Insights into Interpreters, Compilers, and Optimizers for Neural Networks

Shreya Alladi, Ravikiran Ravindranath, Alberto Ros, Alexandra Jimborean
University of Murcia
Murcia, Spain
shreya.a.s@um.es,ravikiran.r.r@um.es,aros@ditec.um.es,alexandra.jimborean@um.es

Abstract

Frameworks such as PyTorch, TensorFlow, and ONNX Runtime (ORT) utilize handwritten kernels highly optimized for hardware platforms such as x86, but not optimized for RISC-V. As RISC-V CPUs and accelerators emerge through open-source initiatives, mainstream frameworks such as PyTorch struggle to keep up with rapidly evolving hardware. Adapting these handwritten kernels for new configurations requires significant rework. Additionally, the Python interpreter introduces runtime overhead that increases inference time.

This study presents an end-to-end compiler pipeline to generate executable binaries from pre-trained PyTorch models. We expose MLIR dialects and LLVM IR, enabling automatic optimizations offered by MLIR and LLVM tailored for RISC-V, with potential extensions to various hardware architectures. We analyze the optimizations provided by PyTorch and ORT and discuss the advantages and limitations of the LLVM compiler for neural networks. We highlight the optimizations available for the common layers and emerging layers in Recurrent Neural Networks (RNNs), Long-Short-term memory (LSTMs), and Transformers.

In addition, we present a case study on nsNet2, with code variants of increasing complexity (static and dynamic). All in all, instead of relying on the manual fine-tuning of the neural networks performed in PyTorch and ORT for specific platforms, this comparative study will help to introduce robust automatic compiler optimizations to neural network workloads.

CCS Concepts

• **Software and its engineering → Compilers**.

Keywords

Neural networks, compilers, LLVM

ACM Reference Format:
Shreya Alladi, Ravikiran Ravindranath, Alberto Ros, Alexandra Jimborean. 2025. Insights into Interpreters, Compilers, and Optimizers for Neural Networks. In *Proceedings of Workshop on Compilers, Deployment, and Tooling for Edge AI 2025 (CODAI '25).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CODAI '25, January 20, 2025, Barcelona, ES
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The utilization of Deep Neural Networks (DNN) has surged exponentially over the past twenty years. As data volumes grow, increasingly complex networks are employed to enhance the accuracy of predictions. The majority of Neural Networks (NN) are developed mostly using either interpreters or Just-In-Time (JIT) frameworks such as PyTorch, TensorFlow, and PyTorch JIT. Python, an interpreted language used by PyTorch and TensorFlow, exhibits a significant performance slowdown compared to static languages such as C [51, 13]. Although PyTorch and Tensorflow use highly optimized manually fine-tuned kernels written in C or C++, the Python wrapper can introduce performance delays due to library calls [45]. Moreover, these frameworks are optimized for Intel, ARM, and AMD CPUs and GPUs, and not for RISC-V platforms.

The RISC-V open-source Instruction Set Architecture (ISA) is gaining popularity [23], leading to the development of RISC-V cores [11, 2] and dedicated accelerators for AI workloads based on RISC-V ISA (or extended RISC-V ISA) [50, 17, 38, 14]. While efforts have been made to port PyTorch to RISC-V [6], they faced significant challenges. Not all required libraries are readily available for RISC-V (e.g. cpu info [8, 7]), and substantial efforts are needed to rewrite optimizations for RISC-V, leading to large performance gaps between the x86 and RISC-V architectures [7]. Alternatively, there are ongoing efforts to lower pre-trained models to C [21, 13, 5], a compiled programming language. However, this approach still requires implementing operations and/or hardware-specific optimizations manually to achieve performance comparable to PyTorch and Open Neural Network Exchange (ONNX)-Runtime (ORT).

A static compilation pipeline combined with automatic compiler analysis is essential to run neural networks efficiently on RISC-V CPUs, accelerators, and edge devices. The process begins by translating the neural network into Low Level Virtual Machine (LLVM) Intermediate Representation (IR) [18] or Multi-level Intermediate Representation (MLIR) dialects [19], followed by MLIR or LLVM optimizations. Finally, it ends with the generation of an executable binary. To analyze code effectively and optimize it for specific architectures and accelerators the LLVM IR abstraction is widely used. LLVM passes are used for static analysis, loop analysis, loop vectorization, loop tiling, and multiple compiler optimizations. This approach leverages the extensive and robust optimization set developed by the LLVM community. Moreover, and most importantly, future optimizations will be automatically available. Additionally, optimizations can be extended and adapted for the characteristics of each use case and target platform, to achieve maximum performance.

