all variables involved in the operations, including the inputs and outputs, should share the same int8 data type. When this condition is met, the original floating-point division operation can be transformed into an integer division operation to obtain the integer quotient directly.

Listing 6 demonstrates the application of this strategy. In the provided example, both the dividend and divisor in the original version are typed as floating-point data. Consequently, when performing an int8 division operation, the dividend and divisor must undergo forced conversion into floating-point numbers for the division, followed by conversion back to int8 to obtain the final result. The proposed strategy introduces an additional step using a division instruction, 'div,' which eliminates the need for reverting to a floating-point representation. This step facilitates direct SIMD integer division, denoted by the double slash (//). Within a 64-bit system architecture, SIMD vectorization can be leveraged for the custom hardware instruction 'div,' dividing the elements into eight components, each comprising an 8-bit integer (int8). As the int8 data type cannot represent decimal places, the result is an integer division outcome, yielding a quotient value of QUOT8 (refer to Section V-B4 for more details).

```

#Original
quotient_float = dividend/divisor


#After Rewrite, if the data types of the
    dividend and divisor are int8, we use a
    custom instruction, quot8, to return the
    integer quotient.

quotient_int = dividend // divisor
```

**Listing 6.** **Rewriting strategy for quantized integer division: if the value range is from −128 to 127 or from 0 to 255, then we can rewrite the integer division operation in hybrid script syntax and use a custom SIMD instruction QUOT8 to improve performance.**

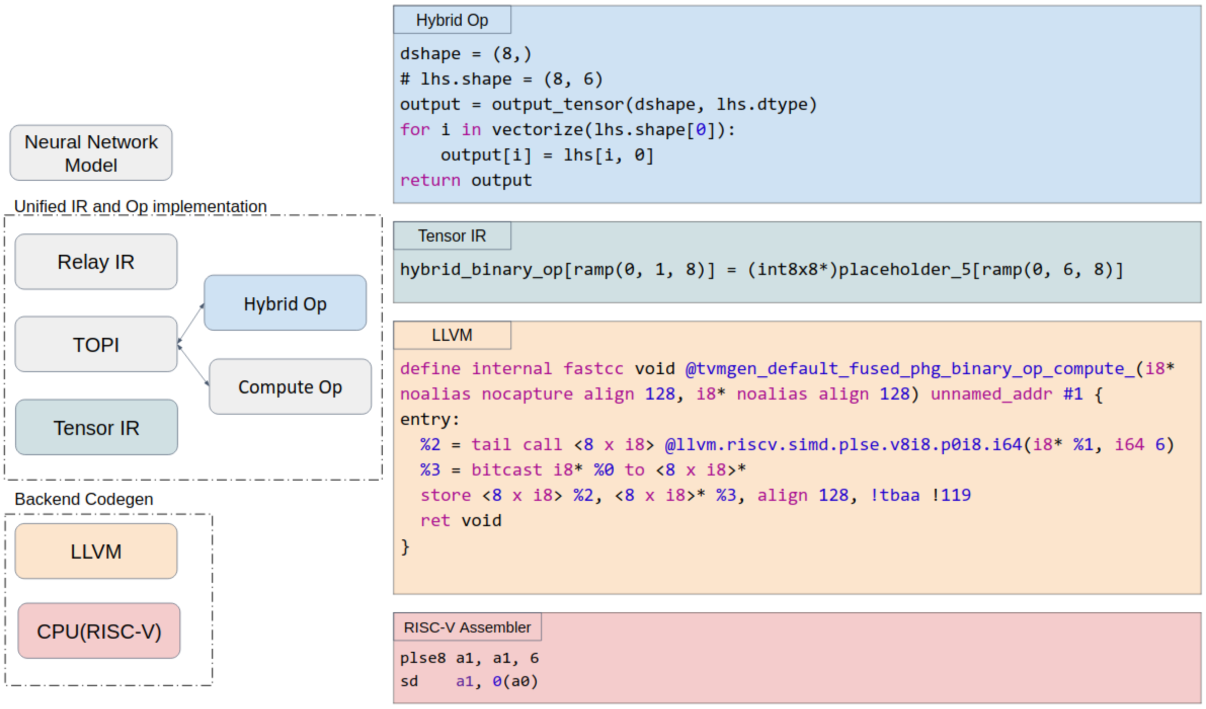
**Algorithm 2** CodeGen Custom Instruction LLVM IR From TVM TIR

```
    Input  : TIR
    Output: LLVM IR with custom instruction
 1: procedure RISCV::CreateStrideLoad(LoadOp)
 2:     ID = ::llvm::Intrinsic::riscv_custom_plse8
 3:     BasePtr = tvm::CreateBufferPtr(LoadOp)
 4:     Stride = LoadOp.stride
 5:     return tir::Call(ID, BasePtr, Stride)
 6: end procedure
 7: procedure RISCV::VisitLoadNode(LoadOp)
 8:     if IsSIMDType(LoadOp) and LoadOp.stride > 1 then
 9:         CallNode = CreateStrideLoad(LoadOp)
10:         return tir::CreateLLVMIntrisic(CallNode)
11:     else
12:         return tvm::llvm::VisitLoadNode(Op)
13:     end if
14: end procedure
```

## V. DESIGN OF CUSTOM INSTRUCTIONS BASED ON THE RISC-V P EXTENSION

### A. CUSTOM INSTRUCTION FLOW

In addition to the original RVP instructions, we propose custom instructions to improve the preprocessing and post-processing performance. Additional steps were needed to enable these custom instructions in TVM. The first step was to define the instruction format and opcode. The second step was to add the new custom instructions to the RISC-V



```
dshape = (8,)
# lhs.shape = (8, 6)
output = output_tensor(dshape, lhs.dtype)
for i in vectorize(lhs.shape[0]):
    output[i] = lhs[i, 0]
return output
```

```
hybrid_binary_op[ramp(0, 1, 8)] = (int8x8*)placeholder_5[ramp(0, 6, 8)]
```

```
define internal fastcc void @tvmgen_default_fused_phg_binary_op_compute_(i8*
noalias nocapture align 128, i8* noalias align 128) unnamed_addr #1 {
entry:
  %2 = tail call <8 x i8> @llvm.riscv.simd.plse.v8i8.p0i8.i64(i8* %1, i64 6)
  %3 = bitcast i8* %0 to <8 x i8>*
  store <8 x i8> %2, <8 x i8>* %3, align 128, !tbaa !119
  ret void
}
```

```
plse8 a1, a1, 6
sd    a1, 0(a0)
```

**FIGURE 4.** Enabling RISC-V custom instructions. We take PLSE8 as an example to show how we enable the transformation of custom instructions from TVM to RISC-V assembly language.

| | 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| **Custom Instruction** | **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| plu32 | 1 | index | base addr | 6 | destination | 11 |
| psu32 | 1 | source | index | 6 | base addr | 43 |
| quot8 | 1 | divisor | dividend | 0 | destination | 11 |

| | 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| **Custom Instruction** | **imm[31:20]** | **rs1** | **funct3** | **rd** | **opcode** |
| plse8 | stride | base addr | 3 | destination | 11 |
| psse8 | stride | source | 7 | base addr | 43 |

| | 31 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| **Custom Instruction** | **imm[31:15]** | **funct3** | **rd** | **opcode** |
| pcsv8 | immediate[31:15] | 7 | destination | 11 |

**FIGURE 5.** Custom instruction format proposed to improve preprocessing and postprocessing.

GNU toolchain. As a third step, we also needed to define the instruction behavior in the RISC-V simulator Spike because we used the Spike to conduct our experiments. Once the behavior of a custom instruction had been defined in the simulator and the RISC-V GNU toolchain, TVM implementation was needed. Fig. 4 shows the process of generating custom instructions from hybrid script to RISC-V assembly. We use the custom instruction PLSE8 as an example. PLSE8 is the memory stride load instruction, which can load data from memory to registers at given distances. To generate the PLSE8 instruction, we obtain data for every six elements from an $8 \times 6$ tensor, as described in the hybrid script shown in Fig. 4. To utilize the PLSE8 instruction, a TIR with the corresponding pattern generated from the hybrid script, is needed; then, the TIR can be transformed through CodeGen into LLVM IR, which includes the intrinsics corresponding to the custom instruction. The stride memory load instruction plse8 will be briefly introduced in a later subsection. Algorithm 2 shows how CodeGen processes for this custom instruction using TIR. The algorithm starts by visiting the TIR LoadOp in line 7 and checking whether this LoadOp is of the SIMD type and whether its stride is greater than that in line 8. If this condition is true, then the algorithm proceeds to line 1 to create the TIR CallNode by the LLVM intrinsic ID, the base pointer, and the stride value from lines 2 to 5. After creating the TIR CallNode, the algorithm proceeds to line 10 to generate the LLVM intrinsic. Finally, we use the corresponding LLVM IR to generate the RISC-V assembly.

