RISC-V Meets TFLite Micro: Insights from MLCommons Tiny Benchmark

Fabrizio Maria Aymone
Politecnico di Milano
Milan, Italy
fabrizio.aymone@gmail.com

***Abstract*—This paper explores the deployment of TensorFlow Lite for Microcontrollers (TFLite Micro) on RISC-V architectures, focusing on the MLCommons Tiny benchmark. Given the rising importance of TinyML, integrating TFLite Micro with the open-source RISC-V architecture can significantly enhance the performance and scalability of edge devices. This integration not only promotes the use of customizable, open-source solutions but also fosters innovation in hardware design tailored for specific machine learning workloads. The study details the cross-compilation process for obtaining RISC-V 32 implementations of the benchmark models and evaluates their performance using Spike and Gem5 simulators. Key metrics such as CPI and number of branch mispredictions are analyzed to understand the efficacy of in-order and out-of-order core designs. Our findings reveal that the simpler architecture of in-order cores demonstrates comparable performance to out-of-order due to the parallelizable nature of TinyML workloads, offering insights for optimizing RISC-V processors in resource-constrained environments.**

***Index Terms*—RISC-V, TFLite Micro, Spike, Gem5, Out-of-order, MLCommons-tiny.**

I. Introduction

The unstoppable growth of Internet of Things (IoT) tiny devices has pervaded everyone daily lives and revolutionized entire industries ranging from automotive, to consumer and to smart. At the same time, the super-human performance achieved in the last decade by Deep Learning in tasks such as image recognition [1], [2], natural language processing [3], [4] and speech recognition [5], has drastically changed the entire landscape of computing and signal processing. In such scenario, deploying neural networks (NN) on edge devices signifies unlocking a plethora of disruptive applications. Edge devices, by processing data locally rather than relying on cloud computing, can offer reduced latency, improved privacy, and lower bandwidth usage. However, the micro-controllers (MCUs) and sensors are severely resource-constrained, due to their limited embedded memory and compute capabilities. The challenging goal of running NNs on such devices has reunited across the years experts from industry and academia, giving rise to the vibrant field of TinyML. Thanks to the collaboration of leading tech-companies and the contributions from the community, a whole software and hardware ecosystem has been gradually developed. Within the latter, TensorFlow Lite for Microcontrollers (TFLite Micro) [6] has become the standard choice as library for inference at the edge. The robust support and continuous updates from the TensorFlow community ensure that TFLite Micro remains at the forefront of TinyML developments. Recently, there has been increasing interest in the RISC-V architecture [7], [8] due to its open-source nature, flexibility, and potential for customization. RISC-V offers advantages such as reduced power consumption, improved performance, and scalability, making it highly suitable for embedded systems and IoT devices. Therefore, providing a RISC-V implementation of TFLite Micro is pivotal in leveraging these advantages and further advancing the capabilities of TinyML on diverse hardware platforms. By combining these cutting-edge technologies, we aim to address some of the most pressing challenges in the field of edge computing. This paper addresses the MLCommons-Tiny benchmark [9] by providing a RISC-V implementation for the models considered via the TFLite Micro C++ API and profiles them. This paper is organized as follows: section II cites the main related works in the known literature; section III describes the cross-compilation process for obtaining the RISC-V 32 implementation of the models in the MLCommons-Tiny benchmarks; section IV reports the results of the profiling conducted with Spike and Gem5; section V concludes the paper by proposing further and future developments. Our objective is not only to present the current state of research but also to inspire further exploration and innovation in this rising field.

II. Related works

In literature, there are relatively few works specifically focusing on TFLite implementations for RISC-V architectures. One notable contribution is the work titled ”Towards Deep Learning using TensorFlow Lite on RISC-V” [10]. In the latter, to effectively accelerate ML on RISC-V processors, ISA extensions derived from the RISC-V vector ISA proposal are introduced. A tool-chain is then created by augmenting the software environment with the right inline assembly support and building the run-time that can effectively map the high-level macros to the low-level ISA execution. Basic compiler support for the extended instructions using C inline assembly functions is then added. The C inline assembly functions are used to implement TensorFlow Lite kernel operations such as convolution and matrix multiplication. These optimized functions are added to TensorFlow Lite source code and cross-