Initiatives such as Torch-MLIR [46] and Intermediate Representation Execution Environment (IREE) [16] aim to address this problem by lowering neural network models to MLIR abstraction, as running Python on embedded systems is heavy in terms of memory usage. However, these tools, developed independently, are not yet fully compatible end-to-end, from neural network model conversion to MLIR [19] or LLVM IR [18] abstraction and the generation of executable binaries for RISC-V [15].

The IREE compiler consists of two components: IREE turbine, formerly known as shark turbine, and IREE (IREE compiler + IREE runtime). The IREE runtime is currently unable to generate executables for RISC-V (64-bits) [15]. IREE does not expose the generated LLVM IR, which hampers the creation of executable binaries through the LLVM project [24]. Instead, it exposes higher-level dialects such as the torch dialect and the linalg dialect. However, the generated MLIR code is not compatible with upstream MLIR due to the custom dialect *"Utils"* and hence it cannot be lowered to LLVM IR and subsequently to executable binaries without the use of the IREE Runtime. Additionally, running the IREE runtime on edge devices is complex and requires significant memory due to dependencies such as Python. TinyIREE [22], the subset of IREE generates a compact NN workload that are optimized for embedded systems. Since TinyIREE is built on IREE, it inherits the same issues as IREE. Torch MLIR frameworks also face challenges in producing abstractions that align with the established MLIR and LLVM ecosystems, often resulting in limited support for JIT, incompatible MLIR dialects, or LLVM IR. This can lead to integration issues and difficulties in exploiting the full potential of LLVM and MLIR.

This work aims to provide a comparative analysis of the manually fine-tuned optimizations of frameworks like PyTorch and ORT, along with the advantages and limitations of the LLVM compiler for neural network workloads. Additionally, we present an end-to-end compilation pipeline to generate executable binaries for neural networks, utilizing MLIR and LLVM IR as intermediate representations. We also analyze the nsNet2 usecase and extract insights to enable powerful automatic compile-time optimizations specifically tailored for RISC-V-based CPUs and accelerators.

In this paper, we make the following contributions:

- Compile/interpret these use cases using different frameworks, including PyTorch, PyTorch-JIT, ONNX-Runtime, and onnx-mlir-llvm (OML).
- Provide performance comparisons and insights regarding the optimizations performed in each framework.
- Build a compilation chain onnx-mlir-llvm (OML) that exposes compiler intermediate representation IR to enable powerful optimizations.
- Extend and enhance the compilation chain by compiling onnx-mlir runtime libraries and porting them to RISC-V platforms, addressing the typical compatibility limitations of state-of-the-art tools, which by default are only compatible with x86 architectures.
- Study complex benchmarks, such as dynamic neural networks and transformers, and analyze in-depth a use-case (nsNet2), comparing its behavior and performance when compiled with various frameworks and on two distinct ISAs (x86 and RISC-V). The analysis of the automatic optimization and vectorization capabilities for this use case unveiled the compiler limitations in autovectorizing neural networks, which we aim to address in our future work.

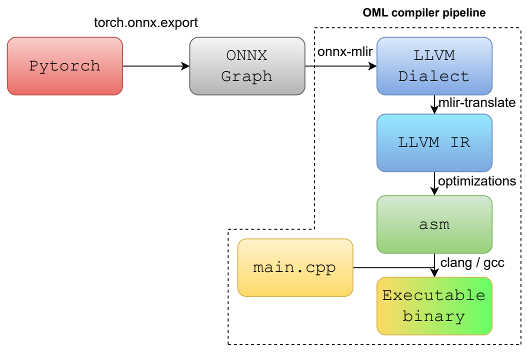

Figure 1: OML compiler pipeline to generate executable binary for neural networks

2 Compiling neural network models to executable binaries for RISC-V

We propose a multi-stage compiler pipeline onnx-mlir-llvm (OML) shown in Figure 1, to expose MLIR and LLVM IR and generate an executable binary for neural networks. The pipeline starts with exporting the neural network to the ONNX graphs. PyTorch offers `torch.onnx.export` and `torch.onnx.dynamo_export` (beta) functions to export PyTorch models to ONNX computation graph. Likewise, frameworks such as Tensor-Flow and Scikit-learn offer methods to convert the models to ONNX graphs. The next step is to translate ONNX graph to the LLVM dialect of MLIR [19] using ONNX-MLIR [20], going through the affine dialect. The LLVM dialect is a wrapper class for LLVM IR, which maps LLVM IR within MLIR by defining the corresponding operations and types.