## B. CUSTOM INSTRUCTIONS

RISC-V, an open Instruction Set Architecture (ISA), comprises base instruction sets and various extensions, among which the P extension is utilized in our experiments. In addition to the originally defined instruction set, RISC-V allows for user-defined instructions. Instruction opcode is specifically reserved for these instructions in defined positions, distinct from the standard instruction set. Notably, RISC-V allocates a custom 4-cluster instruction type with an instruction length of 32 bits. In this study, we have devised six custom instructions featuring unique 0/1 opcodes to augment the P-extension set. These instructions encompass four memory load and store instructions, one for broadcasting, and another for integer division.

### 1) PLU32 AND PSU32

General load/store instructions involve reading or writing a single data element from or to a given memory address. We design two custom instructions plu32 and psu32, to not only read and write multiple data elements at one time but also read and write by a given index.

plu32 is the memory index load instruction. In this instruction, a base register (rs1) stores the base address, and a general-purpose register (rs2) contains two 32-bit integers (offsets), which are added to the base register, to obtain the memory location from which to fetch the data. The fetched data are then converted into two 32-bit data elements to be stored in the destination register(rd), a 64-bit general-purpose register that can hold two 32-bit integers.

psu32 is the memory index store instruction. In this instruction, rd stores the base address, rs1 contains two 32-bit integers (offsets), rs2 also contains two 32-bit integers (data), and two memory locations are obtained by adding the two offsets to the base store. Then, the two int32 data values in rs2 are stored in these two memory locations. The instruction format is shown in Fig. 5.

Listings 7 to 9 compare the differences between baseline and custom instructions corresponding to the same hybrid script. In the baseline version(Listing 8), out[indices[i]] and updates[i] need to be loaded into registers a4 and a5, respectively, through two load instructions and one add instruction. Then, the addition is completed through the add instruction, and the result needs to be written back to the memory through one addi instruction and one load instruction. In contrast to the baseline version, only 10 instructions are needed when using our custom instructions. First, out[indices[i]] is loaded into register a2 via the custom instruction plu32, and then the addition operation is completed through kadd32. Finally, the result is written back to the memory through psu32.

The pattern of the hybrid script used to generate the plu32 and psu32 instructions is shown in line 2 of Listing 7. On the LHS of the plus-equal Op, the output tensor takes the indices indicated by the tensor of the other indices. Once the TIR description corresponding to this pattern is obtained, in this example, we can reduce the number of instructions by seven by using our custom instructions plu32 and psu32.

```
1 for i in range(indices.shape[0]):
2     out[indices[i]] += updates[i]
```

**Listing 7.** **Hybrid script for index load and store.**

```
.LBB1_1:
   slli  a3, a3, 32
   srai  a3, a3, 32
   slli  a3, a3, 2
   add   a3, s2, a3
   lw   a4, 0(s0)
   lw   a5, 0(a3)
   add   a4, a5, a4
   sw   a4, 0(a3)
   addi  s1, s1, 4
   addi  s0, s0, 4
   addi  a0, a0, -1
   beqz  a0, .LBB1_4
   lw   a3, 0(s1)
   blt   a2, a3, .LBB1_1
.LBB1_3:
   add   a3, a3, a1
   j  .LBB1_1
```

**Listing 8.** **Baseline version of index load.**

```
.LBB1_1:
   ld  a1, 0(s1)
   plu32  a2, s2, a1
   ld  a3, 0(s0)
   kadd32  a2, a2, a3
   psu32  s2, a1, a2
   addi  s1, s1, 8
   addi  s0, s0, 8
   addi  a0, a0, -1
   bnez  a0, .LBB1_1
```

**Listing 9.** **Optimized version of index load.**

### 2) PLSE8 AND PSSE8

General load/store instructions involve reading or writing a single data element from or to a given memory address. We design plse8 and psse8 to not only read and write multiple data elements at one time but also read and write by a given stride.

plse8 is the memory stride load instruction. In this instruction, rs1 is the base register, which immediately records the value of the stride. This instruction grabs a byte from each of eight memory locations by the base address and stride and stores these eight pieces of data in the 64-bit destination register rd.

psse8 is the memory stride store instruction. In this instruction, rd stores the base address and immediately records the value of the stride, and rs1 holds the eight-byte data that are to be stored in the memory. This instruction stores the eight data elements in rs1 into eight memory locations by to the base address and stride. The instruction format is shown in Fig. 5.

Listings 10 to 12 compare the differences between baseline and custom instructions corresponding to the same hybrid script. In the baseline version, 32 instructions are needed to complete the addition of the eight int8 elements of two arrays. For each element, two lb instructions are needed to load the data from the LHS and RHS arrays into registers a3 and a4. Then, the contents of registers a3 and a4 are added, the result is stored back into a3, and finally, the result from register a3 is stored back into the memory through the store instruction. In contrast, only four instructions are needed when using our custom instructions. The first and second instructions load the eight elements of the first and second arrays with a stride of 6 into a2 and a1, respectively; for this, the custom instruction is used plse8. Then, using the SIMD add instruction kadd8, the eight elements in each register are added and the results are stored in register a1; then the results are stored back into memory with stride 8 through the psse8 instruction. In this way, the number of instructions can be reduced by twenty-eight through the use of our custom instructions. The pattern of the hybrid script used to generate plse8 and psse8 instructions is shown in line 2 of Listing 10. Because the hybrid script is row-major, annotating the column loop with the vectorize keyword, then will access the memory with a fixed-length offset, referred to as the stride. Once the TIR corresponding to this pattern is obtained, we can generate the corresponding LLVM Intrinsics.

```
1 //In NMS the shapes of the LHS and RHS are both N
      x 6, where N is the number of bounding boxes,
      and each bounding box has 6 data elements.
2 for i in vectorize(8):
3     output[i, 0] = lhs[i, 0] + rhs[i, 0]
```

**Listing 10. Hybrid script for stride load and store.**

```
lb  a3, 0(a2)
lb  a4, 0(a1)
add  a3, a4, a3
sb  a3, 0(a0)
lb  a3, 6(a2)
lb  a4, 6(a1)
add  a3, a4, a3
sb  a3, 6(a0)
lb  a3, 12(a2)
lb  a4, 12(a1)
add  a3, a4, a3
sb  a3, 12(a0)
lb  a3, 18(a2)
lb  a4, 18(a1)
add  a3, a4, a3
sb  a3, 18(a0)
lb  a3, 24(a2)
lb  a4, 24(a1)
add  a3, a4, a3
sb  a3, 24(a0)
lb  a3, 30(a2)
lb  a4, 30(a1)
add  a3, a4, a3
sb  a3, 30(a0)
lb  a3, 36(a2)
lb  a4, 36(a1)
add  a3, a4, a3
sb  a3, 36(a0)
lb  a2, 42(a2)
lb  a1, 42(a1)
add  a1, a1, a2
sb  a1, 42(a0)
```

**Listing 11. Baseline version of stride load and store.**

```
plse8  a2, a2, 6
plse8  a1, a1, 6
kadd8  a1, a1, a2
psse8  a0, a1, 6
```

**Listing 12. Optimized version of stride load and store.**

### 3) PCSV8

Typically, addi (addition of a constant value) is used to set a register to a given value. We design a custom instruction pcsv8, which copies an immediate value to each element of rd, with an element size of eight bits. There are eight elements in the RV64 architecture. The instruction format is shown in Fig. 5.

Listings 13 to 15 compare the differences between baseline and custom instructions corresponding to the same hybrid script. In the baseline version, 24 instructions are needed to complete the operation on an eight-element array, adding a constant value to each element. In each iteration, the LHS element needs to be loaded into register a2 through the load instruction (ld). Then, a constant value of 3 is added to a2 through the addi instruction, and the result stored in register a2 is subsequently written back to the memory through the store instruction. In contrast, only 4 instructions are needed when our custom instruction is used. First, the ld instruction is used to load the continuous data of the HLS array into a1; then the constant value of 3 is copied to each element in a2 through the pcsv8 instruction. Subsequently, a1 and a2 are added through kadd8 and the result is written back to a1. Finally, the result stored in a1 is written back to the memory using the store instruction.

