Enabling Automatic Compiler-Driven Vectorization of Transformers

Shreya Alladi
Computer Engineering Department
University of Murcia
Murcia, Spain
shreya.a.s@um.es

Alberto Ros
Computer Engineering Department
University of Murcia
Murcia, Spain
aros@ditec.um.es

Alexandra Jimborean
Computer Engineering Department
University of Murcia
Murcia, Spain
alexandra.jimborean@um.es

***Abstract*—Compiling neural networks and Transformers for edge devices faces significant challenges due to resource constraints and the reliance on manually optimized operations for performance among others. These limitations hinder the scalability and portability of neural networks on resource-constrained platforms, such as edge devices utilizing the RISC-V ecosystem. Addressing these issues, this paper introduces innovative techniques to overcome the inefficiencies of current compilation methods and reduce dependence on manual optimizations.**

This work proposes a novel compilation flow, ONNX-MLIR-LLVM (OML), which leverages MLIR and LLVM IR to enable automatic optimizations and generate stand-alone RISC-V binaries. Through comprehensive analysis, we identify key barriers preventing the auto-vectorizer from handling vectorization-friendly operators, particularly reduction operations and vectorization-unfriendly data layouts. We address these through a versatile MLIR reduction detection pass and a compile-time transpose pass, respectively.

Our automatic transformations (OML-vect) unlock the capabilities of the MLIR affine super-vectorizer, reducing reliance on manual vectorization. Evaluations on both x86 and RISC-V across eight neural networks and Transformer models demonstrate that automatic vectorization via OML-vect achieves, on average, 5% and 59% on x86 and RISC-V, respectively, compared to baseline (manually vectorized libraries), offering an efficient and portable solution for edge device deployments.

***Index Terms*—Compiler, neural networks, vectorization, reduction detection, data layout optimizations.**

I. Introduction

Artificial Intelligence (AI) has experienced a remarkable rise, driven by rapid advancements in algorithms, data available for training, and software frameworks. Particularly, Transformer architectures [68] have reshaped the AI domain. With attention mechanisms, Transformers serve as the foundation for state-of-the-art models such as BERT, VIT, and GPT [61]. This evolution has also influenced the need for specialized hardware accelerators, compiler optimizations, and neural network frameworks to address the growing computational requirements of AI workloads.

PyTorch [3], for instance, has introduced optimizations to enhance the most common layers (self-attention and multi-head-attention) using fused kernels, achieving up to a 2x speedup compared to PyTorch without these optimizations for inference on CPUs and GPUs [55]. These improvements, coupled with its extensive ecosystem and widespread adoption, have established PyTorch as a leading framework for neural network research and deployment [22]

PyTorch relies on hand-optimized kernels written in C or C++. These kernels leverage techniques such as SIMD vectorization, tiling, unroll and jam, and multi-threading to deliver high performance on supported hardware. While these optimizations ensure efficient execution, the Python interface can introduce overheads due to frequent library calls [65]. Additionally, PyTorch is primarily optimized for architectures such as Intel, ARM, and AMD CPUs and GPUs, resulting in limited or no support and no optimizations for emerging ISAs such as RISC-V. This creates challenges for developers and organizations seeking to utilize RISC-V's open-source and flexible ecosystem for AI applications. Although efforts have been made to port PyTorch to RISC-V, they have encountered challenges [14]. Essential libraries are not readily available for RISC-V (e.g., cpuinfo [16]), and substantial effort is required to port the (low-level) manual optimizations to RISC-V, leading to a considerable performance gap between x86 and RISC-V architectures [38]. To close this gap between established and emergent architectures, a static compiler flow with automated analysis and optimization is crucial for efficient inference of neural networks and Transformers.

One such attempt was made through the Intermediate Representation Execution Environment (IREE) compiler [26]. IREE consists of two components: IREE Turbine (formerly Shark Turbine) and the IREE runtime. Nevertheless, currently, the IREE runtime does not support generating executables for RISC-V 64-bit [25]. Furthermore, IREE does not expose the LLVM IR, which restricts the ability to create binaries using the upstream LLVM toolchain without the entire IREE runtime framework. Instead, it exposes higher-level dialects such as Torch and Linalg. However, the Multi-Level Intermediate Representation (MLIR) [27] code produced by IREE is incompatible with upstream MLIR due to the use of a custom dialect *Utils* [24]. This prevents the lowering of IREE-generated MLIR abstractions to LLVM IR and generating executables without the IREE runtime. Additionally, deploying the IREE runtime on edge devices is resource-intensive, requiring significant memory due to dependencies such as Python. TinyIREE [29] was proposed as an alternative lightweight machine learning execution environment designed

for embedded systems, small-footprint edge devices, and bare-metal platforms. Still, it cannot generate RISC-V 64 bit code.

Two other frameworks, Torch-MLIR [67] and ONNX-MLIR [28], transform high-level neural network representations, PyTorch and Open Neural Network Exchange (ONNX), respectively, to a compiler representation (MLIR).

Torch-MLIR, however, encounters difficulties in aligning its abstractions with the MLIR and LLVM ecosystems. These limitations include restricted support for just-in-time (JIT) compilation [66], and challenges in fully leveraging LLVM IR. Since Torch-MLIR and IREE have been designed independently, they lack end-to-end compatibility in converting neural network models into MLIR or LLVM IR abstractions and generating executable binaries for RISC-V architectures [25].

In contrast, ONNX-MLIR exposes LLVM Dialect and is compatible with the MLIR and LLVM ecosystems, but has its own limitations. By default, ONNX-MLIR doesn't support the RISC-V backend and hence cannot generate an executable for RISC-V CPUs [51].

Additionally, ONNX-MLIR does not utilize MLIR's Affine super-vectorizer but instead replaces critical layers such as matrix multiplication with manually vectorized implementations. While this approach achieves performance gains, it lacks flexibility and scalability. This decision stems from limitations in MLIR's reduction pass [44], which fails to recognize reduction patterns in matrix multiplication when using the complex operations such as math.fma(fused multiply and add operation from *Math* Dialect). This capability to deconstruct complex ops and detect reduction patterns is absent in the current MLIR infrastructure. The reduction pass is essential because the Affine super-vectorizer relies on identified reduction operations to safely and effectively apply vectorization strategies along the reduction dimension [45]. Additionally, matrix multiplication often exhibits non-contiguous memory access patterns, which are not auto-vectorized by the MLIR vectorization pass, as this is not beneficial, further hindering the usage of the MLIR vectorization pass.

Our proposal is a compilation flow ONNX-MLIR-LLVM (OML) for RISC-V, based on ONNX-MLIR, introduces practical innovations that significantly enhance auto-vectorization within MLIR which allows us to *automatically* optimize the neural networks both in MLIR and LLVM. This process entails progressively lowering the neural network models from frameworks such as PyTorch or ONNX into high-level MLIR dialects, such as the ONNX dialects, followed by (mid-level) MLIR dialects such as Affine [1]. These are then further lowered to the LLVM dialect, LLVM IR, and finally to assembly to produce executable binaries.

Such a hierarchical approach facilitates (multi-level) compiler-driven optimizations, both hardware-agnostic and tailored to the target platform. While higher-level dialects (such as Affine) enable general-purpose optimizations, lower-level transformations leverage LLVM's capabilities to achieve hardware-specific performance enhancements.

This work addresses state-of-the-art's limitations in compiling neural networks and Transformer models and generating an executable for RISC-V. Furthermore, through automatic code transformations, it enables automatic vectorization on these applications, in a bid to reduce the dependence on manually optimized operations.

In this paper, we make the following contributions:

- We propose a compilation flow ONNX-MLIR-LLVM (OML) that (1) exposes MLIR and LLVM IR to enable automatic optimizations and (2) generates stand-alone RISC-V binaries (Section IV. The entire framework, along with the ported libraries, will be made publicly available to encourage adoption and further research.
- We analyze the reasons why the auto-vectorizer cannot handle (apparently) vectorization-friendly operators, such as matmul, and identify two of its main challenges: (1) reduction operations are not automatically identified and (2) the data layout is not suitable for vectorization (Section III). We tackle these challenges one by one.
- Identifying reduction operations: We exhibit the limitations of the mainstream MLIR pass to identify reduction operations and how this prevents auto-vectorization. Next, we propose a more versatile, stand-alone MLIR reduction pass compatible with upstream MLIR, able to identify multiple reductions and build reduction maps, which are essential for the MLIR super-vectorizer (Section IV-C). We show that our pass identifies on average 2.5× more reductions than the mainstream MLIR pass (Section VI-A). Helps in MLIR interoperability (affine to vector transformation)
- Data layout transformations: We analyze the data layouts that prevent the auto-vectorizer from being applied due to the performance penalties of the vectorized code. In response, we write a stand-alone pass that transposes the vectorization-unfriendly matrices at compile time, thus exposing contiguous memory locations (Section IV-A).
- Finally, we demonstrate how our automatic transformations to prepare the code for vectorization (OML-vect) effectively enables the mainstream MLIR super-vectorizer (Section VI-B).
- We demonstrate the benefits of our approach both on x86 and RISC-V on a set of eight neural networks and Transformer models and show that automatic vectorization enabled by our passes outperforms the manually vectorized libraries. We achieve on-par performance compared to ONNX-MLIR manually vectorized libraries (92% compared to ONNX-MLIR without manual optimizations). On RISC-V we surpass the performance of the ONNX Runtime for 4 out of 8 models. ONNX Runtime is highly optimized manually for x86, but its performance significantly falls behind on RISC-V, emphasizing the need for automatic optimizations (Section VI-B).