ONNX-MLIR doesn't expose LLVM IR, hampering the creation of executable binaries through the LLVM project [24]. OML exposes LLVM IR by lowering the LLVM dialect to obtain LLVM IR, thus aiding the generation of stand-alone executable binaries. The obtained LLVM IR is an entry point to the LLVM compiler, enabling automatic optimizations and the generation of assembly (asm) code through the robust LLVM static compiler (llc) back-end. The assembly code is then passed to a native assembler and linker to generate a native executable. ONNX-MLIR currently does not have built-in support for the RISC-V ISA [29]. We statically compile ONNX-MLIR runtime libraries such as OMInstrument.c, OMRandomNormal.c, OMTensor.c, etc., to generate RISC-V runtime libraries. [1] The generated RISC-V runtime library, assembly code, and the main file (to call inference for neural networks) are linked using clang (or gcc) to generate an executable optimized for the RISC-V architecture[2]. The proposed approach generates statically linked binaries for neural networks bypassing the runtime overhead of Python and other shared libraries [30]. Our approach contrasts onnx-mlir which can generate only shared library files and therefore requires more memory.

OML exposes MLIR dialects and LLVM IR for a wide range of optimizations, from high-level (such as vectorization, buffer allocation and optimization) to low-level and hardware-specific (such as

[1]Since these libraries were not readily prepared for RISC-V, the porting process involved manually fixing compile-time errors [29].
[2]statically compiled ONNX-MLIR runtime libraries and the OML pipeline flow is found at [28]

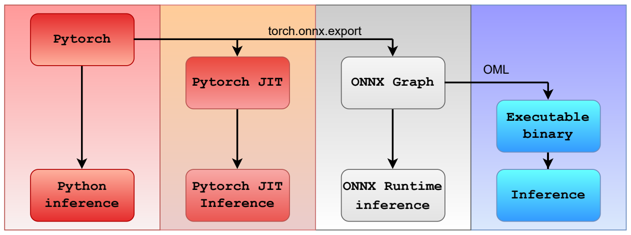

Figure 2: Pipeline illustration to compile/interpret neural network workloads using PyTorch, ORT and OML

macro-ops fusion, fine-tune vectorization, specific ISA extensions, etc., to fully leverage hardware capabilities).

3 Analysis of optimizations offered by State-of-the-Art frameworks

Since 2019, PyTorch has experienced a surge in its adoption, primarily due to dynamic computation graph, ease of debugging, data parallelism, and a rich ecosystem of handwritten libraries [12]. Unlike PyTorch, ORT is specifically engineered to boost the performance of machine learning models at the cost of development ease. It aims to optimize the neural network execution across various platforms, ensuring smooth integration and improved inference speeds. We decided to evaluate the performance and analyze the optimizations of PyTorch and ORT to understand their strengths and limitations with a focus on neural network workloads, as shown in Figure 2. We inferred the model using the Python interpreter and used PyTorch JIT for compatible models such as AlexNet, and MobileNet. We inferred the ONNX graph using ORT. Finally, we compiled the ONNX graphs to generate executable binaries as discussed in Section 2.

Both PyTorch and ONNX-Runtime utilizes external and manually optimized libraries such as MKL-DNN [27] and XNNPACK [47] to leverage hardware-specific optimizations and to achieve better performance. Notably, MKL-DNN and MKL functions such as `mkldnn_relu`, `mkldnn_max_pool2d`, and `mkldnn_binary_fusion` leverage hardware-specific optimizations, including vector extensions such as AVX512 instructions on compatible CPUs, thereby improving performance through parallelism and efficient memory management. Additionally, PyTorch's `fusion_unary_attr_map` functionality enables efficient fusion of unary operations in activation functions, e.g., ReLU with attribute mapping, and reducing data movements, thus enhancing performance. `mkldnn_binary_fusion` further optimizes convolution operations by fusing them with compatible binary operations, reducing data movements. These optimizations in PyTorch ensure that complex neural network computations are executed efficiently on x86 CPUs. Moreover, PyTorch uses the external library XNNPACK [47] for function calls such as sigmoid, Global Average Pooling, and Leaky ReLU, which implements optimized algorithms for these NN layers [9, 10].

ORT uses efficient graph optimizations on ONNX graphs, such as removing redundant nodes and computations, statically computing constants, and fusing multiple nodes into a single node. These optimizations run before partitioning the graphs into subgraphs, ensuring they apply to all execution providers. Furthermore, ORT optimizes sub-graphs for different hardware configurations based on the assigned Execution Providers (EP), e.g., convolution and activation fusion on CPU EP. These optimizations are specific to the assigned EP. This approach maximizes performance across CPUs and ORT supported EPs.

We aim to harness the insights gained from existing frameworks in our compilation pipeline. While many of these optimizations have been manually fine-tuned, we aim to bring these optimizations to our compiler and tailor them in an automatic manner.