The pattern of the hybrid script used to generate the pcsv8 instruction is shown in line 2 of Listing 13. In this case, we annotate a loop with the vectorize keyword, and the loop involves a binary Op with one tensor and one constant scalar. Once the TIR corresponding to this pattern is obtained, we can generate the corresponding LLVM Intrinsics. In this example, we reduce the number of instructions by 20 by using the custom instruction pcsv8.

```
1 for i in vectorize(8):
2     output[i] = lhs[i] + int8(3)
```

**Listing 13. Hybrid script for constant add.**

```
lb   a2,  0(a1)
addi  a2, a2,  3
sb   a2,  0(a0)
lb   a2,  1(a1)
addi  a2, a2,  3
sb   a2,  1(a0)
lb   a2,  2(a1)
addi  a2, a2,  3
sb   a2,  2(a0)
lb   a2,  3(a1)
addi  a2, a2,  3
sb   a2,  3(a0)
lb   a2,  4(a1)
addi  a2, a2,  3
sb   a2,  4(a0)
lb   a2,  5(a1)
addi  a2, a2,  3
sb   a2,  5(a0)
lb   a2,  6(a1)
addi  a2, a2,  3
sb   a2,  6(a0)
lb   a1,  7(a1)
addi  a1, a1,  3
sb   a1,  7(a0)
```

**Listing 14. Baseline version of constant add.**

```
ld  a1,  0(a1)
pcsv8  a2,  3
kadd8  a1,  a1,  a2
sd  a1,  0(a0)
```

**Listing 15. Optimized version of constant add.**

#### 4) QUOT8

In the RISC-V basic instruction set, an instruction for integer division (div) is provided. Here we further extend it to a SIMD version, by designing quot8, which is an integer division instruction that performs elementwise division on rs1 and rs2 and returns the quotients to rd. The element size is eight bits. There are eight elements in the RV64 architecture. The instruction format is shown in Fig. 5.

Listings 16, 17 and 18 compare the differences between baseline and custom instructions corresponding to the same hybrid script. In the baseline version, 32 instructions are needed to complete the division of eight pairs of int8 elements. For each data pair, two load instructions are needed to load the data from the memory to register a3 and a4. Then, integer division is performed, and the results are stored back to a3. Subsequently, the contents of a3 are written back to the memory through the store instruction. In contrast, only four instructions are needed when our custom instruction is used. First, two load instructions(ld) are used to load two sets of eight consecutive int8 data elements into registers a2 and a1 from the memory. Then, the integer division and quotient operations are performed through the custom instruction quot8, and the results are written back to register a1. Finally, the results are written back to the memory through the store instruction. The pattern of the hybrid script used to generate the quot8 instruction is shown in line 2 of Listing 16. In this case, the div function, which represents our integer quotient function, is used. Once the TIR corresponds to this pattern, we can generate the corresponding LLVM intrinsics. In this example, we reduce the number of instructions by 26 by using our custom instruction quot8.

```
1 for i in vectorize(8):
2     output[i] = lhs[i] // rhs[i]
```

**Listing 16. Hybrid script for division.**

```
lb a3,  0(a2)
lb a4,  0(a1)
div a3,  a4,  a3
sb a3,  0(a1)
lb a3,  1(a2)
lb a4,  1(a1)
div a3,  a4,  a3
sb a3,  1(a0)
lb a3,  2(a2)
lb a4,  2(a1)
div a3,  a4,  a3
sb a3,  2(a0)
lb a3,  3(a2)
lb a4,  3(a1)
div a3,  a4,  a3
sb a3,  3(a0)
lb a3,  4(a2)
lb a4,  4(a1)
div a3,  a4,  a3
sb a3,  4(a0)
lb a3,  5(a2)
lb a4,  5(a1)
div a3,  a4,  a3
sb a3,  5(a0)
lb a3,  6(a2)
lb a4,  6(a1)
div a3,  a4,  a3
sb a3,  6(a0)
lb a2,  7(a2)
lb a1,  7(a1)
div a1,  a1,  a2
sb a1,  7(a0)
```

**Listing 17. Baseline version of division.**

```
ld   a2,  0(a2)
ld   a1,  0(a1)
quot8   a1,  a1,a2
sd   a1,  0(a0)
```

**Listing 18. Optimized version of division.**

### C. RUNNING EXAMPLE: NONMAXIMUM SUPPRESSION (NMS)

NMS is widely used for object detection in computer vision and deep learning applications to retain the bounding boxes that are most likely to correspond to objects of interest. Here, we use NMS as an example to illustrate the performance improvement archived with our rewriting strategies and custom instructions. The input for NMS is a set of sorted object data, where each object has six pieces of associated information: the confidence of the object, the class of the object, and the coordinate information of the upper left and lower right corners. All the candidate objects are sorted by their confidence; the higher the confidence value is, the higher the ranking. Fig. 6 shows the flow of NMS. In the first step, the first element is selected as the candidate, and it is assumed that this candidate indeed contains an object.

Second, other bounding boxes are suppressed if they are too close. Out of the remaining bounding boxes, the next top-scoring box is then selected, and the first and second steps are repeated until no more bounding boxes remain. Third, all bounding boxes that have not been suppressed are returned as results.

Listing 19 shows the baseline version of NMS, in which we cannot vectorize if-else statements, nor min, max add, or other operations. We apply our rewriting rules for a RISC-V 64-bit hardware target with the P extension; accordingly, the vector size is eight and each element is of the int8 type. Listing 20 shows the optimized version of the NMS algorithm, which we flatten into two loops. The number of iterations of the inner loop is equal to the vector size, while for the outer loop, the number of iterations is the original range divided by 8. The main difference between Listings 19 and 20 is that the code of the latter is vectorizable, whereas that of the former is not because of the if-else construction from Lines 11 to 14 in Listing 19, which cannot be vectorized. In Listing 20, we rewrite the if-else construction using the ternary operation strategy described in Section IV-B to make it vectorizable.

```
1  for i in range(valid_count):
2    bbox_a t/b/l/r
3    for j in range(i, valid_count):
4      bbx_b t/b/l/r
5      # calculate overlap width and height
6      w = max(zero, min(a_r, b_r) - max(a_l, b_l))
7      h = max(zero, min(a_b, b_b) - max(a_t, b_t))
8      overlap_area = h * w
9      union_area = (a_r - a_l) * (a_b - a_t) + (b_r
       - b_l) * (b_b - b_t) - overlap_area
10     iou = zero if union_area <= zero else
       overlap_area / union_area
11     if iou >= iou_threshold:
12         is_valid_box = 0
13     else
14         is_valid_box = 1
```

**Listing 19.** Baseline version of the NMS algorithm.

```
1  XLEN = 8 #V8I8
2  iou_threshold_invs = int8(1/iou_threshold)
3  for i in range(valid_count):
4    for j in range(i, valid_count/ XLEN):
5      for k in vectorize(XLEN):
6        max_l = max(bbox_a_l, bbox_b_l)
7        min_r = min(bbox_a_r, bbox_b_r)
8        max_t = max(bbox_a_t, bbox_b_t)
9        min_b = min(bbox_b_b, bbox_b_b)
10       w = min_r - max_l if min_r > max_l else 0
11       h = min_b - max_t if min_t > max_t else 0
12       overlap_area = h * w
13       overlap_area = 1 if overlap_area == 0 else
       overlap_area
14       u = area_bbox_a - overlap_area
15       u = u + area_bbox_b
16       iou =  union_area / / overlap_area
17       is_valid_box = 0 if iou <=
       iou_threshold_invs else 1
```

**Listing 20.** Optimized version of the NMS algorithm.

Once the hybrid script is vectorizable, we divide the axis with iterator j in line 3 of Listing 19 by XLEN, resulting in lines 4 and 5 of Listing 20, using the strategy described in Section IV-A. Notably, we use the vectorize annotation in line 5 to tell TVM that the instructions in this loop should be generated into SIMD instructions. Lines 6 to 9 in Listing 19 compute the areas of the intersection. and union between two boxes. To maintain the accuracy of the data type of int8, it is necessary to check whether the width and height of the intersection area are greater than zero in lines 10 and 11 of Listing 20. To compute the area of the union of the two boxes under the constraints on the value range for the int8 type, it is necessary to first subtract the intersection area from the area of one box and then add the area of the other box to ensure that the value of the union area will be as precise as possible, as shown in lines 14 and 15 of Listing 20 and described in Section IV-C. line 10 in Listing 19 computes the intersection over union (IoU) value. Because the area of intersection of two boxes is always smaller than the area of their union, area of their union, the corresponding integer division operation cannot be directly applied in the int8 version of NMS, as the calculated IoU value would always be zero. Therefore, we modify the algorithm to calculate the union divided by the intersection and check the result against the inverse of the IoU threshold, as expressed in lines 16 and 17 of Listing 20. Thus, the rewriting of the entire NMS algorithm is completed.