II. Background

This section provides a description of auto-vectorization in MLIR as well as an overview of ONNX and ONNX-MLIR.

A. Auto-Vectorization in MLIR

Auto-vectorization is an optimization technique that automatically converts scalar operations in loops into vector operations, enabling parallel execution. This process is key to improving performance, especially for loop-heavy tasks such as those found in neural networks.

In MLIR, auto-vectorization is facilitated through passes such as the affine super-vectorizer, accompanied by helper passes such as *the reduction pass*. These passes analyze loop nests, identify opportunities for vectorization, and transform them into efficient vector operations, maximizing hardware performance, and reducing execution time.

1) Reduction Identification: Reductions are essential operations used to combine values across loop iterations, such as summing elements in an array or finding the maximum value in a dataset. Efficiently handling reductions is crucial for achieving optimal performance, particularly in loop-intensive programs. In MLIR, a reduction pass helps identify and optimize these operations within loop nests.

To this end, it detects generic reductions within loop nests by examining a list of iteration-carried arguments (*iterCarriedArgs*) and the position of the potential reduction argument (*redPos*). If a reduction is detected, the utility returns the reduced value and a topologically sorted list of combiner operations involved in the reduction. If no reduction is found, the utility returns a null value.

The matching algorithm relies on several key invariants: the first combiner operation must involve both the iteration-carried value and the reduced value as operands; the iteration-carried value and combiner operations must be side-effect-free, with a single result and use; and the combiner operations must be nested within the code region (operation) performing the reduction. In addition, the reduction def-use chain must end with a terminator operation that yields the next iteration or output values in the same order as the iteration-carried values. Furthermore, *iterCarriedArgs* must include all iteration-carried values of the operation that performs the reduction.

Identified reduction operations are usually used to generate reductionMaps and will be sent to the Affine loop vectorizer to vectorize along the reduction dimension.

2) Affine Loop Vectorizer: The process of vectorizing affine loops begins by identifying super-vectorization patterns in nested affine loops, defined by specific annotations like reduction, or vectorization, and contiguous memory accesses. These patterns are located within the AffineForOp tree structure.

Once detected, the algorithm evaluates the potential for vectorization by assessing performance gains. If profitable, the vectorizer iteratively rewrites loops and operations in topological order. Scalar operations (e.g., affine load/store, constants, and uniform operands) are converted into vectorized forms, and the loop is coarsened with the operations inside transformed into vectorized versions.

If vectorization succeeds, the scalar loop is replaced with the vectorized version in the intermediate representation (IR). If it fails, the scalar loop remains. The algorithm then continues to check and vectorize subsequent patterns while avoiding interference from previously vectorized loops.

Input: Matrix A of size $m \times k$, Matrix B of size $k \times n$
Output: Matrix C of size $m \times n$ where $C = AB$

for $i = 0$ to m:
  for $j = 0$ to n:
    $C[i][j] = 0$
    for $l = 0$ to k:
      $C[i][j] += A[i][l] \cdot B[l][j]$

Fig. 1: Standard Matrix Multiplication (`MatMul`)

B. ONNX Graph

ONNX is an open-source format for artificial intelligence models, including deep learning and traditional machine learning. ONNX aims to provide a common language that any machine learning framework can use to describe its models. ONNX implements a Python runtime that can be used to evaluate ONNX models and to evaluate ONNX operations. This is intended to clarify the semantics of ONNX and help to debug ONNX tools and converters. It is not intended to be used for production, and performance is not a goal.

C. ONNX-MLIR

ONNX-MLIR is an open-source compiler that generates optimized inference code from ONNX models. Built on the MLIR infrastructure in LLVM, it uses a structured lowering pipeline: the ONNX graph is first converted to the ONNX dialect and then lowered to the Krnl dialect for loop-level optimizations.

At the Krnl stage, operations like MatMul and GEMM are optimized using manual libraries with techniques such as tiling, unroll and jam, and vectorization. The code is then lowered to the Affine and LLVM dialects.

The final LLVM dialect is compiled into a shared library, which, combined with runtime libraries and a main program, enables efficient inference across diverse platforms.

1) MatMul: A basic code of `MatMul` is provided in Figure 1. The innermost loop of `MatMul` consists of a combiner or reduction operation of addition. This reduction operation is often vectorized by accumulating partial sums across vector lanes, which are then reduced to a final scalar result. The problem is that B is iterated in columns, and hence it cannot be vectorized by MLIR. To solve this, ONNX-MLIR has implemented a different algorithm to improve vectorization benefits.

The MatMul algorithm implemented in ONNX-MLIR works differently, as illustrated in Figure 2. Assuming a matrix multiplication $C = AB$, for each element of matrix A, the corresponding row of matrix B is selected. Specifically, for element A[i][k], the k-th row of matrix B is chosen, and the value of A[i][k] is multiplied by each element in the k-th row of B. This multiplication is carried out using

Input: Matrix A of size $m \times k$, Matrix B of size $k \times n$
Output: Matrix C of size $m \times n$ where $C = AB$
VL = Vector Length

for $i = 0$ to m:
  for $j = 0$ to n with step size VL:
    $C[i][j : j + \text{VL} - 1] = 0$
    for $l = 0$ to k:
      $A\_brd = vector\_\text{broadcast}(A[i][l], \text{VL})$
      $C[i][j : j + \text{VL} - 1]$ += $A\_brd \circ B[l][j : j + \text{VL} - 1]$

Fig. 2: Vectorized ONNX-MLIR `MatMul`

vectorized operations to enhance performance. Unlike basic MatMul where B is accessed column-wise (see Figure 1). This algorithm accesses matrix B row-wise, it ensures that matrix B data accesses are consecutive and thus can be vectorized better. The resulting products are accumulated and stored in the corresponding row (i-th row) of matrix C. This process is repeated iteratively, adding the partial sums to compute the final product of A and B. The algorithm continues until all elements are processed and the complete matrix product C is obtained. In addition, ONNX-MLIR performs a 2D unrolling pattern for full tiles. 2D unrolling is key to achieve significant performance improvements [11].

As partial values of the result matrix C are computed in each iteration, this algorithm requires loading and storing the i-th row of matrix C multiple times. This repeated access leads to an increase in the number of vector load and store operations.

III. Motivation

The Transformer architecture, particularly the attention mechanism, has become a foundational component in generative AI and various other domains. The attention mechanism computes attention weights to quantify the relative importance of different elements within the input, which are then used to modulate the contribution of each input token to the output.

Matrix multiplication (MatMul) operations account for a substantial portion of the execution time in both the attention layers and overall Transformer models. For example, in the BERT model, approximately 97% of the execution time in the attention layer is spent on MatMul operations. This pattern holds across BERT-based Transformer models, where MatMul operations typically contribute to around 95% of the total execution time. In addition, other reduction operations such as `maximum` are also widely used, particularly in functions such as `SoftMax` [68]. To auto-vectorize this mechanism in MLIR, reduction operations are identified and passed to the auto-vectorizer, which then vectorizes the loop along the reduction dimension [45].

A. Challenges with Reduction Passes in MLIR

The MLIR framework `match-reduction-test` [44, 46] a reduction pass designed to identify simple reduction patterns, such as addition (*addf*) operations from the *arith* dialect. However, it fails to recognize more complex reduction patterns, such as reduction operations from the *math* dialect, e.g. fused multiply-add (FMA), maximum, or minimum operations. In certain scenarios, it may even fail to identify *addf* operations [43]. Table I summarizes the reduction operations commonly found in Transformer models, indicating whether the MLIR reduction pass is capable of identifying or fails to recognize them. In addition, the reduction pass operates at the function level and produces results that cannot be directly passed to the affine super-vectorizer. However, all of these reduction operations are candidates to be identified by the OML-reduction.

TABLE I: Ability of MLIR to identify common reduction operations in Transformers

Kernel	Dialect	Reduction op.	Identified	Candidates
MatMul (float)	math	fma	No	Yes
MatMul (float)	arith	addf	Yes	Yes
MatMul (int)	arith	addi	Yes	Yes
SoftMax	arith	maxnumf	No	Yes

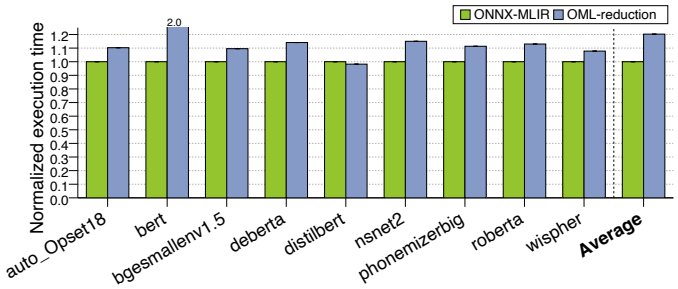


Fig. 3: Performance degradation on on Intel Xeon (x86) due to forceful vectorization when data access is non-consecutive

Our goal: Identify reduction operations across multiple dialects and build a comprehensive reduction loop map [39]. This map could be directly used by the Affine Super-Vectorizer in MLIR to enable a more effective and extensive vectorization of reduction operations.

B. Data Layout vs Auto-Vectorization Cost Model

Vectorization is extensively studied, but auto vectorization still lacks capability to match manually vectorized libraries. One of the key challenge for auto vectorization is non-contiguous memory accesses and data layout transformation. Vectorization is beneficial when the memory accesses are contiguous, while non-contiguous memory accesses can lead to performance degradation. Figure 1 illustrates a basic pseudocode for MatMul, clearly showing that matrix A has contiguous memory accesses, in contrast to matrix B. MLIR's *vectorizeLoops* [45] does not vectorize MatMul due to its data access patterns (see Section II-A2). Forcing vectorization against the indication given by the cost model leads to performance degradation as shown in Figure 3. Prior works [56, 31] have explored runtime data layout optimization, they come with runtime overheads (see subsection VII-D

Our goal: Convert to a more vectorization-friendly data layout at compile-time, ensuring that memory accesses target contiguous data.