4 Results and discussion

This section presents our findings for 10 pre-trained neural network models. The training and testing datasets used for these models are detailed in the corresponding references. We studied six basic models: MobileNet [37], AlexNet [36], YOLOv6 [48], YOLOv7 [49], ResNet_101 [39] nsNet2 (default) [4, 26]. There are two dynamic neural networks: two versions of nsNet2 and two transformer models, Google BERT [3] and RoBERTa [40]. We verified the correctness and compared the performance of the verified models, as shown in Figure 2. The trained PyTorch models were inferred using the PyTorch-Python interpreter. Second, we used PyTorch JIT to JIT compile the compatible models and perform inference. Third, ONNX graphs were inferred with the ORT-default. Fourth, we lowered the ONNX graphs to executable binaries through our compilation pipeline OML discussed in Section 2. All models were inferred on two platforms: x86_64 Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz and Xilinx U55C FPGA emulating Atrevido 423 RISC-V 64-bit core with V8 vector unit [2, 43].

Correctness We verified the correctness of twelve models on both ISAs (x86 and RISC-V). The models were trained using the PyTorch framework, and PyTorch inference values serve as the baseline. We observed that for most models, the relative and absolute error is less than 1e-3. Classification models such as Mobilenet and Google BERT can accommodate this error range without significantly affecting the final prediction accuracy. For models like YOLOv6, YOLOv7, and NsNet2 this error is insignificant after post-processing steps like non-maximum suppression and confidence thresholding. But for Model Unet and Super ImageNet, we noticed large errors leading to incorrect results, likely because of failure in the torch.onnx.export function. Addressing these errors is beyond the scope of this paper, hence we focus on performance analysis of verified ONNX graphs only.

Experiments on x86. Figure 3 shows our results for the ten verified models inferred on the x86 Intel CPU, comparing the speedups of ORT and OML normalized to PyTorch. The execution time of 10 runs was averaged to ensure consistent results. ORT-default consistently outperforms PyTorch and PyTorch JIT, achieving on average 45% faster execution time compared to PyTorch, due to the use of hand-tuned libraries and dedicated Execution Providers optimized for specific hardware platforms (see Section 3). Both PyTorch and ORT are multi-threaded by default and disabling parallelization requires building PyTorch and ORT from source. We did not attempt this in a bid to evaluate the best performance achievable with mainstream frameworks (PyTorch and ORT in our study). PyTorch JIT (not shown in the graph) offers speedups of 30% over the eager mode PyTorch execution, for compatible models such as AlexNet and MobileNet. However, PyTorch JIT may return incorrect results

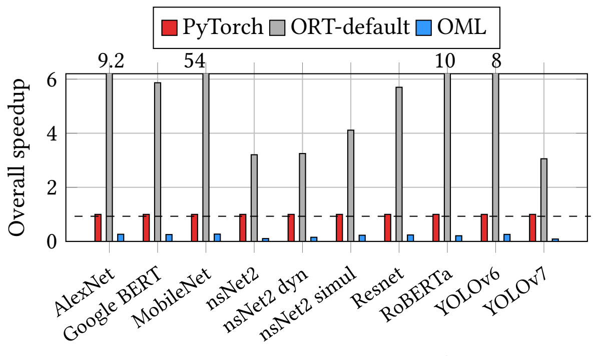

Figure 3: Speedup of ORT-default and OML (single-threaded) on x86 normalized to PyTorch

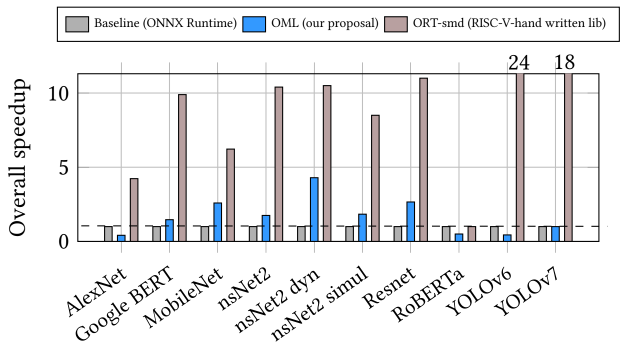

Figure 4: Speedup of OML and ORT-SMD on RISC-V normalized to ORT-default

for more complex models with variable input tensor dimensions, such as Google BERT, RoBERTa, and all nsNet2 versions [25]. Both PyTorch and ORT use Python as a high-level wrapper for user interaction. PyTorch and ORT leverage C++ for their core operations to ensure high performance through hardware-specific optimizations. Libraries such as libTorch [31] and MklDNN are implemented in C++, with certain kernels, e.g., the single precision softmax operation directly invoking assembly code in ORT for faster execution. In contrast, our approach uses LLVM out-of-the-box optimizations (-O3 at both the opt and llc levels). It is a single-threaded execution and does not use any hand-tuned libraries like MKD-DNN or MLAS. Consequently, it is difficult to surpass the performance of PyTorch and ORT-default, which have been fine-tuned and multi-threaded on the x86 architecture.