## VI. EXPERIMENTS

To evaluate the performance of the proposed method, we use NMS as an example and compare the baseline NMS algorithm with the version optimized using our hybrid script rewriting strategies and the RISC-V P extension with our proposed custom instructions. We use the Microsoft COCO [12] dataset as the input data for testing and the Single Shot MultiBox detector(SSD) [28] as the neural network model for object detection.

### A. EXPERIMENTAL FLOW AND ENVIRONMENTS

Our experimental flow is divided into three major steps: compiling the compute graph through TVM, using the LLVM and GNU toolchains to compile the kernel code and TVM runtime into RISC-V ELF, and running the compiled RISC-V ELF executable on the RISC-V simulator Spike with test data to obtain the final results. Taking NMS as an example, the experimental flow is shown in Fig. 7. The first step is to compile the NMS algorithm through TVM, which generates a kernel code, execution graph, and set of parameters (Param). The kernel code is in the LLVM IR format and is the implementation of the NMS algorithm. The execution graph is a file in JavaScript Object Notation (JSON) format that describes the operation execution sequence. Param contains the pre-trained weights or constant data used in neural network model inference. There is no weight information in NMS; therefore, there are no parameters in our experiments. The RISC-V assembly code is generated as part of the LLVM tool. The host code is the main program that drives

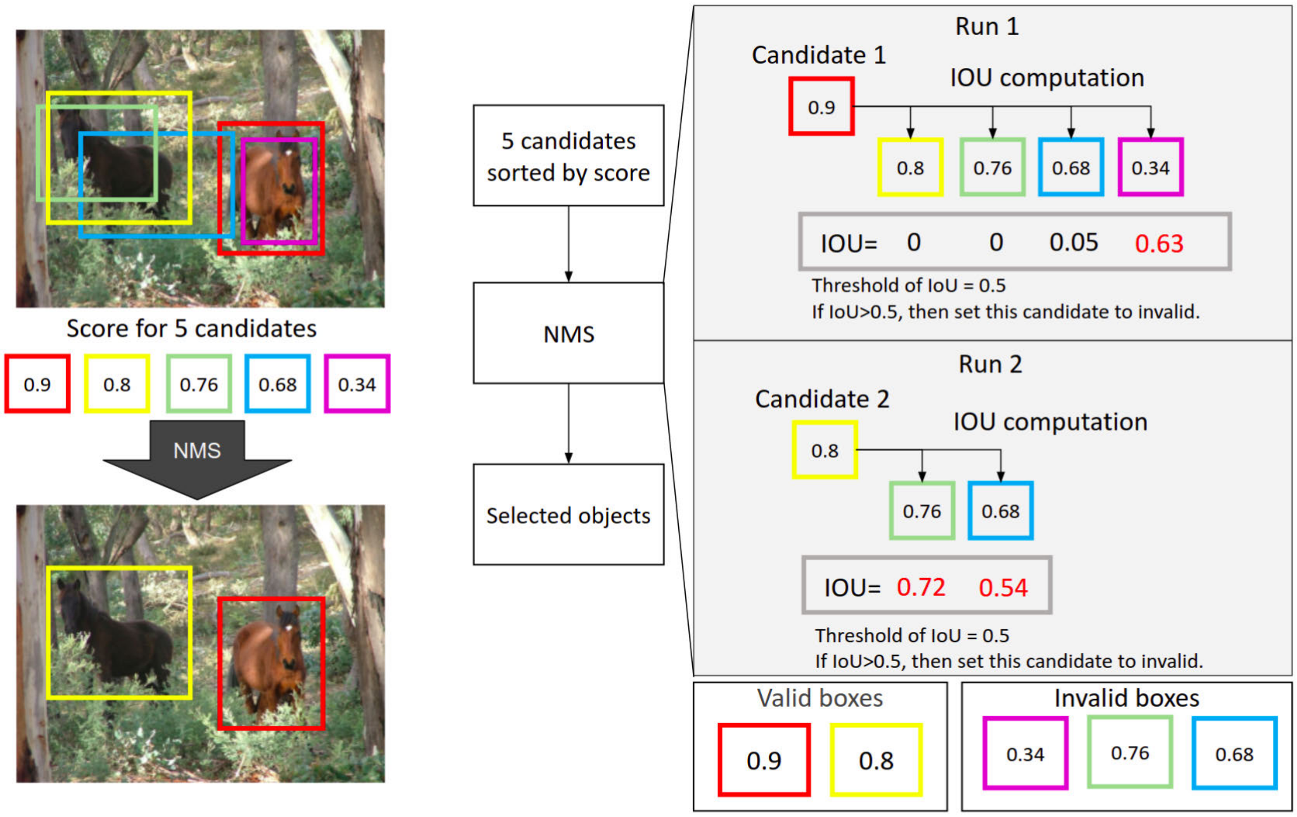


**FIGURE 6.** Flow of the nonmaximum suppression(NMS) algorithm. In this example, the input consists of five candidate bounding boxes. After the NMS operation, three are excluded and the remaining two represent the ranges of the detected objects.

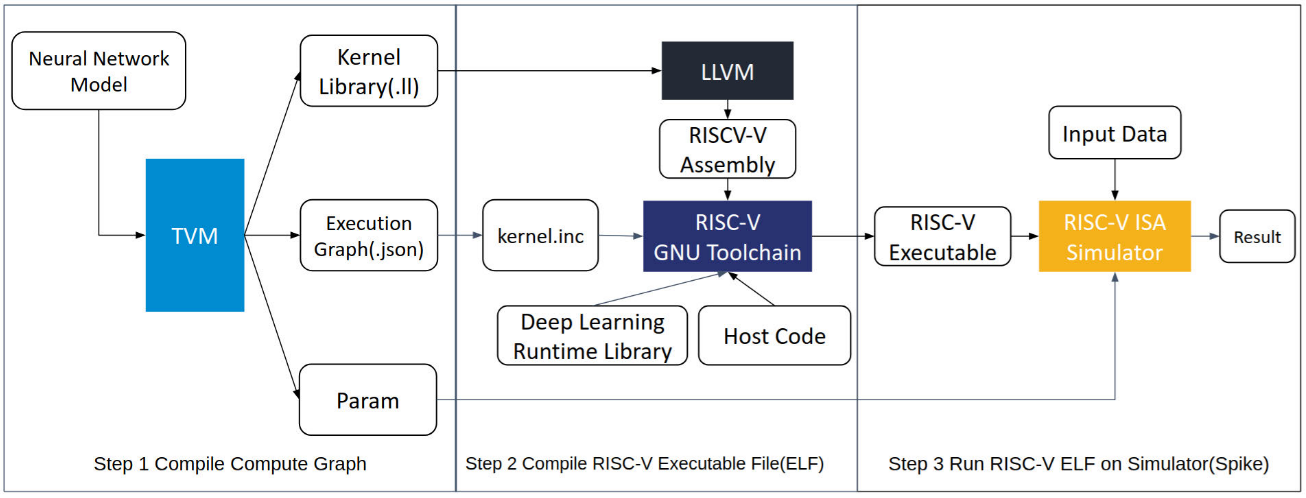


**FIGURE 7.** Compilation and execution flow with TVM on RISC-V.

the entire NMS execution flow, which includes topological sorting of the execution graph to obtain the operation process, adding parameters to the neural network model, performing inference, and generating results. The RISC-V ELF file is a RISC-V executable file compiled via the RISC-V GNU toolchain. The input bounding boxes are the data generated by the object detection model. We use TVM to produce a model containing the NMS operator and other transformed operators; thus, TVM generates the kernel code, execution graph, and parameters. To run on the RISC-V simulator, a RISC-V executable must be compiled using the GNU toolchain with objects generated by TVM and our host code,