C. ONNX-MLIR Vectorization

Section II illustrates the current approach in ONNX-MLIR for vectorizing MatMul operations. The decision to employ manual vectorization libraries arises from MLIR's reduction pass inability of identifying a broad range of reduction operations, as well as the inability of the affine vectorizer to vectorize MatMul operations due to their data access patterns. Although the ONNX-MLIR's manual optimizations ensure consecutive memory accesses for both input matrix A and B, it significantly increases the number of loads, stores (see Section II-C1). For a matrix multiplication with A of size $1\times8\times768$ multiplied by B of size 768×768 (a common case found in the Bert model [8]), this approach, vector loads account for 3.53% of total loads and vector stores are 54.72% of total stores[1]. Consequently, increasing the number of loads and stores can lead to memory bottlenecks, weakening performance due to excessive data transfers. Optimizing memory accesses is crucial to improve computational efficiency.

Our goal: Accumulate the partial results and write the final sum to the output matrix C only once to reduce memory operations as much as possible.

D. Putting It All Together

Given these challenges, the process of improving auto-vectorization can be broken down into four key steps:

1) Apply data layout changes (e.g., by statically transposing matrices) to yield memory accesses to contiguous data.
2) Update the MatMul algorithm to accumulate the partial results and write the final sum to the output matrix only once.
3) Detect reductions.
4) Apply vectorization.

Although data layout changes can be performed either before or after detecting reductions, it is generally more efficient to carry out this step at the highest abstraction level. In Section IV, we discuss our proposal in detail.

IV. OML-Vect Compilation Flow

We propose two multi-stage compiler pipelines named ONNX-MLIR-LLVM (OML) and ONNX-MLIR-LLVM-vect (OML-vect), shown in Figure 4 on the bottom, with blue and purple boxes, respectively. Unmodified sections of the pipeline are shown in gray. Furthermore, our pipeline leverages MLIR to implement compiler passes highlighted in pink, such as a reduction pass and a data preparation pass, to enhance auto-vectorization and improve performance.

The pipeline begins by exporting the pre-trained neural network to ONNX graphs. The ONNX graph is then traversed, and its data layouts are modified to optimize the code for auto-vectorization. Subsequently, the graph is lowered to the ONNX dialect, followed by the krnl dialect, and finally to the affine dialect using ONNX-MLIR. At the affine dialect level, reduction patterns are identified and the ***Affine loop-vectorizer*** is invoked. The vectorized code is then progressively lowered to SCF, CF, and LLVM dialects, ultimately generating LLVM IR, assembly, and eventually standalone binaries are generated.

We present the proposed OML-vect flow in detail. We begin by discussing data layout modification in Section IV-A, highlighting the efficiency of modifying the data layout (compile-time) at the ONNX graph level. Next, we cover the lowering of the ONNX graph to the Affine dialect in Section IV-B, followed by the reduction pass in Section IV-C, and discuss the process of generating executable binaries in Section IV-D.

A. Data Layout Transformations

We traverse the ONNX graph bottom-up to identify operations that cannot be auto-vectorized due to vectorization-unfriendly data access patterns. One such example is matrix multiplication MatMul ($A\times B$) (see Section III-B). At compile-time, to convert the data layout into one suitable for vectorization, we transpose the second matrix in the multiplication (B matrix).

Matmuls in Transformers can be classified into two types based on the second matrix's role, such as whether it is available at compile time or not: Weighted MatMuls and Generic MatMuls.

1) Weighted MatMuls: For MatMuls with embedded weights available at compile time (see Figure 5a MatMul (1)), we transpose the second input, the B matrix (which represents the weight matrix in this case) at compile time and annotate the node with metadata by setting *transB* to **1** indicating the transposed weight matrix [50] (see Figure 5b MatMul (1)). OML-Vect does not alter ONNX-graph semantics. We leverage the flags *transA* and *transB [23]* in the ONNX ecosystem, without introducing any new attributes. Since the weight matrix is used only within this kernel, the original matrix is not retained, avoiding data duplication and reducing the memory footprint.

2) Generic Matmuls: For non-weighted MatMuls, where both A (first input) and B (second input, which is not the weight matrix in this case) matrices have runtime inputs unavailable at compile time (Figure 5a MatMul (2)), a different optimization strategy is required. In the Transformer attention layers, non-weighted MatMuls (or Generic Matmuls) typically involve a transpose node providing input to the MatMul [68], particularly for the B matrix (the second input). This characteristic can be leveraged to optimize the transpose operation without introducing additional runtime overhead.

To achieve this, we traverse the ONNX graph to locate the B input for the matrix multiplication (MatMul) operation (see Figure 5a, green box). The traversal stops when a non-scalar operation, such as a transpose, is encountered [2]. After identifying the transpose node, we can introduce a second transpose operation to ensure the "transposed B" [3] is used as input to the MatMul. But as per the "Composition of Two

[1]Statistics obtained with Spike[58], a RISC-V ISA simulator

[2]If the node is not a transpose operation, it is skipped, as these cases fall outside the scope of our use cases (all Transformers)

[3]The matrix B is not a 2D matrix; the most common ones are 4D matrices, which are transposed across any two dimensions. Therefore, Transpose(Transpose(B)) is not the same as the original B matrix.

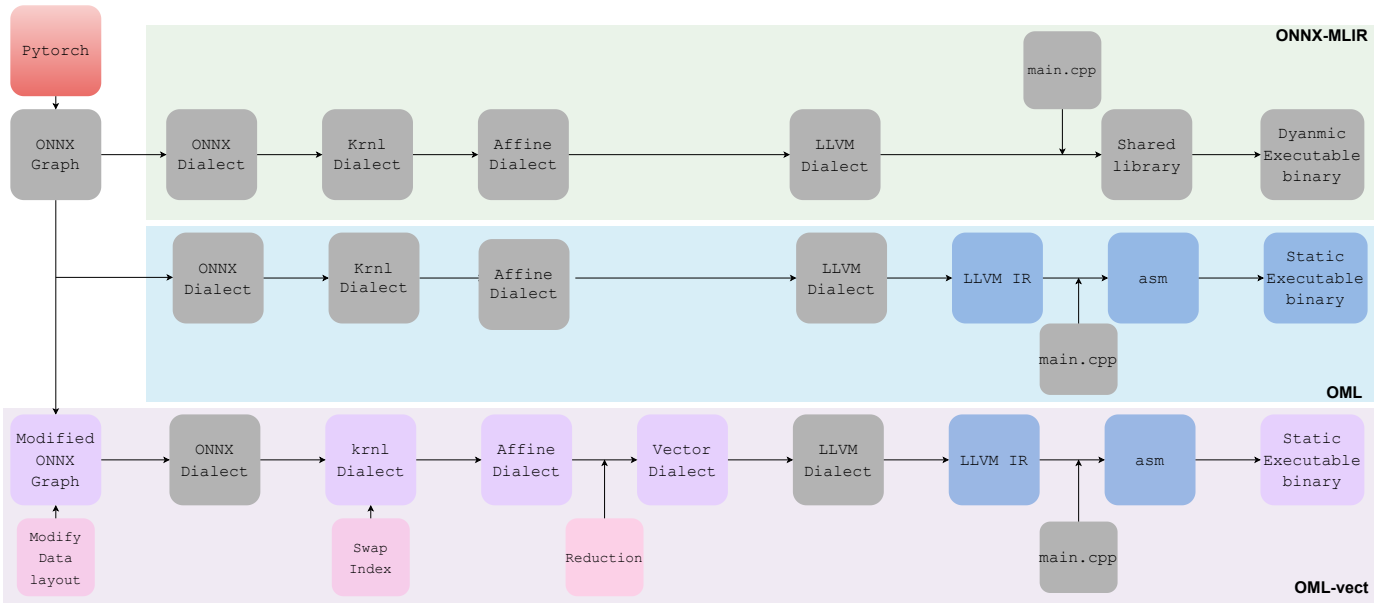
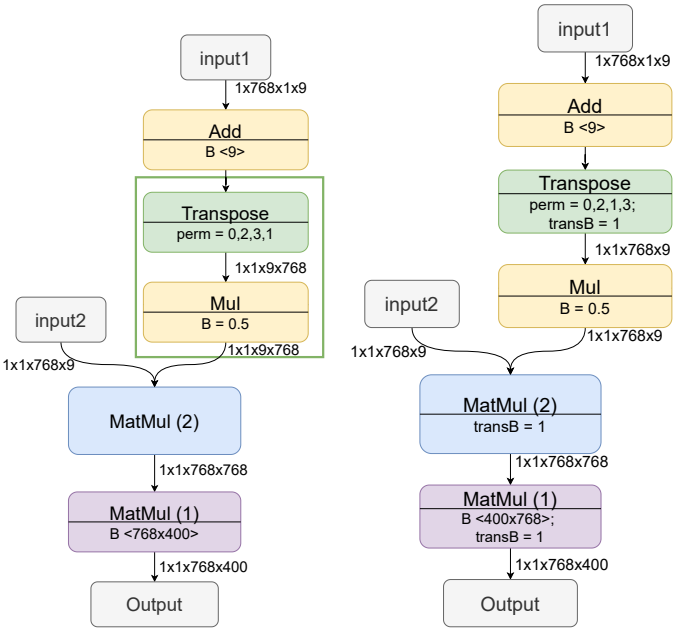


Fig. 4: Overview of ONNX-MLIR (prior work), OML, and OML-vect (our proposals) compilation flow



(a) Before (b) After

Fig. 5: ONNX graph before and after data layout optimization

Transpose Operations," [4] we do not need two separate transpose operations. Instead, we can adjust the permutation of the existing transpose node. Algoirthm 1 shows the algorithm to combine for Composition of Two Transpose Operations. This adjustment allows us to eliminate the need for an additional transpose, directly optimizing the MatMul input as shown in an example below.