In AlexNet, ORT fuses the General matrix multiply (GEMM) layer with the adjacent activation layer ReLU by identifying adjacent layers that can be fused and generating a new, optimized ONNX graph accordingly. ORT utilizes partial loop unrolling for GEMM and convolution layers with a fixed unrolling factor of 8 for double and single precision to exploit better SIMD AVX-512 vectorization. In contrast, PyTorch's default eager execution mode uses pragma hints to guide vectorization. For the GEMM layer, PyTorch combines loop unrolling, and nested loop splits to enhance vectorization [32]. Additionally, PyTorch also employs template-driven partial loop unrolling [33]. PyTorch's convolution operations search for unary operator fusion, for instance, fusing convolution with ReLU. In addition to this default mode, PyTorch-2 [1] introduced TorchDynamo as the JIT compiler for PyTorch and TorchInductor as the default compiler backend. PyTorch JIT compiler works by tracing PyTorch models into an intermediate representation (FX graph) that can be optimized and executed more efficiently using graph optimization techniques such as constant folding, dead code elimination, and operator fusion. The scheduling phase of TorchInductor does operator fusion, reordering kernels, and bufferization. These optimizations help PyTorch JIT to achieve an average speedup of 30% compared to eager mode execution. However, this speed-up is observed after a couple of warm-up runs for inputs with the same shape. PyTorch JIT needs warm-ups to fine-tune optimizations such as the fusion of elementwise and pointwise operations and tensor reduction operations.

Drawing inspiration from the optimized libraries of PyTorch and ORT, we can integrate these optimizations into the MLIR and LLVM Compiler. These optimizations can be extended to various layers found in RNNs, LSTM, and Transformer models. Consequently, these optimizations will apply to all hardware platforms and neural network kernels, eliminating the need for manual rewrites.

Experiments on RISC-V. Figure 4 shows our results comparing: (1) ORT: the open-source version of ORT, cross complied for a RISC-V architecture; (2) OML, compiled using our compiler pipeline presented in Section 2, and (3) ORT-SMD: ORT ported to RISC-V by Semidynamics [41] with a developed and optimized Execution Provider (EP) that takes advantage of Semidynamics' vector and tensor units [43, 42]. More than 40 ONNX operators are already supported and optimized for the Semidynamics hardware [2]. Due to the lack of available libraries for building PyTorch on RISC-V [7], PyTorch performance could not be evaluated on this architecture.

For most neural networks, the performance of OML falls between the open-source ORT-default, and the ORT-SMD manually optimized version. Unlike for x86, there is less fine-tuning for RISC-V for the ORT-default, thus OML is almost on par. ORT-SMD brings hand-tuned optimizations, but only for selected operators for instance GEMM, convolution, and matmul. The OML compiler pipeline (1) exposes both MLIR and LLVM intermediate representations and (2) leverages the LLVM back-end. First, MLIR and LLVM IR exposure enables high-level, target-agnostic compile-time optimizations. Second, relying on the LLVM back-end brings in dedicated hardware-specific optimizations. Moreover, the LLVM back-end can be extended to support emerging hardware, benefiting from all prior optimizations.

Discussion. We found that PyTorch and ORT are highly optimized for layers such as convolution and matrix multiplication, which are predominant and are bottlenecks in traditional neural networks such as AlexNet and MobileNet. However, there are fewer optimizations for layers found in RNN and RNN-based LSTM, and Transformer models, such as tanh activation, layer normalization, encoders, decoders, and Gated Recurrent Unit (GRU) [44]. These layers are becoming increasingly popular with the rise of Transformers and will be an immediate target for automatic optimizations.

5 Case study: nsNet2

In this section, we analyze the use case of nsNet2 [4, 26], a model specifically designed for edge devices such as earphones. It is developed for real-time denoising, necessitating compiler optimizations. We studied three versions: (1) nsNet2 baseline including recurrent