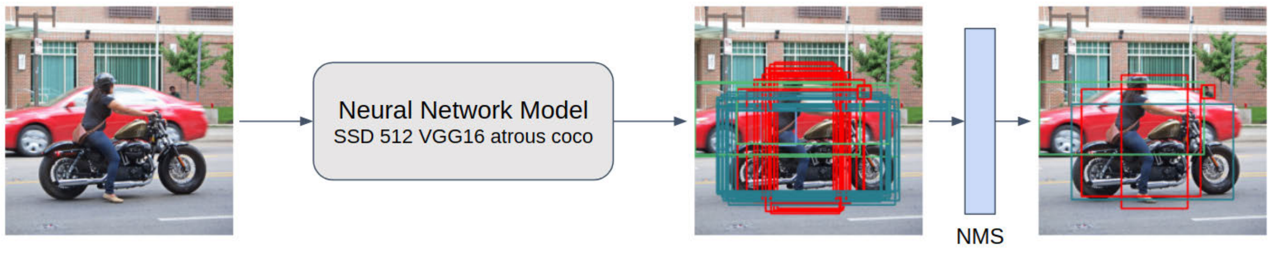


**FIGURE 8.** Object detection flow with the Single Shot MultiBox Detector (SSD) neural network model.

which is also the reason why we use LLVM tools to compile the kernel code into assembly code. Fig. 8 shows the object detection flow from the input image to the SSD model for object detection and finally to the NMS algorithm for postprocessing. We run this flow based on TVM.

### *B. EXPERIMENTAL RESULTS*

The Microsoft COCO [12] dataset is used to evaluate the performance and accuracy of the baseline and optimized NMS algorithms. There are 287k images in the COCO 2017 dataset, including 118k for model training, 41k for the testing trained model, 5k for validation, and 123k unlabeled images. We use the 5k validation images as the input data to evaluate the instruction count and accuracy Fig. 9 shows the experimental results for two example test images from the COCO dataset. Panels (a) to (d) correspond to the image with ID 281553, which contains five labeled objects, and panels (e) to (h) correspond to the image with ID 465404, which contains 54 labeled objects. Panels (a) and (e) show the original images, panels (b) and (f) show the ground-truth labeled data, panels (c) and (g) show the objects detected with the baseline version of NMS, and (d) and (h) show the objects detected with the optimized version of NMS.

We use the instruction count as a performance indicator. We measured the number of instructions required to execute NMS in Spike. Fig. 10 shows the speedup results for the 5,000 validation images. The results show that with our optimizations the performance is improved by factor of 1.28 to 15.29 on the COCO dataset. We find that when there are more objects in the image, the speedup will be greater; thus in Fig. 11, we also present the number of objects in comparison with the speedup results. We take images 255664 and 498919 from the COCO dataset as specific examples. For image 255664 which contains two objects, the baseline version requires 228,201,029 instructions to perform the NMS operations; in the optimized version, only 176,912,542 instructions are needed, corresponding to a speedup of 1.28. For comparison, in image 498919, there are 23 labeled objects; in this case, the baseline version requires 9,775,751,198 instructions, whereas the optimized version requires only 639,265,800 instructions, corresponding to a speedup of 15.29. This is mainly because in the NMS algorithm, if there are more objects, more loops must be executed, and the optimized version uses SIMD instructions to process multiple data elements simultaneously to improve performance. We further analyze the distributions of the instructions before and after optimization, as illustrated in Fig. 12 and Fig. 13. Before optimization, the top 10 most frequently occurring instructions account for 66% of the total; after optimization, our analysis reveals reductions ranging from 98.48% to 100% for these instructions, as shown in Table 2. Additionally, the top 10 SIMD instructions after optimization are presented in Table 3.

In the baseline version, the most frequently used instruction is "bne". This is because the NMS algorithm involves numerous conditional operations, including checking for a nonzero intersection between two bounding boxes before calculating the IoU and evaluating whether the computed IoU exceeds a specified threshold. In the optimized version,

**TABLE 2.** Instruction count reduction rate for the top ten instructions in the baseline version of the NMS algorithm.

| Instruction | Reduction Rate | Instruction Count |
|---|---|---|
| bne | 98.48% | 1379438049 |
| c.slli | 98.83% | 948394138 |
| flw | 100.00% | 917879562 |
| c.addi | 98.65% | 890761084 |
| c.srli | 99.42% | 877753370 |
| c.add | 100.00% | 755816377 |
| slli | 98.91% | 684818484 |
| xori | 100.00% | 673417622 |
| fle.s | 100.00% | 592662150 |
| mul | 98.31% | 549143792 |

**TABLE 3.** Top 10 subword SIMD instructions utilized in the optimized version of the NMS.

| Instruction | Instruction Count | Description |
|---|---|---|
| umin8 | 21,331,848 | unsigned minimum |
| plse8 | 18,593,968 | memory stride load |
| cmpeq8 | 10,790,904 | integer compare equal |
| scmplt8 | 11,290,824 | signed compare less than |
| psse8 | 1,499,760 | memory stride store |
| ksub8 | 758,128 | signed saturating subtraction |
| smin8 | 508,168 | signed minimum |
| smax8 | 508,168 | signed maximum |
| ksll8 | 508,168 | saturating shift left logical |
| khm8 | 254,084 | signed saturating Q7 multiply |

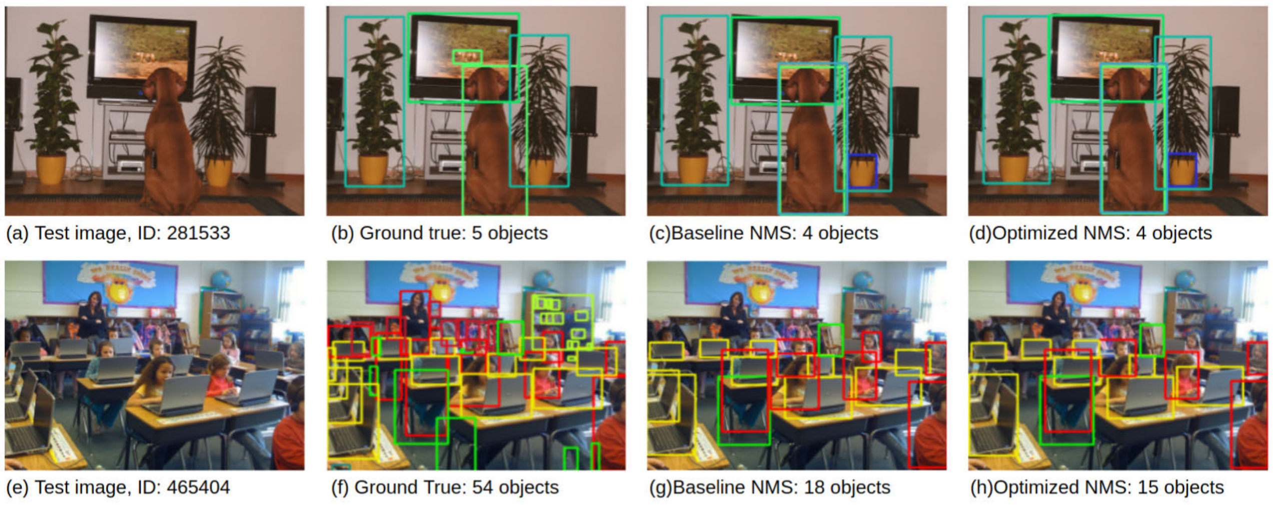

(a) Test image, ID: 281533
(b) Ground true: 5 objects
(c)Baseline NMS: 4 objects
(d)Optimized NMS: 4 objects
(e) Test image, ID: 465404
(f) Ground True: 54 objects
(g)Baseline NMS: 18 objects
(h)Optimized NMS: 15 objects

**FIGURE 9.** Test images from the COCO dataset and corresponding detection results. The colored boxes mark the locations of the detected objects. Upper row: test image 281533. Lower row: test image 465404.