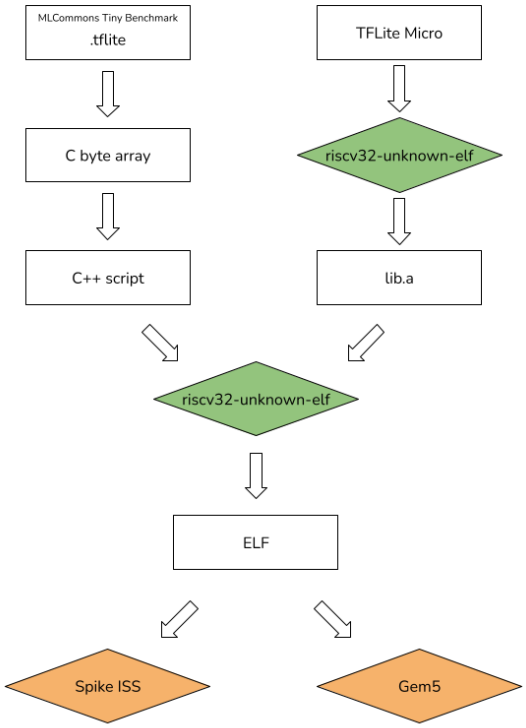

Fig. 1: Cross-compilation and profiling flowchart

compiled them RISCV target. The Spike instruction set simulator (ISS) is modified to support the extended instructions. Subsequently, Spike is used for functional verification and for retrieving the number of instructions committed for a selected benchmark of machine learning models. "An end-to-end RISC-V solution for ML on the edge using in-pipeline support" [11] continues the previous work by discussing the micro-architecture implementation of a machine learning processing unit, RV-MLPU, to support the execution of the vectorized machine learning kernels in the pipeline. These papers differ from the contributions of this work. First of all, they target TFLite, while this work adopts TFLite Micro. The two are different libraries and should not be confused as they are not compatible. Secondly, they benchmark using Spike ISS by collecting the number of instructions committed. Being a simple ISS, Spike does not implement any processor architecture model; therefore, it cannot provide number of cycles and other relevant feature. This work, on the other hand, uses Gem5 to conduct a more complete and accurate profiling, obtaining metrics such as number of cycles and CPI. Moreover, this study takes a holistic approach by examining both in-order and out-of-order core designs, providing a more comprehensive understanding of their respective performances in the context of TinyML workloads.

III. Cross-compilation

The MLCommons Tiny benchmark [9] is composed by four different industry standard workloads. They consists in three supervised tasks (keyword spotting, visual wake words, image classification) and one unsupervised task (anomaly detection). The NNs adopted are respectively DS-CNN [12], MobileNet [13], ResNet [14], which are Convolutional Neural Networks (CNNs), and AutoEncoder (AE) [15], which is Fully Connected (FC). The precise structure and the source code of these models are publicly available in a github repository [16]. Specifically, the fp32 tflite files were considered for each model[1]. In order to use the tflite models in our C++ scripts, it is first needed to convert them into C byte arrays. This process can be done by using the command `xxd -i model.tflite > model.cc`. The model file containing the C byte array will then be included in the C++ script to be processed leveraging TFLite Micro C++ API. In order to obtain a functioning RISC-V version of the latter, the structure of the TFLite Micro repository [17] was thoroughly studied and the Makefile which generates the static compiled library .a was identified in `tensorflow/lite/micro/tools/make/Makefile`.

[1]Beware that at the time of writing the DS-CNN tflite `kws_ref_model_float32.tflite` in the repository [16] has been generated as a hybrid model, namely includes a mix of int8 and fp32 operation.

Understanding the repository structure and the build process is crucial for successful cross-compilation and integration with RISC-V tools. In the first lines of the Makefile the CXX_TOOL, CC_TOOL, AR_TOOL variables are initialized to the standard gcc toolchain. By assigning to these variables the corresponding tool for our toolchain, i.e. `riscv32-unknown-elf`, and calling make, the .a library is generated in the `gen` folder. Then, the C++ scripts written by us for calling the model and performing an inference are compiled linking the previously generated static library. In this way, ELF files are obtained and are ready to be profiled. The cross-compilation process is reported in Fig. 1.

IV. Profiling

When it comes to profiling performance, two different tools are adopted: Spike and Gem5. The first is an ISS, which does not implement any microprocessor architecture model. Therefore, it will be used to validate the results of the inference of the model and retrieve the number of instructions committed. The second tool is Gem5, a highly flexible and modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture. Specifically, it provides detailed insights into microarchitectural behaviors such as cache hits and misses, pipeline stalls, branch predictions, and execution latencies. The C++ scripts introduces in the Cross-compilation section are actually two: one for the Spike simulation and one for Gem5.