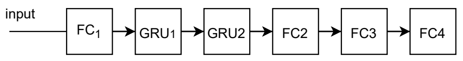

Figure 5: nsNet2 baseline (nsNet2) topology

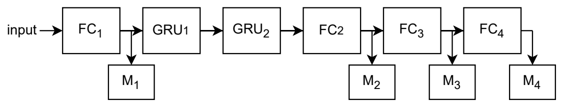

Figure 6: nsNet2 dynamic layerwise (nsNet2 dyn) topology

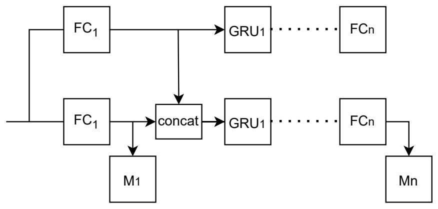

Figure 7: nsNet2 dynamic simultaneous extralayers (nsNet2 simul) topology

(GRU) and fully-connected (FC) layers: FC-GRU-GRU-FC-FC-FC as shown in Figure 5. Each fully-connected layer is followed by a ReLU activation, subsetting the layers' activations to yield a mask. The last layer is an exception, featuring a sigmoid nonlinearity. (2) nsNet2 dynamic layerwise (nsNet2 dyn), with exit stages after each fully-connected (FC) layer, and with each exit-stage (M) trained one after the other as shown in Figure 6, (3) nsNet2 dynamic simultaneous extra-layers (nsNet2 simul), with exit-stages (M) after each layer and additional data paths in the form of duplicate layers, and all exit stages are trained jointly as shown in Figure 7. These versions provide varying input dimensions and different numbers of GRU and matmul layers with variable sizes. We present the optimizations of ORT and comment on the optimizations and limitations of the LLVM compiler. We used the PyTorch profiler [34] and vtune [35] to retrieve thread information and PyTorch library calls for x86. No similar profiling tools are available for RISC-V, which prevented us from profiling on this architecture.

On x86, 32 threads were created for the nsNet2 baseline execution and 35 threads for the nsNet2 dyn and nsNet2 simul, respectively. PyTorch uses inter- and intra-operators level multi-threading and other optimizations as discussed in Section 3.

We analyzed the LLVM vectorization pass reports for the nsNet2 baseline model and found that 37 loops were considered for vectorization and only 11 were vectorized. Among the top reasons that prevented vectorization were: (i) unsafe floating-point operations (even with `ffastmath` flag), (ii) scalar cost being optimal, and (iii) unvectorizable types such as function calls in the loop body. We expect that through specific optimizations such as loop fusion, partial loop unrolling, and effective vectorization at both the MLIR and LLVM IR levels, we will be able to further increase the performance of nsNet2 (and neural networks in general). In the nsNet2 use case, the GRU layer is computationally intensive and time consuming. In the ORT-default the GRU internally uses the GEMM function, the only optimized function. With GEMM no longer being the bottleneck, other computations such as the activation functions and for loops to compute bias addition become the new bottleneck during GRU layer execution. It is challenging to *manually* incorporate all optimizations discussed in sections 3 and 4 due to the rapid advancements in AI research and hardware research. Conversely, using MLIR and LLVM, these optimizations can be automated and extended to future developments, both in software and hardware.

Table 1: Speedups summary: GRU and matmul in nsNet2 on RISC-V core Atrevido 423

Model	GRU	Matmul	GRU speedup	Matmul speedup	overall speedup
nsNet2 baseline	2	4	1.4	1.2	1.7
nsNet2 dyn	2	4	1.8	1.3	4.2
nsNet2 simul	4	7	1.9	1.25	1.9

Figure 4 shows our results for nsNet2 comparing ORT-default, ORT-SMD, and OML approach, illustrating the performance for the three versions of nsNet2. OML achieves a maximum speedup of 4 for the nsNet2 dyn, significantly outperforming ORT-default. Table 1 shows the number of GRU and Matmul layers in each nsNet2 use case, and their speedups over ORT-default on Atrevido 423 RISC-V core. Although nsNet2 simul has more GRU layers, leading us to expect the highest speedup due to OML's optimization of GRU, the performance gains were limited to 1.9 due to the fine-tuned performance of ORT-default in neural network kernels such as Matmul, GEMM etc. Speedups achieved by manually fine-tuned ORT-SMD highlight the potential for optimization in the ORT-default and the OML. We aim to automatically incorporate similar optimizations of ORT-SMD in the LLVM compiler.

6 Conclusion and future work

This paper studied the optimizations offered by state-of-the-art frameworks such as PyTorch and ORT. We highlighted the significant performance gap between manually fine-tuned libraries and the proposed pipeline OML on x86 architectures. Our findings indicate that neural network libraries for RISC-V are very immature, with the proposed OML mostly outperforming the default implementations. While manual fine-tuning shows promise for optimizing performance on RISC-V, the process is labor-intensive and not scalable. Therefore, our goal is to automate these optimizations through the MLIR and LLVM compiler, bridging the gap and smooth integration of neural network inference on RISC-V architectures.

Acknowledgments

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 (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.

References

[1] Jason Ansel et al. 2024. Pytorch 2: faster machine learning through dynamic python bytecode transformation and graph compilation. In (ASPLOS '24). Association for Computing Machinery, La Jolla, CA, USA, 929–947. ISBN: 9798400703850. DOI: 10.1145/3620665.3640366.

[2] 2023. Atrevido (online). (2023). Retrieved November 4, 2024 from https://semidynamics.com/en/products/atrevido.

[3] 2016. Bertforsequenceclassification (online). (2016). Retrieved November 4, 2024 from https://huggingface.co/docs/transformers/v4.41.3/en/model_doc/bert#transformers.BertForSequenceClassification.

