Virtual Prototype based Analysis of Neural Network Cache Behavior for Tiny Edge Device

Alexander Fratzer[1] Vladimir Herdt[1,2] Christoph Lüth[1] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Alexander.Fratzer@dfki.de, vherdt@uni-bremen.de, christoph.lueth@dfki.de, drechsler@uni-bremen.de

***Abstract*—The demand for AI and specifically machine learning functionality on edge devices (TinyML) is growing. TinyML faces several unique challenges, one of them being the requirement of having a lower memory footprint for storage and inference of neural networks.**

In this paper, we propose and evaluate an approach to make *Convolutional Neural Networks* (CNNs) with higher memory footprint executable on edge devices. The idea is to combine flash memory with a cached access for the inference of CNNs. In order to evaluate the effectiveness of our proposed memory architecture by measuring the cache hitrate at the system level, we build a *Virtual Prototype* (VP) with a dedicated flash device and an exclusive cache. We are using *Tensorflow Lite Micro* (TFLM) for the network inference and mapping all model data and runtime buffers into the cache enhanced flash memory. Multiple experiments with several cache configurations show that a small cache of around 1 KB is able to achieve very high hitrates over 99%. Additionally, the experimental results show that the memory planning of TFLM supports the usage of caching because most memory accesses are adjacent.

I. Introduction

Deep learning (DL), as a class of *Machine Learning* (ML), is a well-known approach in a variety of different applications ranging from social network analysis over autonomous driving to natural language processing [1]. In the lifetime of an ML model, two phases can be distinguished from each other. First, in the so-called training phase the model learns the fundamental characteristics of the data in a supervised context. After a model is trained, it is deployed to a real world application; this is referred to as model inference. However, the training phase of a DL model can demand huge computational resources like memory, electrical power and processing speed. Consequentially, it is common practice to use powerful machines with large amounts of storage capacity and hardware accelerators for this phase, e.g. in a cloud based infrastructure [2]. From a system design point, the inference of DL models is much more restricted. Depending on the specific use case, required throughput rates and real time constraints can prohibit the use of decoupled and distributed solutions. As a consequence, the inference on edge devices, called Embedded Machine Learning or TinyML, has attracted a lot of attention recently. The unique challenges of TinyML applications according to [3] are: low power, limited memory, hardware heterogeneity and software heterogeneity.

In modern embedded systems, memory technologies are of major concern. Currently, no single technology is able meet all requirements for edge devices. *Static Random Access Memory* (SRAM) has fast access speeds and consumes low power but is expensive because of its low density. On the other side, *Dynamic Random Access Memory* (DRAM) is slower and consumes more power but is much cheaper, while Flash Memory is Non-Volatile and consumes low power but is very slow in access [4]. New technologies are emerging [5] but up to this point, they are not production ready and therefore not a viable alternative. Considering the application of TinyML, the lack of low cost, low power, fast accessible and high density memory is a major restriction. For this reason, a lot of research in this field is dedicated to develop new designs for DL models and compression techniques to lower the memory footprint for storage and inference (cf. Section II).

In this paper, we propose and evaluate an alternate approach to making DL models with higher memory footprint executable on edge devices. Instead of reducing the model complexity and optimizing its storage, the complete model and runtime memory is located in a dedicated memory with cached access. We extensively evaluate the effectiveness of our proposed memory architecture by measuring the cache hitrate at the system-level with different cache parameters and *Convolutional Neural Networks* (CNNs). In order to achieve this, we extend an open source RISC-V VP [6] with a flash component and a *Set Associative Cache* (SA-cache). A VP is essentially a simulation model of the entire hardware platform and enables easy observation of various execution metrics through simulation runs at the system level. We are using *Tensorflow Lite Micro* (TFLM) for the network inference and mapping all model data and runtime-buffers into the cache enhanced flash memory. Multiple experiments with several cache configurations show that a small cache of around 1 KB is able to achieve very high hitrates beyond 99%. Moreover, our experiments demonstrate that a small cache is already able to achieve high hitrates on a medium-sized ($\approx$ 4 MB), high end neural network with a large memory footprint. Finally, the experimental results also show that the memory planning of TFLM supports the usage of caching because most memory accesses are adjacent. Additionally, our work demonstrates the advantages of early software development using VPs.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001 and within the project ECXL.

II. Related Work

A lot of current research aims at solving the different challenges TinyML provides. One research direction focuses on the development of resource optimized architectures of neural network. In [7], the micro- and macro-architecture of a CNN is reduced to decrease its size and consequentially its throughput, but without losing much of its accuracy. A similar approach was taken in [8]. It proposed a modified version of the well-known *You Only Look Once* (YOLO) architecture in order to increase its throughput in the inference. In [9], a CNN architecture similar to FastRCNN [10] is used for audio keyword recognition on edge devices.