TABLE I: CSR performance registers via Spike

Model	Tensor Arena	# committed instructions[2]
AE	4,7 KB	1,6 M
ResNet	60 KB	148,5 M
DS-CNN	23 KB	44,7 M
MobileNet	322 KB	114,4 M

V. Spike Functional Simulation

To execute the Spike simulation, the rv32gc isa-version was specified and to handle syscalls (newlib) pk proxy kernel was used in conjunction. In the C++ code written for Spike the memory consumption of TFLite is profiled. TFLite works by statically instantiating a char array, referred to as Tensor Arena, where during runtime all tensors will be allocated. Therefore, it is essential to size this array to be larger than the runtime tensor peak memory used. TFLite Micro API offers the class `RecordingMicroAllocator` to record all the tensor allocations during runtime and find the minimum size for the array. The C++ scripts intended to be used for Spike print the output of this recording so that it is possible to correctly size the char array. Furthermore, an inference for a single sample stored as an array is performed. After instantiating the model and loading the weights and the input tensor, the inference is performed by calling the method `Invoke()` and the output is compared with the one obtained using tflite in python, in order to verify the correctness of the implementation. As the inference workload takes place in the `Invoke()` method, this will be the Region Of Interest (ROI) of the performance profiling. To retrieve the number of instructions committed in the ROI it is necessary to read the `instret` CSR register before and after the considered workload. The difference between these values is then computed to obtain the instructions committed. To do so, the C inline assembly code for reading the CSR register, namely `rdinstret` instruction, is inserted before and after the `Invoke()` method. By reading the source code of Spike, it was discovered that to abilitate the `instret` CSR register it was necessary to specify to Spike the Zicntr extension in the isa flag, even if rv32gc should already include the ability to read such register. As Spike does not implement any hardware model, the value of the `instret` register is by settings the same of `cycle`. In Table I, the Arena allocation size registered by TFLite and the number of committed instructions are reported for each of the four models. It can be observed that the relation between the memory size of the tensors required by the model (activations of layers + weights) and the instructions needed to perform inference is complex to identify. For example, the ResNet requires a Tensor Arena whose size is five times less than the one required by MobileNet. Nevertheless, Resnet commits 30M more instructions than MobileNet. The factors that affect this results are various, where the morphology of the network is the most preponderant. the MobileNetV1 [13] is designed to feature DepthWise convolutions featured by PointWise convolutions, which drastically reduces the number of operations involved with respect to normal cross-channel convolutions. This results in less multiplies and accumulate (MACCs); hence, less instructions committed. Tensor Arena required is only increased by using Depthwise and Pointwise convolutions, as two layers are used instead of one in the case of plain convolution. The number of instructions committed was later compared with the stats obtained with Gem5, obtaining the same number.

```
// Starting Region of Interest (ROI)
m5_work_begin(1,1);

interpreter.Invoke();

// End Region of Interest (ROI)
m5_work_end(1,1);
```

Listing 1: Gem5 ROI profiling

VI. Gem5 Simulation

Using the Gem5 tool, two different cores, one in-order and one Out of Order (OoO), are simulated. To keep the comparison fair, the two cores were designed, when possible, with the

[2]It is equal to the number of committed instructions obtained via Gem5.