[4] Sebastian Braun and Ivan Tashev. 2020. Data augmentation and loss normalization for deep noise suppression. In *Speech and Computer*. Alexey Karpov and Rodmonga Potapova, (Eds.) Springer International Publishing, Cham, 79–86. ISBN: 978-3-030-60276-5.

[5] Alessandro Cerioli, Riccardo Miccini, Clément Laroche, Tobias Piechowiak, Luca Pezzarossa, Jens Sparsø, and Martin Schoeberl. 2024. Neuralcasting: a front-end compilation infrastructure for neural networks. In *2024 11th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 161–168.

[6] Iacopo Colonnelli, Robert Birke, and Marco Aldinucci. 2023. Experimenting with pytorch on risc-v. In RISC-V Summit Europe 2023. Barcelona, Spain. https://hdl.handle.net/2318/1925291.

[7] Iacopo Colonnelli, Robert Birke, and Marco Aldinucci. 2023. 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).

[8] 2017. Cpu info pytorch dependency library (online). (2017). Retrieved November 4, 2024 from https://github.com/pytorch/cpuinfo.

[9] Marat Dukhan. 2019. The indirect convolution algorithm. In arXiv: 1907.02129.

[10] Marat Dukhan and Artsiom Ablavatski. 2020. The two-pass softmax algorithm. *CoRR*, abs/2001.04438. https://arxiv.org/abs/2001.04438 arXiv: 2001.04438.

[11] 2024. The et-soc-1 ai / hpc accelerator chip (online). (2024). Retrieved November 4, 2024 from https://www.esperanto.ai/products/.

[12] 2023. Pytorch: an ever-growing ai framework among ai practitioners (online). (2023). Retrieved November 4, 2024 from https://www.huawei.com/sg/news/sg/2023/pytorch-an-ever-growing-ai-framework-among-ai-practitioners.

[13] Matheus Fellype Ferraz, Birte Kristina Friesel, and Olaf Spinczyk. 2024. Pros and cons of executable neural networks for deeply embedded systems. In *Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI* (CODAI '23). Association for Computing Machinery, Hamburg, Germany, 16–20. ISBN: 9798400703379. DOI: 10.1145/3615338.3618118.

[14] Hasan Genc et al. 2021. Gemmini: enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*.

[15] 2024. Iree-riscv cross compile completely infeasible (online). (2024). Retrieved November 7, 2024 from https://github.com/iree-org/iree/issues/19057.

[16] 2019. Iree: intermediate representation execution environment. (2019). Retrieved November 4, 2024 from https://github.com/openxla/iree.

[17] 2023. Kuleuven-micas. (2023). Retrieved November 4, 2024 from https://github.com/KULeuven-MICAS/snax_cluster.

[18] C. Lattner and V. Adve. 2004. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. DOI: 10.1109/CGO.2004.1281665.

[19] Chris Lattner et al. 2020. Mlir: a compiler infrastructure for the end of moore's law. (2020). arXiv: 2002.11054 [cs.PL].

[20] Tung D. Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E. Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. 2020. Compiling onnx neural network models using mlir. *ArXiv*, abs/2008.08272. https://api.semanticscholar.org/CorpusID:221173137.

[21] Chen Liu, Matthias Jobst, Liyuan Guo, Xinyue Shi, Johannes Partzsch, and Christian Mayr. 2024. Deploying machine learning models to ahead-of-time runtime on edge using microtvm. In *Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI* (CODAI '23). Association for Computing Machinery, Hamburg, Germany, 37–40. ISBN: 9798400703379. DOI: 10.1145/3615338.3618125.

[22] Hsin-I Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. 2022. Tinyiree: an ml execution environment for embedded systems from compilation to deployment. *IEEE Micro*, 42, 5, 9–16. DOI: 10.1109/MM.2022.3178068.

[23] Qiankun Liu, Sam Amiri, and Luciano Ost. 2024. Exploring risc-v based dnn accelerators. In *2024 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, 1–6. DOI: 10.1109/COINS61597.2024.10622495.

[24] 2019. Compiling llvm ir to binary (online). (2019). Retrieved November 7, 2019 from https://borretti.me/article/compiling-llvm-ir-binary.

[25] Nuno P. Lopes. 2023. Torchy: a tracing jit compiler for pytorch. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (CC 2023). Association for Computing Machinery, Montréal, QC, Canada, 98–109. ISBN: 9798400700880. DOI: 10.1145/3578360.3580266.

[26] Riccardo Miccini, Alaa Zniber, Clément Laroche, Tobias Piechowiak, Martin Schoeberl, Luca Pezzarossa, Ouassim Karrakchou, Jens Sparsø, and Mounir Ghogho. 2023. Dynamic nsNET2: Efficient Deep Noise Suppression with Early Exiting. en. In *2023 IEEE 33rd International Workshop on Machine Learning for Signal Processing (MLSP)*. IEEE, Rome, Italy, (Sept. 2023), 1–6. ISBN: 9798350324112. DOI: 10.1109/MLSP55844.2023.10285925.