Example: Given two transpose operations:

- First transpose: $\text{perm}_1 = [0, 2, 3, 1]$ (perm for Transpose node in Figure 5a)
- Second transpose: $\text{perm}_2 = [0, 1, 3, 2]$ (perm to get B_transpose)

Algorithm 1 Algorithm to combine two transpose operations

1) Get perm from the Transpose node.
2) Compute perm_transpose to swap the last two dimensions of the output from step 1.
3) For $i = 0$ to len(perm_transpose) $- 1$:
   compute combined_perm$[i]$ = perm1[perm2$[i]$].
4) Update the perm of the transpose node to the combined_perm.

Instead of applying both transposes separately, we combine them into a single permutation:

$$\text{combined}[i] = \text{perm}_1[\text{perm}_2[i]] \quad \text{for} \quad i = 0, 1, 2, 3$$

This results in the combined permutation:

$$\text{combined} = [0, 2, 1, 3]$$

Thus, permutation attributes are updated accordingly, and metadata is added to the MatMul node to indicate the presence of the transposed B input (see Figure 5b MatMul (2)). As before, we verify that the transpose node is exclusively used by the current MatMul, thus applying this operation in-place has no undesired side effects.

Modifying the data layout of the second matrix (B) impacts the matrix multiplication algorithm. These changes are discussed in Section IV-B, and our updated matrix multiplication algorithm is illustrated in Figure 6.

Unlike runtime-based transpose operations, such as *memref.transpose*, which introduce additional runtime instructions, overherad and additional memory consumption [31, 57, 35], our approach performs these optimizations at compile time. This eliminates runtime overhead and runtime data copying while paving the road to enable auto-vectorization.

Input: Matrix A of size $m \times k$, Matrix B of size $k \times n$ and $B\_Transposed = Tranpose(B)$
Output: Matrix C of size $m \times n$ where $C = AB$

if $transB = 1$:
for $i = 1$ to m:
  for $j = 1$ to n:
    $C[i][j] = 0$
    for $l = 1$ to k:
      $C[i][j] += A[i][l] \cdot B\_Transposed[j][l]$
else:
for $i = 1$ to m:
  for $j = 1$ to n:
    $C[i][j] = 0$
    for $l = 1$ to k:
      $C[i][j] += A[i][l] \cdot B[l][j]$

Fig. 6: OML-vect Matrix Multiplication (`MatMul`)

B. ONNX to Affine Dialect

Once the data layout has been changed by compile-time transposing the input matrix, the modified ONNX graph is passed to ONNX-MLIR. Initially, the ONNX graph is lowered to the ONNX dialect and then to the Krnl dialect flowing ONNX-MLIR flow. The transpose-metadata flag dictates how the matrix multiplication is performed. If set, OML-vect swaps the indices of the B matrix during MatMul as shown in Figure 6; if the flag is not set, the standard MatMul algorithm is employed as shown in Figure 1. Furthermore, based on the data type of the operations, the code generation adapts the MatMul logic: if the operations involve floating-point values, fused multiply-add (FMA) instructions are used; otherwise, for integer computations we retain the ONNX-MLIR algorithm, the MatMul is lowered using multiplication (muli) followed by addition (addi). Finally, the Krnl dialect is lowered to the affine dialect. Affine dialect servers as a structured intermediate representation where transformations such as vectorization are applied to further optimize the generated code.

C. Reduction Pass

The affine dialect represents an abstraction level higher than LLVM IR but lower than linalg, making it suitable for vectorization. The affine dialect provides auto-vectorization functions, which we utilize to enhance performance. The affine super vectorizer relies on LoopReduction [33], which internally uses arith::AtomicRMWKind. LoopReduction is a shared component used by both the affine loop vectorizer and loop parallelism passes in MLIR. The process begins by identifying the innermost loop, detecting potential reductions, creating Loop reduction maps, and invoking the auto-vectorizer.

The reduction pass starts by locating the innermost for loop in the affine representation. We focus on loops that include *iter_args* [40], which are values updated during each iteration and contribute to the final *affine.yield* [40] operation. These *iter_args* are updated using combiner operations such as *addf*, *fmul*, *maxnumf* [41], etc., which are subsets of the arith dialect and are mapped to `AtomicRMWKind` [42] operations known to represent reduction patterns.

If a complex operation consumes *iter_args* and is not directly mapped to `AtomicRMWKind`, we verify whether it can be decomposed into operations that belong to `AtomicRMWKind`. For instance, *math.fma* (fused multiply-add from math dialect) can be split into *mulf* and *addf*, where *addf* serves as a combiner/ reduction operation and is included in `AtomicRMWKind`. This enables our pass to identify complex reduction operations that are not detected by the MLIR reduction pass.

Algorithm 2 Algorithm for identifying reductions

1) Identify the innermost affine loop in the loop nest.
2) Get the `iter_args` (values updated during each iteration).
3) Get the users of `iter_args` in the loop.
4) Decompose any complex operation that consumes `iter_args` into an instance of AtomicRMWKind if possible.
5) Check if the detected operations maps to AtomicRMWKind.
6) Analyze the input-output shape relationship to verify reduction.
7) Create the `reductionMap` using the parent operation, combiner operation, and associated `iter_args`.

To detect reductions, we also analyze the relationship between input and output shapes. For example, an input tensor with a shape of $\langle$**1×3×f32**$\rangle$ reducing to a scalar **f32** indicates a reduction. Using the parent operation, the specific combiner, and the associated *iter_args*, we construct a *reductionMap* that encapsulates the reduction behavior. Algorithm 2 offers an overview of the reduction algorithm. Finally, the affine super vectorizer is invoked to optimize the identified reduction. The vectorized code is lowered to the LLVM dialect.

D. Generating Executables for RISC-V

The LLVM dialect is a wrapper class for LLVM IR, mapping LLVM IR within MLIR by defining the corresponding operations and types. ONNX-MLIR does not directly expose LLVM IR (see Green box Figure 4), which limits the ability to create executable binaries using the LLVM project [32]. OML and OML-vect address this limitation by translating the LLVM dialect to obtain LLVM IR, enabling the generation of standalone executable binaries. The obtained LLVM IR acts as an entry point to the LLVM compiler, facilitating automatic optimizations and the generation of assembly code through the robust LLC back-end. The assembly code is then processed by a native assembler and linker to produce a native executable.

Although LLVM supports RISC-V, as ONNX-MLIR does not expose LLVM-IR by default, it is not possible to generate ELF for RISC-V out-of-the-box. Exposing LLVM-IR is particularly important for the RISC-V target, as it enables

generating statically linked binaries via the LLVM toolchain. We proposed the OML pipeline to explicitly expose LLVM-IR and use the LLVM toolchain to lower LLVM-IR to statically linked binaries for RISC-V. Thus, we compiled ONNX-MLIR Runtime-libraries to RISC-V, porting manually libraries (OInstrument, OMRandomNormal, OMTensor) and fixed well-known porting issues [51].

Currently, ONNX-MLIR lacks built-in support for the RISC-V ISA [51]. To address this, we statically compile ONNX-MLIR runtime libraries, such as OMInstrument.c, OMRandomNormal.c, and OMTensor.c, to generate RISC-V-compatible runtime libraries. [4] The generated RISC-V runtime libraries, along with the assembly code and the main file (used for neural network inference), are linked using clang (or gcc) to produce an executable optimized for the RISC-V architecture. This approach generates statically linked binaries for neural networks, eliminating the runtime overhead associated with Python and other shared libraries [52]. Unlike ONNX-MLIR, which generates only shared library files and requires additional memory, our approach is more efficient.

E. Discussion

Although transformer models can accept dynamic input-shapes, the weights (second-input matrix) remain static and known at compile-time, as they are fixed after training [34]. In the case of weighted-MatMuls, we modify the data-layout of the weights only (see section IV-A1, since they are compile-time constants). For generic-MatMul operations, in self-attention [62] and cross-attention [17] layers, where a transpose node precedes the second input matrix, we optimize such patterns by permuting the transpose attributes (see section IV-A2). There is no need for a runtime conditional transpose in these cases, but runtime-aware layout handling may be needed for the first-input matrix, which has dynamic shapes. Our focus is on compile-time data-layout optimizations, and such runtime considerations fall outside of our scope.

V. Methodology

To assess the effectiveness of our OML and OML-vect pipelines, we perform a comprehensive evaluation using seven Transformer models and one recurrent neural network model. The training and testing datasets for these models are described in their respective references. The recurrent neural network model is nsNet2 [13, 37]. The eight Transformer models are auto_Opset18, bert_google [9], bgesmallenv [10], deberta [18], distilbert [19], phonemizerbig [54], and roberta [59].

The models are evaluated on two platforms: an x86_64 Intel(R) Xeon(R) E5-2630 v4 CPU @ 2.20GHz and a Xilinx U55C FPGA emulating[5] an Atrevido 423 RISC-V 64-bit core with a 512-bit vector unit [6]. Since we use FPGA emulation to verify RISC-V results, running full Transformer models is time-consuming. Thus, we decided to evaluate only the attention layers on both platforms. The evaluation measures the impact of our proposed optimizations on both x86 and RISC-V architectures.