Another direction aims at developing compression techniques to lower the memory footprint of general purpose neuronal networks to make them more suitable for inference on edge devices. In [11], a compression pipeline to reduce the size of neural networks by a factor of 39 to 49 is presented. The results were achieved on well-known high-end network architectures like AlexNet [12] and VGG-16 [13]. First the network was reduced with pruning, after that quantization was applied to reduce the footprint of the weights and in the last stage the network was encoded with the Huffman-coding algorithm for further compression. Other research focuses on a single task of this pipeline, like [14]. It provides an overview of post training quantization techniques, which are a well-known method to reduce the memory footprint of neural networks. By applying these concepts to DL models, the fundamental datatype used for the models weights is converted to a more convenient one. Moreover, it is a key concept for the use of different hardware acceleration in training and inference. For example, GPUs are generally optimized for floating point operations, however most arithmetical units in *μ-Controllers (μ-C)* can only perform fast integer operations. Hardware support for floating point units is not always available, which makes quantization from floating point values to integer values not only more effective in memory usage but also speeds up arithmetical operations. Quantization is a technique which can be applied in-training as well as post-training. While post-training quantization is very convenient for porting existing networks to edge devices, in-training quantization is known to produce less accuracy loss for the network [15].

The authors of [16] propose a software based swapping technique to exchange neuronal network layers between a small SRAM and a much larger but slower flash memory. They implemented so called swapping-kernels for typical CNNs with an I/O scheduling algorithm and concluded that the system is able to run large CNNs without any loss in accuracy by trading reasonable runtime overhead. Their work is arguably the most similar to ours, as they also try to optimize the memory architecture of the inference running application without modifications to the basic DL model. In contrast, we tackle the problem from a hardware perspective by shifting the execution fully into slow memory and leveraging reactive caching technology to overcome I/O delays.

[17] propose using an application specific cache architecture to speed-up execution of deep learning inference targeting mobile vision applications. Their proposed cache architecture is customized whereas we investigate general purpose caching for TinyML applications.

III. Preliminaries

This section presents relevant background information on virtual prototyping using SystemC (Section III-A) and TFLM (Section III-B).

A. Virtual Prototyping using SystemC

SystemC is a C++ framework for developing and simulating hardware behavior [18]. In combination with the *Transaction-Level Modeling* (TLM) style, which enables efficient modeling of bus transactions, SystemC enables building industrial-proven VPs at the system-level [19]. In contrast to use of direct prototyping hardware like Field Programmable Gateway Arrays (FPGA) SystemC enables faster development cycles. Moreover, the architecture is very flexible and SystemC does not force components to have a specific level of detail. The drawback is that the implementation is only in software which can increases runtime significantly and can make accurate timing analyses more difficult. In this work, we use an existing open source SystemC TLM implementation of a RISC-V VP as a basis to develop a non-volatile external memory and a cache.

B. Tensorflow Lite Micro (TFLM)

TFLM [20] is an interpreter-based runtime environment for execution DL model on edge devices. It is able to run most Tensorflow models on a variety of different hardware platforms including RISC-V. The low runtime overhead and efficient resource management makes it very suitable for our work. TFLM calculates the required memory of a DL model and statically allocates runtime buffer in a previous defined address space called TFLM-arena. The model data self can be accessed by a file system or directly defined inside the application as array.

IV. Methodology

In this section we present our VP-based methodology to use dedicated caching for neuronal network inference and to analyse the effectiveness of this proposed TinyML memory architecture. We start with an overview of our major design decisions (Section IV-A), then we present our VP-based architecture (Section IV-B) and finally provide more details on the cache behavior and integration (Section IV-C).

A. Design Decisions

The fundamental idea behind our design is to model a system which is capable of running inference of larger DL models on tiny devices without changing their architectures. We identified the following three key aspects:

1) DL models must be stored in persistent memory on the edge device which requires a lot of capacity.
2) Overall memory footprint in the inference consists of static model data and dynamic runtime data.

3) Memory I/O is a main bottleneck.

Consequentially, flash memory is the most suited memory technology to persistently store the model data. While inference requires also large amounts of runtime memory, the main memory of μ-Cs often is very limited, especially for tiny edge devices. Hence, the required flash memory can also be dedicated to the task of holding runtime data during inference. To overcome the slow I/O rates of the persistent memory, we use caching as a proven technique for this purpose.