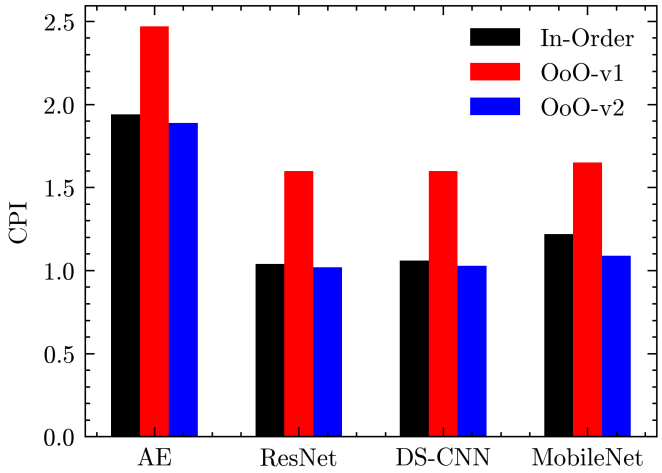

Fig. 2: CPI for Tiny MLCommons on RISC-V simulated cores

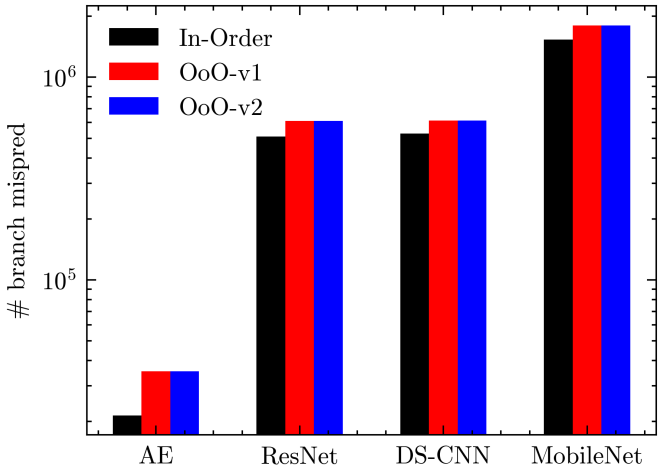

Fig. 3: Mispredected branches for Tiny MLCommons on RISC-V simulated cores

same specs. About memory the two cores were configured with 32KB L1 I$ eight-way set-associative cache , 32KB L1 D$ eight-way set-associative, a 256KB L2$ eight-way set-associative cache and a 8GB DDR3 DRAM. All caches use Least Recently Used (LRU) policy. Both cores feature 3 integer ALUs with latency 1 clock, 3 integer multipliers/dividers with latency 3 clocks for multiplication and 8 clocks for division, 3 floating-point ALUs with latency 4 clocks and 3 floating-point multipliers/dividers with latency 4 clocks. A tournament branch predictor was also implemented. The two cores are running at 1GHz. The in-order core was constructed using gem5 MinorCPU class, an in-order processor model with a fixed four-stage pipeline (Fetch1, Fetch2, Decode, Execute) but configurable data structures and execute behaviour. It is intended to be used to model processors with strict in-order execution behaviour. On the other hand, the OoO core was constructed using the gem5 O3CPU class, which models an out-of-order processor with a highly detailed and flexible five-stage pipeline (Fetch, Decode, Rename, Issue/Execute/Write-back, Commit) that supports dynamic instruction scheduling, allowing instructions to be executed as soon as their operands are ready, rather than strictly following the program order. To profile only the Region Of Interest constituted by the call to the `Invoke()` method, md5 functions for ROI profiling were used and the code was linked to md5 library during compilation. The delimiters for profiling are illustrated in Listing 1. Two different versions of OoO cores are considered on the basis of the number of cycle delays introduced during the Decode-to-Rename stage and Rename-to-IEW stage. The first version denoted as OoO-v1 considers a Decode-to-Rename delay of one cycle and Rename-to-IEW delay of two cycles. The second version denoted as OoO-v2 considers for both Decode-to-Rename and Rename-to-IEW delays of zero cycles.

Simulating the three cores on the four inference workload of Tiny MLCommons via Gem5, several metrics are obtained. Figure 2 shows the Clocks per Instructions (CPI) for the in-order and Out of Order core.The results for the CPI values across the four workloads indicate that the in-order core consistently achieves better performance metrics than the OoO-v1 and comparable performance to OoO-v1. For AE, the in-order core registers a CPI of 1.94, whereas the OoO-v1 core has a higher CPI of 2.47 and the OoO-v2 core 1.89. Similarly, for ResNet, the in-order core shows a CPI of 1.04, compared to the OoO-v1 core's 1.6 and the OoO-v2 core's 1.02. In DS-CNN, the in-order core achieves a CPI of 1.06, similarly to the OoO-v2 core's 1.03, outperforming the OoO-v1 core, which records a CPI of 1.6. Finally, in MobileNet, the in-order core again demonstrates superior performance to the OoO-v1 core with a CPI of 1.22, while the OoO-v1 core shows a CPI of 1.65 and the OoO-v2 core a CPI of 1.09, outperforming all other cores. First of all, effectuating a comparison across the nets, AE shows superior number of clocks per instructions with respect to the other three networks. The reason for such result probably lies in the radically inferior (10x-100x compared to the other models) number of committed instructions (less instructions limit the ability of the processor to amortize the number of possible stall cycles) combined with the morphology of the neural network. The AE is, in fact, the only Fully Connected Network of the benchmark (the other three are CNNs). The calculations for a Dense layer correspond to two nested for loops, which will inevitably result in a certain number of branch mispredictions, which will affect the final CPI by an amount inversely proportional to the instruction executed. On the other hand, Convolutional layers are constructed by six nested for loops and involve much more MACCs operations, rather than a Dense layer, as for each output channel all the channels of the previous layer are multiplied by the kernel. If a sophisticated branch predictor is adopted, the effect of the additional clocks caused by the higher number of for loops, and consequently branch instructions, on the CPI is compensated by the higher number of instructions. Secondly, the reasons behind the better performance of the In-Order core compared to the OoO-v1 are