[4]Since these libraries were not initially prepared for RISC-V, the porting process required manually resolving compile-time errors [51].

[5]The FPGA emulates a production core(Atrevido), being 100% cycle accurate [63]

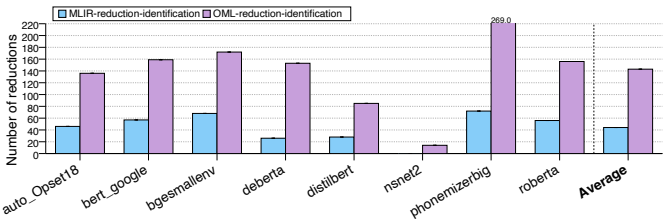


Fig. 7: Number of reductions identified

The evaluation process consists of three stages:

1) **Reduction Pass Evaluation:** We compare the number of reductions identified by our OML reduction pass, with those identified by the reduction pass "match reduction test" in MLIR.
2) **Performance analysis on x86 and RISC-V:** We analyze the performance of attention layers (selecting the most common attention layer for Transformer models and the most common MatMul + GRU layer for nsNet2) under five configurations: *(a)* ONNX-MLIR-no-custom-opts – ONNX-MLIR with O3 optimization level and manual optimizations disabled; *(b)* OML – ONNX-MLIR-LLVM compilation flow using the OML flow, without our reduction identification pass and without our data layout optimization. It uses the default MLIR's reduction pass *test-match-reduction-pass* and runs the MLIR auto-vectorizer and LLVM's auto vectorization pass at OPT O3 pass including SLP vectorization, Loop vectorization, and vplan [7] and LLC O3 pass; *(c)* ONNX-MLIR-default (with manual opts) as our baseline – ONNX-MLIR with O3 optimization level and manual optimizations enabled; *(d)* OML-vect – OML with our OML-reduction pass and with our data layout optimization pass using MLIR auto-vectorization. This configuration shows the cumulative benefits of our techniques when preparing the code for a more effective and efficient auto-vectorization. and *(e)* ORT – ONNX Runtime [49] with hand optimized C/C++ hardware-specific libraries;
3) **Other metrics:** We analyze the instruction count, binary size, and memory footprint and compilation time of the models compiled with OML-vect versus ONNX-MLIR-default (manual).

VI. Evaluation

This section evaluates the impact of OML and OML-vect pipeline on x86 and RISC-V CPUs. We (1) investigate the total number of reductions identified by our pass and missed by MLIR "match reduction test pass", (2) perform a performance analysis of ONNX-MLIR, OML and OML-vect pipeline, and (3) discussion on memory consumption analysis, instruction count and binary sizes.

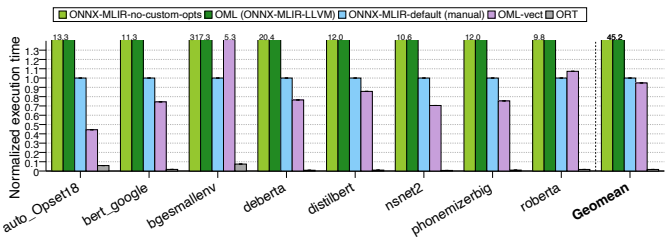


Fig. 8: Performance on Intel Xeon (x86)

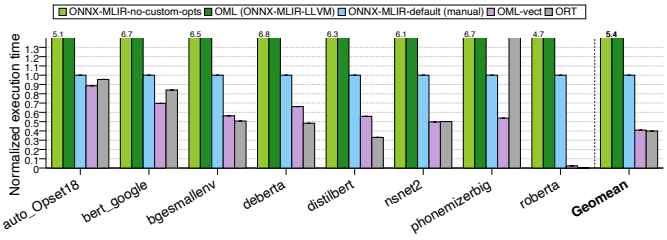


Fig. 9: Performance on Atrevido (RISC-V)

A. Impact of the Reduction Pass

We first analyze the number of reductions identified by the *match reduction test* pass in MLIR compared to our proposed reduction pass. Figure 7 presents the total reductions identified for each of our workloads. Our proposed pass demonstrates significant improvements over the "match reduction test" pass for the following reasons:

- The pass identifies reductions across multiple dialects, such as *arith* and *math*, whereas `match reduction test` works on *arith* dialect only.
- Within the *arith* dialect, our pass identifies reductions that are not handled by the `match reduction test` pass (e.g., *arith.maxnumf*. This operation originates from SoftMax operations found in self-attention layers [68].
- In Transformers, where matrix multiplication is a primary computational bottleneck, it is more efficient to use *math.fma* operations instead of separate *mulf* and *addf* operations. However, the MLIR `match reduction test` pass does not recognize the addition reduction present in *math.fma*, which adversely affects the vectorization of matrix multiplications. In contrast, our approach identifies the addition reduction present in *math.fma* as a reduction operation.

In summary, our OML-reduction introduces a *systematic decomposition strategy to identify and vectorize complex ternary reductions*, including operations such as fused multiply-add (FMA). This fills a gap in MLIR, which lacks such reduction pattern detection. As a result, OML-reduction achieves up to 2.5× greater reduction coverage in NNs than default MLIR, especially benefiting auto-vectorization.

Discussion: OML-reduction detects reductions in non-Transformer workloads, e.g. in elementwise operations such as softmax, maxpooling (and related variants), and in convolutions. Convolutions use fused multiply–add instructions, where add is the reduction operation. However, convolution is inherently non–vectorizable despite the detection of the reductions.

B. Performance Analysis

We evaluated the performance of several approaches (see Section V) compared to ONNX-MLIR-default (manual) as a baseline. Figure 8 shows our results on x86 while Figure 9 shows the results on RISC-V. We focus primarily on Transformer models, where MatMul is the main bottleneck.

MLIR out-of-the-box auto-vectorization fails to vectorize most of the MatMuls due to failure in reduction identifications and Therefore, there are no significant performance gains in ONNX-MLIR-no-custom-opts (light green bar in Figure 8 and Figure 9). The OML-reduction pass identifies more reductions than the original ONNX-MLIR-no-custom-opts flow, yet it does not yield performance gains when applied without data layout optimizations because the cost model (both MLIR and LLVM) predicts performance penalties due to non-contiguous data and hence it prevents vectorization. Hence, no performance improvements in OML (light blue bar).

ORT includes a C/C++ library with manually vectorized code optimized for RISC-V and x86 (bar ORT in Figure 8 and Figure 9). ORT (gray bar) consistently outperforms ONNX-MLIR-default (manual) (see dark-green bar) and OML-vect (dark-blue bar) on x86 due to the extensive optimization, parallelization, and vector intrinsics tailored for x86, delivered by manually fine-tuned libraries. ORT is multi-threaded by default, we tried disabling multithreading through environment variables(OMP_NUM_THREADS=1) and runtime options (opts.op_num_threads=1), but VTune showed multiple threads still running. On x86, the OneDNN library directly queries hardware-core data, overriding OMP settings [48] and giving ORT a performance advantage over ONNX-MLIR and OML-vect. Disabling parallelization requires building the ORT from source. We did not attempt this in a bid to evaluate the best performance achievable with mainstream frameworks (ORT). In contrast, ONNX-MLIR-default (manual) and OML-vect are single-threaded by default. Such optimizations need to be exposed from ORT in MLIR to be leveraged in ONNX-MLIR-based frameworks. This work focuses on automating manual (pattern-based) vectorization as a fist step. Unlike for x86, there is less fine-tuning for RISC-V for the ORT, thus OML-vect outperform ORT on RISC-V for 4 out of 8 models (see Figure 9). ORT includes special optimizations for popular Transformer models such as BERT-based architectures[53]. These optimizations are manually designed and fine-tuned for specific cases. However, in lesser-known models such as phonemizerbig, absence of such targeted optimizations leads to a significant performance gap between ORT and OML-vect.

On average, auto-vectorization in OML-vect (OML-reduction + data layout optimizations) achieves 5% and 59% performance improvements over the baseline on x86 (see Figure 8) and RISC-V (see Figure 9) respectively. These findings highlight the necessity of combining the reduction pass with data layout optimizations to fully leverage auto-

vectorization capabilities of MLIR and LLVM and achieve significant performance gains.

ONNX-MLIR-manual runs a more efficient manually generated MatMul algorithm (see Section II-C1) combined with hand vectorization, unroll and jam, and tiling. Notably, OML-vect approach works on top of the standard MatMul algorithm (Figure 1) and does not tile the matrices. We observed that tiling combined with packing only becomes beneficial for very large matrices, i.e., larger than 3000×1000, and for smaller matrices it incurs additional overhead on x86 and RISC-V. For example, in attention layers from models like BERT and RoBERTa, where matrix dimensions are typically 768×768, tiling introduces overhead.

Transposing the matrix enables consecutive memory access, making auto-vectorization both feasible and beneficial. Without transposition, vectorization is either disabled by the compiler's cost model or, if forced, proves non-beneficial (Figure 3). The ONNX-MLIR implementation adopts a different strategy by interchanging the innermost loops. This improves data reuse for the first matrix (A) while ensuring consecutive memory accesses for the second matrix (B). However, this approach requires computing partial results, which increases the number of loads and stores to the output matrix. RISC-V runtime traces show that the proposed OML-vect reduces vector loads by 18.83% and stores by 99% compared to ONNX-MLIR-manual, thereby improving memory efficiency and data reuse through complie-time data-layout transformation.