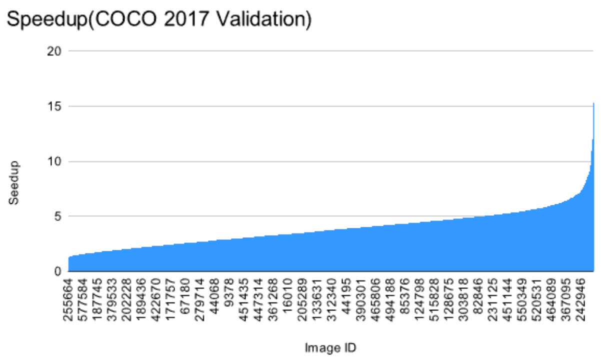


**FIGURE 10.** Results obtained when using the 5000 validation images in the Microsoft COCO 2017 dataset as test data to compare the numbers of instructions used in the baseline and optimized versions.

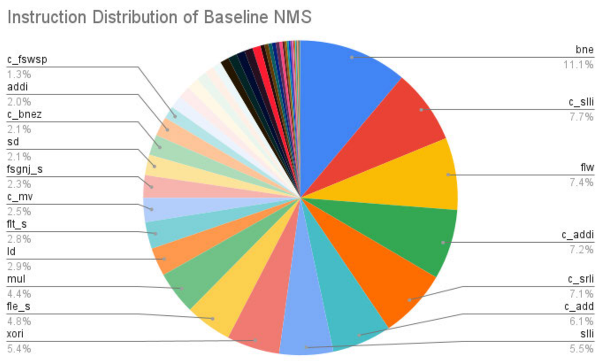


**FIGURE 12.** Statistics for the number of instructions used and executed in the baseline version of the NMS algorithm.

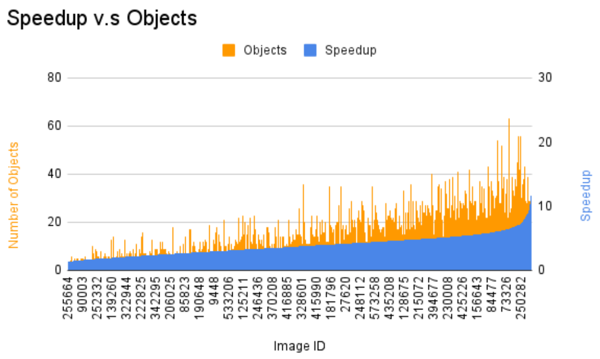


**FIGURE 11.** Results obtained when using 5000 validation images from the Microsoft COCO 2017 dataset as test data, to compare the numbers of instructions in the baseline and optimized versions. This figure shows the speedup and the number of objects for each test image in the Microsoft COCO dataset.

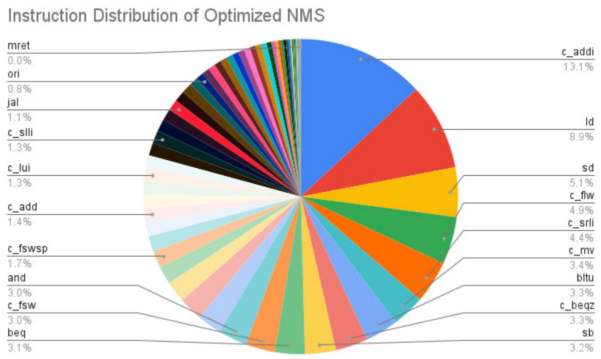


**FIGURE 13.** Statistics for the number of instructions used and executed in the optimized version of the NMS algorithm.

we employ the ternary operation based on rewriting strategy and replace "bne" with "CMPEQ8" and "BPICK". This modification not only enables simultaneous data comparisons through SIMD instructions but also helps avoid potential performance losses associated with branch misprediction. Reducing the instruction count is believed to offer various benefits, although empirical verification is challenging. It has been suggested that such optimizations may enhance

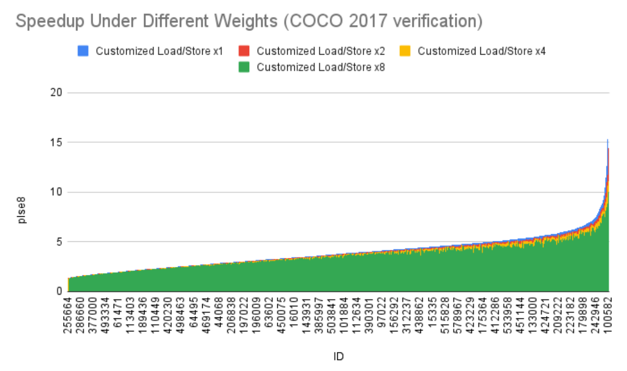


**FIGURE 14.** Comparison of the results obtained when multiplying the instruction counts for the custom load/store instructions by factors of 1, 2, 4, and 8 to analyze the potential speedup under different microarchitectures and memory hierarchies.

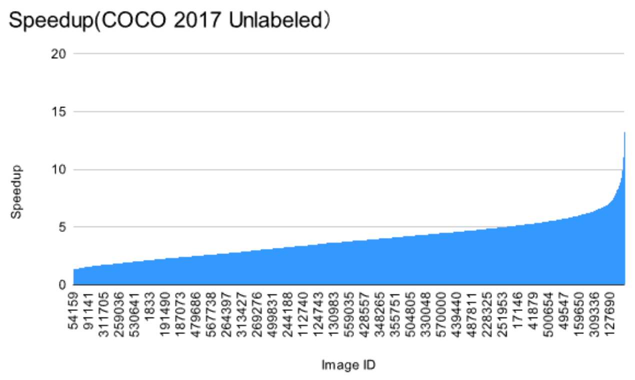


**FIGURE 15.** Results obtained when using 5000 images from the unlabeled category of the Microsoft COCO 2017 dataset as test data to compare the numbers of instructions used in the baseline and optimized versions.

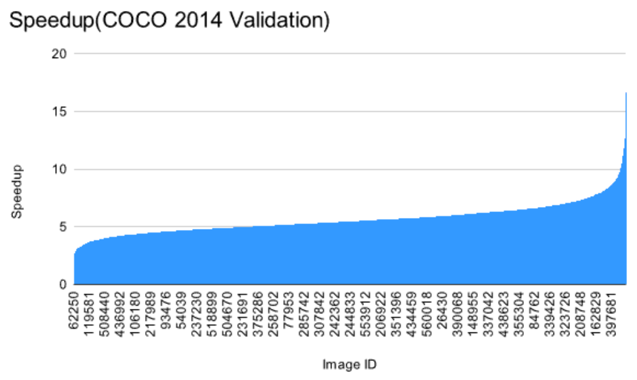


**FIGURE 16.** Results obtained when using 5000 images from the validation category of the Microsoft COCO 2014 dataset as test data to compare the numbers of instructions used in the baseline and optimized versions.

performance by streamlining program execution, potentially expediting task completion and improving overall system efficiency. Lowering CPU power consumption also seems plausible, especially in energy-sensitive environments such as mobile devices and embedded systems. The assumption is that fewer instructions may contribute to increased resource utilization efficiency by minimizing memory access and register usage demands. Another potential outcome could be more compact executable code, which might help address storage space limitations, particularly for systems with restricted memory. In addition to the mentioned advantages, our analysis reveals significant reductions in the number of load and store instructions, reducing the number of read and write operations to memory. A decrease in the quantity of these instructions alleviates pressure on the memory system, potentially resulting in faster access. This is crucial for enhancing the overall system efficiency, especially in applications where high efficiency is essential.

In addition to the number of instructions affecting performance, the cycle of memory load/store instructions may be influenced by differences in the microarchitecture and memory hierarchy. To further analyze the impact of various hardware implementations, we multiplied the instruction counts for the custom memory load/stores (plse8, psse8) by factors of 1, 2, 4, and 8 in the optimized version. We observed a speedup across all images, as shown in Fig. 14 The results indicate that even with the highest weight (8x), the speedup factor can still reach 12.219.

We additionally selected 5000 images from the unlabeled category in COCO 2017 and 5000 images from the validation category in the COCO 2014 dataset to validate the speedup results on different datasets, as illustrated in Fig. 15 and Fig. 16. Here, the performance improvement factors range from 1.27 to 13.2 and from 2.45 to 16.07, respectively.