[27] 2016. Ntel mkl-dnn (online). (2016). Retrieved November 4, 2024 from https://github.com/intel/mkl-dnn.

[28] 2025. Ntel mkl-dnn (online). (2025). Retrieved January 4, 2025 from https://github.com/CAPS-UMU/OML.

[29] 2024. Onnx-mlir risc-v support issue (online). (2024). Retrieved November 4, 2024 from https://github.com/onnx/onnx-mlir/issues/2824.

[30] 2019. Compile using onnx-mlir (online). (2019). Retrieved December 4, 2021 from https://github.com/onnx/onnx-mlir/blob/main/docs/mnist_example/README.md.

[31] 2018. Libtorch (online). (2018). Retrieved November 4, 2024 from https://github.com/pytorch/pytorch/blob/main/docs/libtorch.rst.

[32] 2018. Sample pytorch code for nested loop split (online). (2018). Retrieved November 4, 2024 from https://github.com/intel/mkl-dnn/blob/38afc5d8c9049f650031ebcb165485703f2e5deb/src/cpu/gemm/f32/ref_gemm_f32.cpp#L50-L74.

[33] 2018. Sample for template based loop unrolling (online). (2018). Retrieved November 4, 2024 from https://github.com/intel/mkl-dnn/blob/38afc5d8c9049f650031ebcb165485703f2e5deb/src/cpu/gemm/f32/ref_gemm_f32.cpp#L53-L56.

[34] 2020. Pytorch profiler (online). (2020). Retrieved November 4, 2024 from https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[35] 2022. Vtune profiler (online). (2022). Retrieved November 4, 2024 from https://www.intel.com/content/www/us/en/docs/vtune-profiler/get-started-guide/2024-0/overview.html.

[36] 2016. Alexnet pretrained model (online). (2016). Retrieved November 4, 2024 from https://pytorch.org/hub/pytorch_vision_alexnet.

[37] 2016. Pytorch vision mobilenet v2 (online). (2016). Retrieved November 4, 2024 from https://pytorch.org/hub/pytorch_vision_mobilenet_v2.

[38] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: a sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 58–70. DOI: 10.1109/HPCA47549.2020.00015.

[39] 2022. Resnet 101 (online). (2022). Retrieved November 4, 2024 from https://pytorch.org/vision/main/models/generated/torchvision.models.resnet101.html.

[40] 2016. Roberta facebookai/roberta-base (online). (2016). Retrieved November 4, 2024 from https://huggingface.co/docs/transformers/en/model_doc/roberta.

[41] 2023. Semidynamics (online). (2023). Retrieved November 4, 2024 from https://semidynamics.com/en/.

[42] 2023. Semidynamics (online). (2023). Retrieved November 4, 2024 from https://semidynamics.com/en/technology/tensor-unit.

[43] 2023. Semidynamics (online). (2023). Retrieved November 4, 2024 from https://semidynamics.com/en/technology/vector-unit.

[44] Tristan Stérin, Nicolas Farrugia, and Vincent Gripon. 2017. An Intrinsic Difference Between Vanilla RNNs and GRU Models. In *COGNITIVE 2017 : Ninth International Conference on Advanced Cognitive Technologies and Applications*. Athènes, Greece, (Feb. 2017), 76–81. https://hal.science/hal-01522887.

[45] Jialiang Tan, Yu Chen, Zhenming Liu, Bin Ren, Shuaiwen Leon Song, Xipeng Shen, and Xu Liu. 2021. 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). Association for Computing Machinery, Athens, Greece, 1117–1128. ISBN: 9781450385626. DOI: 10.1145/3468264.3468541.

[46] 2020. Torch-mlir: the torch-mlir project. (2020). Retrieved November 4, 2024 from https://github.com/llvm/torch-mlir.

[47] 2019. Xnnpack github repository (online). (2019). Retrieved November 4, 2024 from https://github.com/google/XNNPACK.

[48] 2022. Yolov6 (online). (2022). Retrieved November 4, 2024 from https://github.com/meituan/YOLOv6/tree/v3.

[49] 2022. Yolov7 github repository (online). (2022). Retrieved November 4, 2024 from https://github.com/WongKinYiu/yolov7.

[50] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2021. Snitch: a tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Transactions on Computers*, 70, 11, 1845–1860. DOI: 10.1109/TC.2020.3027900.

[51] Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. 2022. Quantifying the interpretation overhead of python. *Sci. Comput. Program.*, 215, C, (Mar. 2022), 27 pages. DOI: 10.1016/j.scico.2021.102759.