Without manual optimizations, OML-vect achieves performance comparable to ONNX-MLIR-default (manual) implementation for most attention layers on both x86 and RISC-V CPUs. Given that the attention layers are the most time consuming fraction of the models (ranging between 90%-97%), we observe the same trends when we run the full Transformer model (on x86 and RISC-V).

C. Instruction Count, Binary Size, Memory Footprint, and Compilation Time

We analyzed the binaries size, instruction count, memory footprint and compilation time of the manually optimized ONNX-MLIR library and OML-vect.

Binary Sizes and Instruction Count: We observed that OML-vect's binaries are 2.67% larger than manually optimized ONNX-MLIR's binaries, while OML-vect's instruction count is 2.2% lower.

Memory Footprint: We analyzed the memory footprint on Intel platforms with VTune. Profiling tools required for such an analysis on RISC-V are currently unavailable. Our study shows that OML-vect requires 35% less memory compared to the ONNX-MLIR flow, which is critical when running on resource-constrained devices. The primary advantage comes from avoiding explicit copying of tiles and padding (due to partial tiling). In contrast, ONNX-MLIR always tiles and pads partial tiles, increasing memory usage. For memory-constrained environments, our approach efficiently vectorizes operations, achieving on-par performance while reducing the memory footprint by more than a third.

Compilation Time: With OML-vect we noticed 3%-8.5% of compilation time was spent on our transpose pass and OML-reduction pass. Overall compilation time changes are as follows. After adding transpose and reduction passes, RISC-V compilation rises from 11sec to 23.1sec ($\approx 40\%$ slower), while x86 time drops from 9.5sec to 8.2sec. The increase stems from passes exposing additional vectorizable loops, which expand RVV code, create larger machine-IR graphs for legalization and scheduling, and prolong LLVM's instruction selection and linking, whereas the mature x86-backend lowers such vector code efficiently.

VII. Related Work

A. Neural Network Models for Edge Devices

The rapid development of edge computing has led to specialized accelerators like Google's Edge TPU, designed to optimize neural network inference on resource-constrained devices. However, these accelerators face limitations due to their monolithic design, which does not address the diverse requirements of different neural network models and layers, especially regarding memory bottlenecks and layer-to-layer communication overhead.

In response to these challenges, the Mensa framework [12] introduces a more flexible and efficient approach by using a collection of heterogeneous hardware accelerators.

Mensa also integrates data layout optimizations (such as the im2col operation for convolution) used in the Edge TPU, ensuring compatibility with existing hardware. However, Mensa accelerators communicate between layers only during execution transitions, significantly reducing communication overhead. Notably, the Mensa scheduler efficiently handles inter-layer and inter-accelerator communication, minimizing the memory bottleneck issue observed in existing accelerators.

B. Porting PyTorch to RISC-V

Prior work [15] tried to port PyTorch to RISC V. However, several PyTorch dependencies, such as the Chromium Breakpad library, the SLEEF vectorized Math Library, and the PyTorch CPU INFOrmation library (cpuinfo), were not compatible with the RISC-V ISA. The authors extended support for these libraries for RISC-V. The PyTorch CPU INFOrmation library (cpuinfo) plays a crucial role in optimizing the neural network for the underlying hardware, as it provides a uniform cross-platform layer for accessing information about the host CPU. PyTorch relies on cpuinfo to optimize performance when running on a CPU, e.g., to detect support for SIMD instructions or to pin threads to cores in NUMA architectures. The intrinsically dynamic nature of the RISC-V ISA makes it quite complex to fully port this library.

C. Auto-Vectorization with Imitation Learning

Mendis *et al.* [36] frame SIMD vectorization as a Markov Decision Process and apply DAGGER imitation learning [60]

to guide instruction packing. A Gated Graph Neural Network encodes instruction-graph dependencies, enabling fine-grained packing decisions across multiple node types. The policy is first trained with supervision and then refined through DAGGER.

Evaluation shows a 22.6% greater static cost reduction than LLVM's SLP vectorizer, measured by the LLVM cost model. This demonstrates the potential of learning-based methods to complement LLVM heuristics and provide efficient, competitive vectorization policies.

D. Run-Time Data Layout Optimization

Data layout optimization is crucial for efficient vectorization. Previous works [30, 56] explores data replication, which negatively impacts cache behavior, memory consumption, and adds runtime overhead. Data replication leads to inefficient use of cache because replicated data increases the working set size, causing more frequent cache evictions and reducing cache locality. Additionally, replication increases the memory footprint, as multiple copies of the same data consume more memory, which can lead to higher memory bandwidth usage and reduced scalability for large datasets. Such data layouts achieve improvements only if the benefits of layout optimization outweigh the costs of data replication. SIMD code shows performance improvements only for large array sizes because the overhead of data permutations nullifies the performance benefits of consecutive SIMD load/stores. In contrast, the proposed work moves this data layout optimization to compile time, ensuring that the transformations are vectorization - friendly. This eliminates the runtime overhead associated with data replication and ensures consecutive memory accesses.

VIII. Conclusion

Our proposed auto-vectorization approach OML-vect demonstrates 5% and 59% performance improvements over the baseline (ONNX-MLIR-manual) on x86 and RISC-V, respectively, compared to manually optimized libraries. While on x86 ONNX-MLIR-manual achieves higher performance in some cases – such as bgesmallenv – our approach is entirely automatic, making it more adaptable and efficient across a wide range of workloads and architectures. The OML-reduction pass relies on complementary optimizations, such as complie-time data layout transformations, to achieve highest performance. As demonstrated in this work, we enable compile-time vectorization-friendly data-layout transformations, eliminating runtime transpose overhead and the reliance on external, manually-optimized or manually-ported libraries.

Acknowledgment

This work has been partially funded by the EU's Horizon 2021 research and innovation program (CONVOLVE, g.a. no. 101070374 under HORIZON-CL4-2021-DIGITAL-EMERGING-01), the CDTI within the framework of the "Science and Innovation Missions" program linked to the Strategic Project for Microelectronics and Semiconductors, PERTE CHIP, (PRO-VISION-PRO, g.a. MIG-20231020), the MCIN/AEI/10.13039/501100011033/, the "ERDF A way of making Europe", EU (DAMAS, grant PID2022-136315OB-I00), and the Ramón y Cajal Research Contract (RYC2018-025200-I). The authors thank Semidynamics for their support, useful comments, and discussions along the way.

Appendix

Artifact Appendix

A. Abstract

This is the supporting artifact for the paper titled "Enabling Automatic Compiler-Driven Vectorization of Transformers" as published in CGO 2026. It includes the source code for the proposed tool (oml-vect), along with scripts to compile and reproduce all experimental results presented in the paper. We provide a Docker image containing all necessary dependencies preinstalled, enabling straightforward setup and execution of the experiments. The setup supports running the experiments on an x86 host, and it also allows cross-compilation for RISC-V ISA, and it automatically transfers the statically linked RISC-V binaries to the host to be then copied to the RISC-V target via `scp`. The reported results were obtained on Intel Xeon an x86_64 Intel(R) Xeon(R) E5-2630 v4 CPU @ 2.20GHz and a Xilinx U55C FPGA emulating an Atrevido 423 RISC-V 64-bit core with a 512-bit vector unit [6].

B. Artifact Check-List

- **Algorithm:** Data-layout optimization and reduction identification to enable auto-vectorization in MLIR
- **Program:** ONNX graph representations of transformer and neural network models
- **Compilation:** Publicly available and included in this artifact: LLVM v20.0, MLIR v20.0, and ORT v1.23.2
- **Transformations:** onnx-mlir toolchain to obtain vectorized affine dialect code for each kernel and LLVM to statically compile binaries for x86 and RISC-V, included in this artifact
- **Binary:** Linux executables for x86 and RISC-V included in this artifact and scripts to generate these binaries automatically.
- **Data set:** Included in this artifact, testing datasets used for all models are detailed in the corresponding references in section V
- **Run-time environment:**
  1) x86 - The Docker container (see section E1)
  2) RISC-V - The Docker container to cross-compile binaries and RISC-V CPU with Linux OS to run the binaries (see section E2)
- **Hardware:** Intel Xeon and RISC-V CPU with vector extension
- **Execution:** We recommend running the experiments in an isolated environment, as the results may vary if other processes are active.
- **Metrics:** Execution time and number of reductions identified
- **Output:** CSV files containing the normalized execution times, performance improvements, and console logs reporting the performance improvements.
- **Experiments:** Docker image included in this artifact contains scripts to regenerate the results. Results may vary with respect to hardware parameters such as vector width, L1, L2 cache size, as the unroll-and jam-factor is hardware dependent.
- **How much disk space required (approximately)?:** 80GB
- **How much time is needed to complete experiments (approximately)?:** Dependent on the CPU and its operating frequency. Completing the full benchmark suite requires 15 - 20 mins on

an Intel 13th Gen Intel(R) Core(TM) i7-13700K, 5.4 GHz with 64 GiB RAM
- **DOI:** [2]
- **Publicly available?:** Yes, github [47, 2]

C. Description

1) How Delivered: This artifact is available at Docker-hub [20]

2) Hardware Dependencies: A machine with vector units is required. We evaluated on an Intel Xeon and Atrevido [5] (RISC-V Core). OML-vect auto-vectorizes the code and improves performance. Performance improvement may vary with the hardware, therefore, we recommend similar platforms to reproduce the results.

3) Software Dependencies: Working Docker installation. This has been tested on a Linux x86 host machine with Docker v28.1.1. For RISC-V, we recommend Bianbu Star 2.1.7 or LINUX OS.