several. First of all, some fundamental considerations should be made. The nature of machine learning workloads are matrix multiplications, which are independent, parallelizable and don't show any dependency hazard. The benefits brought by the resolution of hazards performed by out-of-order execution are, therefore, not evident in such a workload. In such scenario, the burden of the additional complex logic circuitry needed to handle reorder buffers, register renaming, load-store queues, etc. outweighs the unnoticeable positive effects of out-of-order execution. Alternatively, a minimal in-order pipeline proves to be more effective by saving unnecessary clocks. In particular, by reducing to zero the Decode-to-Rename delay and Rename-to-IEW delay of the OoO-v1 core, the OoO-v2 core achieves CPI very similar to the In-Order core or superior like in the case of MobileNet. Ablation studies were effectuated tuning other architecture parameters like the number of entries in ROB, the size of the LSQ etc., but they didn't sort any effect on the CPI. Our study points out that the delays relative to register renaming stage have a crucial effect on performance when dealing with ML workloads and when comparing with in-order versions.

Figure 3 reports on a logarithmic scale the number of committed branch mispredictions for the in-order and two versions of Out of Order cores. For the in-order core, the number of committed branch mispredictions are 21 million for AE, 510 million for ResNet, 526.8 million for DS-CNN, and 1.5 billion for MobileNet. For OoO-v1 and OoO-v2, the committed branch mispredictions are 35 million for AE, 608.9 million for ResNet, 610.5 million for DS-CNN, and 1.8 billion for MobileNet. As expected, Resnet, DS-CNN and MobileNets show a higher number of branch mispredictions compared to AE as they have by architecture more branch instructions, consequently being more susceptible to mispredictions. The observations and hypothesis that were made during the analysis of the CPI results, on the disparity between AE and the CNNs are corroborated by this data. Taking as a metric the ratio of mispredicted brances on instructions commited, it is obtained for the in-order core 1.4% for AE, 0.3% for ResNet, 1.1% for DS-CNN and 1.3% for MobileNet. Furthermore, it is possible to notice that in-order processor shows less mispredictions than the two Out of order processors. The reason for that are manifold and surely the combination of aggressive speculative execution, complex control flow and dynamic scheduling plays an important role. Also, in this case the in-order processor demonstrates to be the most suitable for such machine learning workload.

TABLE II: # ROB reads

Model	AE	ResNet	DS-CNN	MobileNet
OoO-v1	5 M	342 M	104 M	271 M
OoO-v2	3.9 M	242 M	74 M	195 M

In Table II and III, the number of ROB reads and ROB writes are reported. As shown, the number of ROB writes is nearly the same for both OoO-v1 and OoO-v2, as expected by considering that the program executed is the same and the architecture is almost the same except for rename-stage delays. On the other hand, OoO-v1 features a greater number of ROB reads w.r.t. OoO-v2 by approximately 30% for AE, 41 % for ResNet, 40.5% for DS-CNN and 39% for MobileNet. The reason for the greater amount of ROB reads should be attributed to the greater amount of clock cycles (hence, greater CPI) of OoO-v1. As OoO-v1 spends more clock cycles per instructions, the number of reads to the ROB entries to check their status, dependencies, or when they are about to retire is more. In table IV, the number of renamed instructions are reported. The data shows to be slightly more than the number of instructions committed. This is due to the fact that renamed instructions count also speculative instructions that are renamed, but are not committed eventually due to branch mispredictions.