We used the API provided for the Microsoft COCO dataset to obtain the ground-truth labels for the test images and calculate the mean average precision. Fig. 17 compares the results for the 5,000 test images between the baseline and optimized versions. In Fig. 17, IoU denotes the IoU threshold, and 0.50:0.95 denotes a range of IoU thresholds from 0.5 to 0.95 in increments of 0.05, corresponding to a total of 10 different IoU threshold; "area" is used to indicate the scales across which the average precision is calculated, with "small" scales corresponding to area $< 322$ pixels, "medium" scales corresponding to $322 <$ area $< 962$ pixels, and "large" scales corresponding to area $> 962$ pixels. The notation "maxDets" indicates the maximum number of detected objects. For example, the first column represents the average precision achieved across all 10 IoU thresholds and all different scales, where the maximum number of detected objects is 100. Our experimental results show that the optimized version exhibits a loss in average precision that ranges from 0.002(@[IoU=0.50:0.95 | area=small | maxDets=100]) to 0.03(@[IoU=0.50:0.95 | area=large | maxDets=100]). The main reason for this is that for the IoU computation, it is necessary to compute the areas of two bounding boxes, which requires multiple instructions, and the value range of int8 may not be sufficient to hold the area value, which may cause a box with a lower score to be suppressed when two objects of the same class overlap.

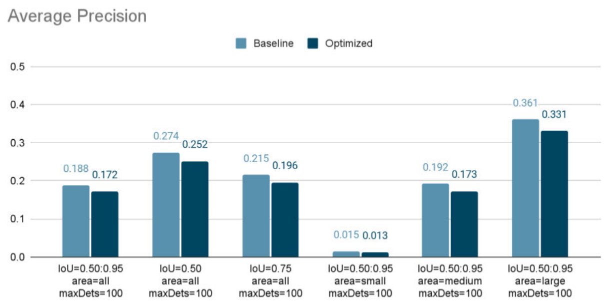


**FIGURE 17. Average precision of the baseline and optimized NMS algorithms on the Microsoft COCO 2017 dataset.**

## VII. RELATED WORKS AND DISCUSSION

The end-to-end deep learning flow includes preprocessing, model inference, and postprocessing steps. Previous research aiming to enhance the performance of preprocessing and postprocessing has focused on two aspects: software and hardware. Liang et al. [29] proposed a customized hardware accelerator specifically designed for deep learning algorithm preprocessing and postprocessing. However, while custom hardware can be effective at accelerating specific algorithms, its applicability is limited to optimizations tailored for those algorithms. Due to these constraints, achieving hardware acceleration across different algorithms is challenging. Hence, the present study proposes custom instructions that integrate software and hardware design aspects. This approach enables the potential of customized hardware to be exploited for various algorithms, thereby improving program execution performance. On the software side, Kim et al. [30] introduced task-level pipeline parallelism to maximize CPU and GPU resource utilization, consequently improving object detection performance. However, acceleration relying solely on software also has limitations. In our research, we introduce new instructions for calculations commonly used in algorithms and integrate software and hardware design to maximize performance.

Nonmaximum suppression (NMS) is widely used as a postprocessing technique for object detection models and computer vision tasks. The goal is to select the most representative targets from among overlapping candidate detection boxes to eliminate redundant detections. The NMS algorithm sorts candidate boxes based on their predicted confidence scores and gradually adds the highest-scoring boxes to the final detection results while removing any boxes with overlap above a specified threshold. However, NMS faces performance challenges, particularly in terms of computational cost. The computational expense of pairwise comparisons with many candidate boxes, and overlap calculations can significantly affect the processing speed. Additionally, determining an appropriate overlap threshold is crucial, as an excessively high or low threshold can result in missed targets or improper box selection. In the realm of NMS optimization, Bolya et al. [31] analyzed the operational characteristics of NMS and reported that the computation time can be effectively reduced by representing the intersection-over-union (IoU) operation between bounding boxes as a symmetric matrix and utilizing GPUs. However, this method sacrifices a small amount of accuracy due to redundant suppression. Building upon this concept, Zheng et al. [32] further modified their research to maintain the mean average precision(mAP). However, these methods primarily rely on GPU acceleration, which may not be suitable for low-power embedded systems.

In recent years, many applications have begun to leverage the advantages of RISC-V for optimization and acceleration in lower-power embedded systems. For example, in [33], the utilization of the P extension of RISC-V led to accelerated model execution based on TVM. The acceleration achieved through the P-extension enables faster inference computations. Reference [34] used the RISC-V vector extension to accelerate commonly used scan operations in recurrent neural networks (RNNs), which are crucial for processing sequential data in applications such as natural language processing and speech recognition. References [35] and [36] focused on accelerating computations for convolutional neural networks (CNNs) and the k-nearest neighbors (k-NN) algorithm, respectively, through the introduction of custom instructions. However, these research endeavors primarily concentrated on accelerating model inference without addressing acceleration during the preprocessing and postprocessing stages. In our research, we use NMS, a commonly used postprocessing step in object detection, as a case study. We design custom instructions specifically to accelerate NMS computations, contributing to the broader improvement of essential tasks in the object detection pipeline.

The choice of using the instruction count as a metric is constrained by the current limitations of our experimental environment. Currently, only Spike supports the simulation of RVP instructions. Future research could explore more accurate performance metrics, such as execution time, memory behavior, and power consumption analysis. Such exploration is contingent upon the future support of gem5 [37] or a RISC-V performance model (Olympia) [38]. Contingent on this support integrating these advanced simulation environments would enable a more comprehensive and precise analysis of our proposed optimizations.

## VIII. CONCLUSION

In this research, we propose optimization methods for TVM hybrid script, employing rewriting strategies and custom RISC-V instructions. Our experiments demonstrate a notable reduction in instruction count by a factor of approximately 1.28 to 15.29, with a minimal decrease in average precision (0.002 to 0.03). In the particular example of the NMS algorithm, the reduction in instructions correlates with the number of objects in the target image. Experiments were conducted on a single CPU core utilizing the RISC-V P extension with our custom instructions, oriented toward low-power embedded systems.

## REFERENCES

[1] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 11, pp. 3212–3232, Nov. 2019.

[2] M.-T. Chiu, H.-Y. Cheng, C.-Y. Wang, and S.-H. Lai, "High-accuracy RGB-D face recognition via segmentation-aware face depth estimation and mask-guided attention network," in *Proc. 16th IEEE Int. Conf. Autom. Face Gesture Recognit. (FG)*, Dec. 2021, pp. 1–8.

[3] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," 2017, *arXiv:1711.07128*.

[4] D. Seo, H.-S. Oh, and Y. Jung, "Wav2KWS: Transfer learning from speech representations for keyword spotting," *IEEE Access*, vol. 9, pp. 80682–80691, 2021.

[5] U. Kamath, J. Liu, and J. Whitaker, *Deep Learning for NLP and Speech Recognition*, 1st ed. Berlin, Germany: Springer, 2019.

[6] I. Fursov, A. Zaytsev, P. Burnyshev, E. Dmitrieva, N. Klyuchnikov, A. Kravchenko, E. Artemova, E. Komleva, and E. Burnaev, "A differentiable language model adversarial attack on text classifiers," *IEEE Access*, vol. 10, pp. 17966–17976, 2022.

[7] M. Abdelaty, R. Doriguzzi-Corin, and D. Siracusa, "DAICS: A deep learning solution for anomaly detection in industrial control systems," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 2, pp. 1117–1129, Apr. 2022.

[8] K.-J. Wang and Y.-C. Qiu, "A system deployment model of multi-CCD automatic optical inspection for economical operations," *IEEE Access*, vol. 10, pp. 58040–58049, 2022.

[9] Y. Deng, X. Pan, X. Wang, and X. Zhong, "Vison-based 3D shape measurement system for transparent microdefect characterization," *IEEE Access*, vol. 7, pp. 105721–105733, 2019.

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, Carlsbad, CA, USA, 2018, pp. 579–594.

[11] K.-H. Chen, C. Su, C. Hakert, S. Buschjäger, C.-L. Lee, J.-K. Lee, K. Morik, and J.-J. Chen, "Efficient realization of decision trees for real-time inference," *ACM Trans. Embedded Comput. Syst.*, vol. 21, no. 6, pp. 1–26, Nov. 2022.

[12] T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.* Zurich, Switzerland: Springer, 2014, pp. 740–755.