To test the state-of-the-art tool ORT on the RISC-V platform, we used proprietary software Semidynamics ONNX-Runtime [64]. The open source alternatives, such as the Python package (pip install OnnxRuntime), are not supported on RISC-V. Hence, while we used ORT as a point of comparison to our proposal, it cannot be installed on RISC-V without manual porting. This underlines the need for an automatic neural networks compiler for RISC-V, such as oml-vect.

D. Installation

Download the docker image repository using the command

```
docker pull shreyasubhash/omlvect:r1
```

Verify availability of the image

```
docker images -a | grep "omlvect"
```

E. Experiment Workflow

1) x86: To compile, run, and verify the results discussed in this paper (Figure 7, Figure 8), on host machine (x86 Intel), **navigate to the directory where the output should be stored** and run the command:

```
docker run -ti --volume ${PWD}:/workdir/shared \
  shreyasubhash/omlvect:r1 \
  bash -c "/workdir/scripts/infer.sh"
```

This command will compile the benchmarks as described in section IV, execute them (for x86), process the execution time and number of reductions identified for each kernel, and generate CSV files. The resulting CSV files can be accessed on the host machine.

2) RISC-V: To cross-compile and generate binaries for RISC-V, run the command:

```
docker run -ti --volume \
    ${PWD}:/workdir/shared  \
    shreyasubhash/omlvect:r1 \
    bash -c "/workdir/scripts/compile-riscv.sh"
```

This command will cross-compile all the benchmarks as described in section IV, and the statically linked ELFs will be available on the host machine ${PWD}/elf-rvv along with *infer-rvv.sh* script to run the files on the RISC-V CPU and generate the corresponding CSV file.

To obtain speedup measurements for the RISC-V platform, first copy the *runtime.csv* file generated on the RISC-V CPU into the **${PWD}** directory on the host system. Then execute the following command to process the results:

```
docker run -ti --volume \
 ${PWD}:/workdir/shared \
 shreyasubhash/omlvect:r1 \
 bash -c "/workdir/scripts/get-rvv-speed-up.sh \
 /workdir/shared/<your-file>.csv"
```

Details of all generated CSV files are provided in Table II for both x86 and RISC-V experiments.

TABLE II: Summary of output CSV files and their contents.

Filename	Information
Runtime.csv	Raw runtime
normalised_runtime.csv	Normalised runtime compared to baseline (`onnx-mlir-no-cust-opts`)
performance.csv	Percentage improvement in performance over the baseline
reduction_counts.csv	Total number of reduction identified by MLIR and OML-vect

F. Evaluation and Expected Results

We expect OML-vect will autovectorize the code and reduce the execution time, but the performance improvements may vary across different hardware platforms, since the vector width and unroll-and-jam factor are hardware-dependent.

G. Reusability

The following instructions summarize how to reproduce the OML-vect workflow. See the README in the Docker image *shreyasubhash/omlvect:reusable* [21] and the GitHub repo [47]

1) OML-Vect Toolchain Workflow: Here we describe the full OML-vect workflow—from modifying the toolchain to generating and executing the final ELF.

a) Code Changes for OML-Vect: The following subsection details the complete OML-vect workflow, including the required toolchain updates, compilation steps, and execution of the resulting ELF.

- Incorporate the revised MatMul algorithm in the file‘src/Conversion/ONNXToKrnl/Math/MatMul.cpp‘ as discussed in Section IV-B
- Implement the reduction pass and invokes the MLIR vectorization pipeline in file ‘src/Conversion/KrnlToAffine/ConvertKrnlToAffine.cpp‘ as discussed in section IV-C

Additional Scripts and other file modifications

- Preprocessing steps for data layout transformation are described in Section IV-A.
- A CMake file enables cross-compilation when ONNX-MLIR runtime libraries (Section IV-D) are not available for the target hardware.

Follow the build instructions in the onnx-mlir repository [47] with modifications to files as mentioned above to compile **oml-vect**

b) Data Layout Modification: Run:

```
data-layout.sh <input onnx graph>
```

This command applies Algorithm 1 (Section IV-A) to generate transposed MatMul's operand B.

c) Get Vectorized MLIR Code: Run:

```
<oml-vect-build-path>/bin/onnx-mlir -O3 \
  --EmitLLVMIR \
  --vlen=<vector length> \
  --uf1=<unroll factor k> \
  --uf2=<unroll factor m> \
  --uf3=<unroll factor n> \
  --EmitLLVMIR <input.onnx>
```

This command integrates the updated MatMul algorithm (Algorithm 1, Section IV-B), runs the reduction-mapping pass (Algorithm 2, Section IV-C), and then invokes MLIR super-affine vectorization using the resulting reduction map to generate vectorized LLVM Dialect.

d) Get LLVM IR: Run:

```
mlir-translate \
  --mlir-to-llvmir <input.mlir> > <output.ll>
```

This command converts vectorized MLIR to LLVM IR.

e) Optimize LLVM IR: Run:

```
opt -O3 -S \
    <input.ll> -o <output.opt.ll>
```

This command applies LLVM IR O3 level optimizations

f) Get Assembly Code: Run:

```
llc -O3 \
  -mcpu=<MCPU> --filetype=asm <input.opt.ll> \
  -o <output.s>
```

This command generates architecture-specific assembly.

g) Compile ONNX-MLIR Runtime Libraries: Set the property *CMAKE_C_COMPILER* in *oml-vect/runtime* to desired compiler for target Hardware and
Run:

```
cd oml-vect/runtime && cmake .
```

This command cross complies the runtime support libraries cross compiled for given target hardware required by ONNX-MLIR and OML-vect generated kernels.

h) Get ELF for Target Hardware: To generate ELF by Linking assembly, runtime libraries from step 7, and main.cpp and other dependencies into the final ELF executable run:

```
./get-elf.sh <main.cpp> \
    -march=<march> <asm_file.s>
```

i) Run the ELF Using the Command:

```
./<output_elf> <input_tensor_as_numpy_array.c>
   <dim1*dim2*dim3*...>
```

2) Portability Across Hardware: Porting OML-vect to any new hardware requires no modifications to our methodology. One only needs to re-run Steps 6–9 with the target architecture's -mcpu/-march values. Because OML-vect relies exclusively on LLVM, all LLVM-supported hardware targets are inherently supported.

3) Running New Benchmarks: To add new benchmarks, it is sufficient to:

- Re-execute the data-layout preprocessing (Step G.1.B) to construct a new ONNX graph with the transposed B input
- Run the provided script 'run-infer.sh' using the new input tensors to run the inference using the command
  `run-infer.sh <ELF name>  \`
  `<input_tensor_as_numpy_array.c> \`
  `<dim1*dim2*dim3*...>'`

H. Notes

Depending on the host machine configuration, Docker commands might require elevated privileges (sudo). Shell scripts inside ${PWD}/shared/elf-rvv folder may need execute permissions, run *(sudo) chmod +x filename*

REFERENCES

[1] *Affine Dialect*. 2022. URL: https://mlir.llvm.org/docs/Dialects/Affine/.

[2] Shreya Alladi, Alberto Ros, and Alexandra Jimborean. *OML-vect: Enabling Automatic Compiler-Driven Vectorization of Transformers*. Version v1. Dec. 2025. DOI: 10.5281/zenodo.18006548. URL: https://doi.org/10.5281/zenodo.18006548.

[3] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2025, pp. 929–947. ISBN: 9798400703850. DOI: 10.1145/3620665.3640366. URL: https://doi.org/10.1145/3620665.3640366.

[4] George B. Arfken and Hans J. Weber. *Mathematical Methods for Physicists: A Comprehensive Guide*. 7th ed. Academic Press, 2013.

[5] *Atrevido (online)*. 2023. URL: https://semidynamics.com/en/products/atrevido.

[6] *Atrevido 423 with V8 vector unit (online)*. 2025. URL: https://riscv.org/news/2023/07/semidynamics-announces-fully-customisable-4-way-atrevido-423-risc-v-core-for-big-data-applications/.

[7] *Auto-Vectorization in LLVM (online)*. 2010. URL: https://llvm.org/docs/Vectorizers.html#the-loop-vectorizer.

[8] *BertForSequenceClassification (online)*. 2020. URL: https://huggingface.co/textattack/bert-base-uncased-yelp-polarity.

[9] *BertForSequenceClassification (online)*. 2016. URL: https://huggingface.co/docs/transformers/v4.41.3/en/model_doc/bert#transformers.BertForSequenceClassification.

[10] *bge-small-en Model by BAAI*. 2025. URL: https://huggingface.co/BAAI/bge-small-en.

[11] Uday Bondhugula. *High Performance Code Generation in MLIR: An Early Case Study with GEMM*. 2020. arXiv: 2003.00532 [cs.PF]. URL: https://arxiv.org/abs/2003.00532.

[12] Amirali Boroumand et al. *Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks*. 2021. arXiv: 2109.14320 [cs.AR]. URL: https://arxiv.org/abs/2109.14320.

[13] Sebastian Braun and Ivan Tashev. "Data Augmentation and Loss Normalization for Deep Noise Suppression". In: *Speech and Computer*. Ed. by Alexey Karpov and Rodmonga Potapova. Cham: Springer International Publishing, 2020, pp. 79–86. ISBN: 978-3-030-60276-5.

[14] Iacopo Colonnelli, Robert Birke, and Marco Aldinucci. "Experimenting with PyTorch on RISC-V". In: RISC-V Summit Europe 2023. Barcelona, Spain, 2023. URL: https://hdl.handle.net/2318/1925291.