TABLE III: # ROB writes

Model	AE	ResNet	DS-CNN	MobileNet
OoO-v1	3.2 M	299 M	90 M	230.6 M
OoO-v2	3.2 M	298 M	90 M	230.2 M

TABLE IV: # renamed instructions

Model	AE	ResNet	DS-CNN	MobileNet
OoO-v1/-v2	1.6 M	150.3 M	45 M	115 M

VII. Conclusion

This paper presented a comprehensive evaluation of running TensorFlow Lite for Microcontrollers (TFLite Micro) on RISC-V architectures using the MLCommons Tiny benchmark. Through meticulous cross-compilation and profiling with Spike and Gem5, we examined the performance of both in-order and out-of-order cores. The results indicate that in-order cores achieve comparable or superior performance to their out-of-order counterparts in terms of Clocks Per Instruction (CPI) for TinyML workloads. This is largely due to the independent, parallelizable nature of matrix multiplications that characterize machine learning tasks, which do not significantly benefit from the hazard resolution capabilities of out-of-order execution. Out-of-Order architecture feature more complex control logic, resulting in larger area and power consumption. Our results, at the same time, showed that the Out-of-Order core with zero delays associated to the rename stage have similar CPIs to the In-Order core. The latter, by having a simpler control logic, should therefore be preferred to OoO, as it allows for lower power consumptions which are essential in embedded tiny devices, while achieving

comparable performance.

Furthermore, our analysis and ablation studies on the architectural parameters of Out-of-Order core underscore the relevance of delays associated to the rename stage when evaluating performance on these specific workloads. This study highlights the potential of RISC-V architectures, particularly in-order cores, for efficiently running neural network inferences on resource-constrained edge devices. The workflow for workload and processor analysis has been thoroughly detailed, offering to students and practitioners resource material for developing new research.

The implications of our findings are significant for the field of TinyML, especially considering the growing demand for intelligent processing on edge devices. RISC-V, with its open-source nature and customizable architecture, provides a versatile platform for implementing efficient machine learning solutions. By optimizing the deployment of TFLite Micro on RISC-V, we can achieve substantial improvements in performance, energy efficiency, and cost-effectiveness, which are crucial for the widespread adoption of TinyML applications.

Our research also opens several avenues for future work. One area of interest is the exploration of additional RISC-V optimizations, such as the implementation of vector instructions, which could further enhance the performance of neural network inferences. Additionally, investigating alternative TinyML frameworks and comparing their performance on RISC-V architectures could provide valuable insights into the most effective tools for edge AI applications. Extending this study to more complex neural network architectures and real-world applications will also be crucial in validating our findings and ensuring their applicability across a wide range of use cases.

By leveraging the flexibility and open-source nature of RISC-V, we can continue to push the boundaries of TinyML performance and capability. The collaborative efforts of the academic and industrial communities will be essential in driving innovation and overcoming the challenges associated with running sophisticated machine learning models on resource-constrained devices. Ultimately, our work contributes to the ongoing advancement of TinyML, paving the way for the development of smarter, more efficient, and more accessible edge computing solutions.

REFERENCES

[1] D. Reis, J. Kupec, J. Hong, and A. Daoudi, "Real-time flying object detection with yolov8," 2024.

[2] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick, "Segment anything," 2023.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[5] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," 2022.

[6] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "Tensorflow lite micro: Embedded machine learning on tinyml systems," 2021.

[7] E. Cui, T. Li, and Q. Wei, "Risc-v instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023.

[8] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2164, p. 20190155, Dec. 2019. [Online]. Available: http://dx.doi.org/10.1098/rsta.2019.0155

[9] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. D. Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, and X. Xuesong, "Mlperf tiny benchmark," 2021.

[10] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, "Towards deep learning using tensorflow lite on risc-v," 2019.

[11] Z. Azad, M. S. Louis, L. Delshadtehrani, A. P. Ducimo, S. Gupta, P. Warden, V. J. Reddi, and A. M. Joshi, "An end-to-end risc-v solution for ml on the edge using in-pipeline support," 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:220248380

[12] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," 2017. [Online]. Available: https://arxiv.org/abs/1610.02357

[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[15] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," 2021. [Online]. Available: https://arxiv.org/abs/2003.05991

[16] "MLCommons tiny benchmark github repository," (Date last accessed 18-June-2024). [Online]. Available: https://github.com/mlcommons/tiny

[17] "Tflite micro github repository," (Date last accessed 18-June-2024). [Online]. Available: https://github.com/tensorflow/tflite-micro