[13] T.-L. Chen, Y.-R. Chen, M.-S. Yu, and J.-K. Lee, "NNBlocks: A blockly framework for AI computing," *J. Supercomput.*, vol. 77, no. 8, pp. 8622–8652, Aug. 2021.

[14] M.-Y. Lai, C.-Y. Sung, J.-K. Lee, and M.-Y. Hung, "Enabling Android NNAPI flow for TVM runtime," in *Proc. 49th Int. Conf. Parallel Process.*, Edmonton, AB, Canada, Aug. 2020, pp. 1–8.

[15] Y. Wen, Q. Guo, Z. Du, J. Xu, Z. Zhang, X. Hu, W. Li, R. Zhang, C. Wang, X. Zhou, and T. Chen, "Enabling one-size-fits-all compilation optimization for inference across machine learning computers," *IEEE Trans. Comput.*, vol. 71, no. 9, pp. 2313–2326, Sep. 2022.

[16] X. Han and Y. Zhang, "Decomposition-coordination-based voltage control for high photovoltaic-penetrated distribution networks under cloud-edge collaborative architecture," *Int. Trans. Electr. Energy Syst.*, vol. 2022, pp. 1–20, Jan. 2022.

[17] L. Zheng and T. Chen, "Optimizing deep learning workloads on ARM GPU with TVM," in *Proc. ReQuEST*, Williamsburg, VA, USA, 2018, pp. 1–9.

[18] P. Y. Chang, T. L. Chen, Y. T. Huang, M. S. Yu, and J. K. Lee, "C++OpenCL4TVM: Support C++OpenCL kernel for TVM NN operators," in *Proc. IWOCL*, Bristol, U.K., 2022, pp. 1–2.

[19] T. Yang, X. Han, H. Li, W. Li, and A. Y. Zomaya, "Parallel scientific power calculations in cloud data center based on decomposition-coordination directed acyclic graph," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 2491–2502, Jul./Sep. 2023.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.

[21] E. Cui, T. Li, and Q. Wei, "RISC-V instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24696–24711, 2023.

[22] E. Cheshmikhani, B. Peccerillo, A. Mondelli, and S. Bartolini, "A general framework for accelerator management based on ISA extension," *IEEE Access*, vol. 10, pp. 120702–120713, 2022.

[23] C.-C. Lin, C.-L. Lee, J.-K. Lee, H. Wang, and M.-Y. Hung, "Accelerate binarized neural networks with processing-in-memory enabled by RISC-V custom instructions," in *Proc. 50th Int. Conf. Parallel Process. Workshop*, Lemont, IL, USA, Aug. 2021, pp. 1–8.

[24] H. Lin, P. Chen, Y.-S. Hwang, and J.-K. Lee, "Devise rust compiler optimizations on RISC-V architectures with SIMD instructions," in *Proc. 48th Int. Conf. Parallel Process.*, Kyoto, Japan, Aug. 2019, pp. 1–7.

[25] B. Marshall, D. Page, and T. Pham, "Implementing the draft RISC-V scalar cryptography extensions," in *Proc. Hardw. Architectural Support Secur. Privacy*, Virtual, Greece, Oct. 2020, pp. 1–8.

[26] S.-C. Wang, L.-Y. Yu, L.-A. Her, Y.-S. Hwang, and J.-K. Lee, "Pointer-based divergence analysis for OpenCL 2.0 programs," *ACM Trans. Parallel Comput.*, vol. 8, no. 4, pp. 1–23, Dec. 2021.

[27] S.-C. Wang, L.-C. Kan, C.-L. Lee, Y.-S. Hwang, and J.-K. Lee, "Architecture and compiler support for GPUs using energy-efficient affine register files," *ACM Trans. Design Autom. Electron. Syst.*, vol. 23, no. 2, pp. 1–25, Mar. 2018.

[28] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. 14th Eur. Conf.* Amsterdam, The Netherlands: Springer, Oct. 2016, pp. 21–37.

[29] S. Liang, X. Ning, J. Yu, K. Guo, T. Lu, C. Tang, S. Zeng, Y. Wang, D. Yang, and H. Yang, "Efficient computing platform design for autonomous driving systems," in *Proc. 26th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Tokyo, Japan, Jan. 2021, pp. 734–741.

[30] R. Kim, G. Kim, H. Kim, G. Yoon, and H. Yoo, "A method for optimizing deep learning object detection in edge computing," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Jeju, Korea (South), Oct. 2020, pp. 1164–1167.

[31] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, "YOLACT++ better real-time instance segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 2, pp. 1108–1121, Feb. 2022.

[32] Z. Zheng, P. Wang, D. Ren, W. Liu, R. Ye, Q. Hu, and W. Zuo, "Enhancing geometric factors in model learning and inference for object detection and instance segmentation," *IEEE Trans. Cybern.*, vol. 52, no. 8, pp. 8574–8586, Aug. 2022.

[33] Y.-R. Chen, H.-H. Liao, C.-H. Chang, C.-C. Lin, C.-L. Lee, Y.-M. Chang, C.-C. Yang, and J.-K. Lee, "Experiments and optimizations for TVM on RISC-V architectures with P extension," in *Proc. Int. Symp. VLSI Design, Autom. Test (VLSI-DAT)*, Aug. 2020, pp. 1–4.

[34] H.-M. Lai and J.-K. Lee, "Efficient support of the scan vector model for RISC-V vector extension," in *Proc. 51st Int. Conf. Parallel Process.* New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 1–8, doi: 10.1145/3547276.3548518.

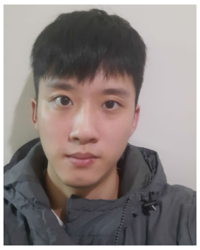
[35] N. Wu, T. Jiang, L. Zhang, F. Zhou, and F. Ge, "A reconfigurable convolutional neural network-accelerated coprocessor based on RISC-V instruction set," *Electronics*, vol. 9, no. 6, p. 1005, Jun. 2020.

[36] H. W. Oh and S. E. Lee, "The design of optimized RISC processor for edge artificial intelligence based on custom instruction set extension," *IEEE Access*, vol. 11, pp. 49409–49421, 2023.

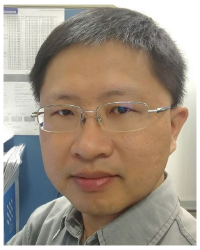
[37] *Gem5*. Accessed: Dec. 6, 2022. [Online]. Available: http://www.gem5.org

[38] *Risc-v Performance Model*. Accessed: Dec. 6, 2022. [Online]. Available: https://github.com/riscv-software-src/riscv-perf-model

**MENG-SHIUN YU** received the B.S. degree in electrical engineering from the National Chin-Yi University of Technology, and the M.S. degree in electrical engineering from National Chung Cheng University. He is currently pursuing the Ph.D. degree with the Department of Computer Science, National Tsinghua University, Taiwan. His research interests include compiler optimization for deep neural networks and computer vision and compiler construction for hardware accelerators.



**CHUAN-YUE YUAN** is currently pursuing the degree with the Department of Computer Science, National Tsing Hua University, Taiwan. His thesis advisor is Prof. Jenq-Kuen Lee. His research interests include compiler optimizations on RISC-V with SIMD computations, AI compiler optimizations, and compiler analysis for program reliability.



**TAI-LIANG CHEN** received the B.S. degree in information management from Shih Chien University, Taipei, Taiwan, in 2002, the M.S. degree in computer science and information engineering from Asia University, Taichung, Taiwan, in 2005, and the Ph.D. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2022. His research interests include visual programming language, parallel computing, and compiler design and optimizations.

**JENQ-KUEN LEE** received the M.S. and Ph.D. degrees in computer science from Indiana University, Bloomington, IN, USA, in 1991 and 1992, respectively. He is currently a Professor with the Department of Computer Science, National Tsing Hua University, Taiwan, which he joined, in 1992. He also leads the research team in the effort to develop AI compilers using RISC-V with SIMD computations. His current research interests include optimizing compilers, AI framework compilers, embedded multicore compilers and systems, and computer architectures. He received the Google Research Award in 2009. In addition, he was a recipient of Taiwan MOEA Economic Contribution Award (Deep Plow Award) in 2010. From 2015 to 2018, he participated in the new version of the OpenCL proposals with the Khronos OpenCL DSP Feature Set. He has given presentations on his research at the TVM Conference, in 2018, 2019, 2020, and 2021.