[15] Iacopo Colonnelli, Robert Birke, and Marco Aldinucci. "Experimenting with PyTorch on RISC-V". In: *RISC-V Summit Europe*. University of Torino, Computer Science Dept. Corso Svizzera 185, 10149, Torino, Italy, June 2023.

[16] *CPU Info PyTorch Dependency Library (online)*. 2017. URL: https://github.com/pytorch/cpuinfo.

[17] *Cross-Attention Mechanism in Transformers*. en-US. Section: NLP. URL: https://www.geeksforgeeks.org/nlp/cross-attention-mechanism-in-transformers/ (visited on 11/27/2025).

[18] *Hugging Face Transformers: DeBERTa Model Documentation*. 2025. URL: https://huggingface.co/docs/transformers/model_doc/deberta.

[19] *Hugging Face Transformers: DistilBERT Model Documentation*. 2025. URL: https://huggingface.co/docs/transformers/en/model_doc/distilbert.

[20] *OML-vect docker image (online)*. URL: https://hub.docker.com/repository/docker/shreyasubhash/omlvect/tags/graphs.

[21] *OML-vect docker image (online)*. URL: https://hub.docker.com/repository/docker/shreyasubhash/omlvect/tags/reusable.

[22] *PyTorch: An Ever-growing AI Framework Among AI Practitioners (online)*. 2023. URL: https://www.huawei.com/sg/news/sg/2023/pytorch-an-ever-growing-ai-framework-among-ai-practitioners.

[23] *Gemm - ONNX 1.21.0 documentation*. URL: https://onnx.ai/onnx/operators/onnx__Gemm.html (visited on 11/27/2025).

[24] *IREE internal dialects*. 2023. URL: https://iree.dev/reference/mlir-dialects/#iree-internal-dialects.

[25] *IREE-RISCV cross compile completely infeasible (online)*. 2025. URL: https://github.com/iree-org/iree/issues/19057.

[26] *IREE: Intermediate Representation Execution Environment*. 2019. URL: https://github.com/openxla/iree.

[27] Chris Lattner et al. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. 2020. arXiv: 2002.11054 [cs.PL].

[28] Tung D. Le et al. "Compiling ONNX Neural Network Models Using MLIR". In: *ArXiv* abs/2008.08272 (2020). URL: https://api.semanticscholar.org/CorpusID:221173137.

[29] Hsin-I Cindy Liu et al. "TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment". In: *IEEE Micro* 42.5 (2022), pp. 9–16. DOI: 10.1109/MM.2022.3178068.

[30] Jun Liu et al. "A compiler framework for extracting superword level parallelism". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: Association for Computing Machinery, 2012, pp. 347–358. ISBN: 9781450312059. DOI: 10.1145/2254064.2254106. URL: https://doi.org/10.1145/2254064.2254106.

[31] Jun Liu et al. "A compiler framework for extracting superword level parallelism". In: *SIGPLAN Not.* 47.6 (June 2012), pp. 347–358. ISSN: 0362-1340. DOI: 10.1145/2345156.2254106. URL: https://doi.org/10.1145/2345156.2254106.

[32] *Compiling LLVM IR to Binary (online)*. 2019. URL: https://borretti.me/article/compiling-llvm-ir-binary.

[33] *LoopReduction in Affine (online)*. 2010. URL: https://mlir.llvm.org/doxygen/structmlir_1_1affine_1_1LoopReduction.html#a10f98858aca24b0a88a38b43759eb246.

[34] *Mastering LLM Techniques: Inference Optimization*. en-US. Nov. 2023. URL: https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/ (visited on 11/27/2025).

[35] *memref.transpose (online)*. 2010. URL: https://mlir.llvm.org/docs/Dialects/MemRef/#memreftranspose-memreftransposeop.

[36] Charith Mendis et al. "Compiler Auto-Vectorization with Imitation Learning". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[37] Riccardo Miccini et al. "Dynamic nsNET2: Efficient Deep Noise Suppression with Early Exiting". en. In: *2023 IEEE 33rd International Workshop on Machine Learning for Signal Processing (MLSP)*. Rome, Italy: IEEE, Sept. 2023, pp. 1–6. ISBN: 9798350324112. DOI: 10.1109/MLSP55844.2023.10285925. URL: https://ieeexplore.ieee.org/document/10285925/ (visited on 01/24/2025).

[38] Gianluca Mittone et al. "Experimenting with Emerging RISC-V Systems for Decentralised Machine Learning". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. CF '23. ACM, May 2023. DOI: 10.1145/3587135.3592211. URL: http://dx.doi.org/10.1145/3587135.3592211.

[39] MLIR. *mlir::affine Namespace Reference*. LLVM Project. n.d. URL: https://mlir.llvm.org/doxygen/namespacemlir_1_1affine.html#ac58a9af2e4bb4c811ad5329293609224.

[40] *MLIR Dialects: Affine For op*. 2025. URL: https://mlir.llvm.org/docs/Dialects/Affine.

[41] *Arith dialect*. 2023. URL: https://mlir.llvm.org/docs/Dialects/ArithOps/#atomicrmwkind.

[42] *AtomicRMWKind*. 2023. URL: https://mlir.llvm.org/docs/Dialects/ArithOps/#atomicrmwkind.

[43] *mlir-git-issue-reduction*. 2023. URL: https://github.com/llvm/llvm-project/issues/61735#issuecomment-1528695423.

[44] *MLIR Test: test-match-reduction.mlir*. 2025. URL: https://github.com/llvm/llvm-project/blob/main/mlir/test/Analysis/test-match-reduction.mlir.

[45] *super-vectorization in affine dialect*. 2022. URL: https://mlir.llvm.org/doxygen/SuperVectorize_8cpp.html#a7b753fbff5d2aa2f6353b3ab53c4dcc6.

[46] *Create a generic reduction detection utility (online)*. URL: https://reviews.llvm.org/D110303.

[47] *OML-vect (online)*. URL: https://github.com/CAPS-UMU/onnx-mlir-caps.

[48] *oneDNN/src/common/dnnl_thread.hpp at e56e7124e14b9ecd7f3a6483d0050d87b1e3270e · uxlfoundation/oneDNN*. en. URL: https://github.com/uxlfoundation/oneDNN/blob/e56e7124e14b9ecd7f3a6483d0050d87b1e3270e/src/common/dnnl_thread.hpp (visited on 11/27/2025).

[49] *ONNX Runtime (online)*. 2018. URL: https://github.com/microsoft/onnxruntime.

[50] *Gemm operator*. 2025. URL: https://onnx.ai/onnx/operators/onnx__Gemm.html.

[51] *ONNX-MLIR RISC-V Support Issue (online)*. 2025. URL: https://github.com/onnx/onnx-mlir/issues/2824.

[52] *Compile using ONNX-MLIR (online)*. 2019. URL: https://github.com/onnx/onnx-mlir/blob/main/docs/mnist_example/README.md.

[53] *ONNX Runtime optimizations: BERT contrib ops (online)*. 2021. URL: https://github.com/ucb-bar/onnxruntime-riscv/tree/2021-12-23/onnxruntime/contrib_ops/cpu/bert.

[54] *phonemizer-big Model by RUPhon*. 2025. URL: https://huggingface.co/RUPhon/phonemizer-big/tree/main.

[55] *A BetterTransformer for Fast Transformer Inference (online)*. 2022. URL: https://pytorch.org/blog/a-better-transformer-for-fast-transformer-encoder-inference/.

[56] Gang Ren, Peng Wu, and David Padua. "Optimizing data permutations for SIMD devices". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 118–131. ISBN: 1595933204. DOI: 10.1145/1133981.1133996. URL: https://doi.org/10.1145/1133981.1133996.

[57] Gang Ren, Peng Wu, and David Padua. "Optimizing data permutations for SIMD devices". In: *SIGPLAN Not.* 41.6 (June 2006), pp. 118–131. ISSN: 0362-1340. DOI: 10.1145/1133255.1133996. URL: https://doi.org/10.1145/1133255.1133996.

[58] *Spike RISC-V ISA Simulator (online)*. 2010. URL: https://github.com/riscv-software-src/riscv-isa-sim.

[59] *RoBERTa FacebookAI roberta-base (online)*. 2016. URL: https://huggingface.co/docs/transformers/en/model_doc/roberta.

[60] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*. 2011. arXiv: 1011.0686 [cs.LG]. URL: https://arxiv.org/abs/1011.0686.

[61] Bo-Kai Ruan, Hong-Han Shuai, and Wen-Huang Cheng. *Vision Transformers: State of the Art and Research Challenges*. 2022. arXiv: 2207.03041 [cs.CV]. URL: https://arxiv.org/abs/2207.03041.

[62] *Self - Attention in NLP*. en-US. Section: NLP. URL: https://www.geeksforgeeks.org/nlp/self-attention-in-nlp/ (visited on 11/27/2025).

[63] *Semidynamics-Evaluate*$_S emidynamics_R ISC - V_T echnology.(online)$. URL: https://semidynamics.com/en/resources/evaluation.

[64] *SMD ORT (online)*. 2025. URL: https://semidynamics.com/en/resources/software#Semidynamics-ONNX-RT.

[65] Jialiang Tan et al. "Toward efficient interactions between Python and native libraries". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1117–1128. ISBN: 9781450385626. DOI: 10.1145/3468264.3468541. URL: https://doi.org/10.1145/3468264.3468541.

[66] *Torch MLIR (online)*. 2020. URL: https://github.com/llvm/torch-mlir/tree/main/projects.

[67] *Torch-MLIR: The Torch-MLIR Project*. 2020. URL: https://github.com/llvm/torch-mlir.

[68] Ashish Vaswani et al. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.