We decided to use a VP to implement and evaluate our design. The main advantages of a VP are the flexibility of the design by modeling in software rather than in hardware. Testing and debugging tasks are easier, and all runtime information of the system is easily accessible. The main drawback is that a precise timing analysis of the runtime is very difficult, but in our case the classical metric to evaluate the performance of a cache is its hitrate rather than its runtime. By placing the required measurements for the hitrate directly inside our cache implementation on the VP-level, all accesses regarding the dedicated memory can be counted accurately together with the cache hits and misses.

We decided to focus on the inference of CNNs, for two reasons: First, CNNs are very popular in cooperation with high dimensional input data (e.g. visual data) which leads to complex architectures. This is because feature extraction of high dimensional data requires large network backbones and consequentially a lot of computational resources. Having more inference capabilities for CNNs on edge device has the potential of opening a lot of new possibilities for TinyML applications. The second reason is that the convolutional layers can be seen as the main operation in a CNN, and a system's ability to execute these layers can be further investigated in order to evaluate its performance.

We have chosen TFLM as the runtime environment for our model inference because it is actively maintained and very stable. Moreover, it is capable of running inference of most DL models which are trained with Tensorflow. This brings our technology stack closer to common industrial technologies and increases transferability of our results to industrial applications.

B. Platform Overview

An overview of our platform is given in Figure 1. The left side of this graphic shows the hardware components which are implemented as the VP, while the right side represents the memory organization of the TFLM application and can be understood as the software side of our design. The application is statically linked with single time allocated memory. Moreover, the raw model data is represented as a static data field and directly compiled into the application. We defined this as the *.modelsection*. Additionally, the statically allocated memory area for the TFLM-arena is marked as *.arenasection*. We leveraged the linker to place these sections on fixed memory locations. In the initialization phase of the VP, an ELF-Loader takes the executable from the host system and places the different memory segments to their predefined locations. We divided the full address space of the VP into two main sections: main memory and flash memory. A dedicated memory mapping component is used to route accesses to the correct device. While the ELF-Loader is operating, caching is disabled. After this, the *.modelsection* and *.arenasection* are placed in the flash memory while all other sections of the applications are placed in the main-memory. Following this phase, the caching is enabled, and the VP starts executing the application.

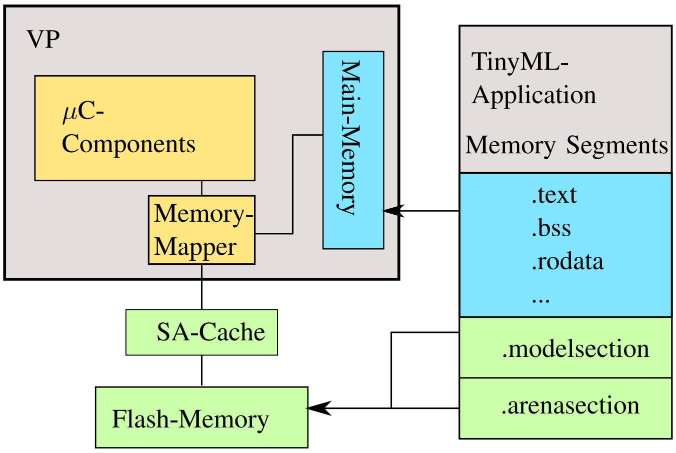

Fig. 1. VP-based edge device platform with TFLM application. The left side shows the hardware, which the VP emulates, and on the right is the software. Data and runtime memory is mapped to the flash memory using a dedicated cache component.

```
template <class ADDR, class CB>
CACHE_OP_STATE cache_operation(ADDRESS addr,
    CacheLine& cache_line, CB operation) {

  bool cache_hit = is_cache_hit(addr,
      cache_line);
  CACHE_OP_STATE cstate = CACHE_OP_STATE::
      UNKNOWN;
  count += 1; // memory accesses

  if (cache_hit) {
    hits += 1; // cache-hits
    cstate = CACHE_OP_STATE::HIT;
  } else {
    misses += 1; //cache-misses
    if (cache_line.dirty) {
      write_back(cache_line);
      cstate = CACHE_OP_STATE::WB_LOAD;
    } else {
      cstate = CACHE_OP_STATE::LOAD;
    }
    load_new_cache_line(addr, cache_line);
  }

  operation();

  cache_line.reset_age();
  return cstate;
}
```

Listing 1. Overview on the integrated caching algorithm. Relevant statistics, such as cache hits, are tracked.

C. Cache Behavior and Integration

We used a single cache to increase I/O access speed from the CPU to the flash. While the read-only section for the model

TABLE I
MEMORY FOOTPRINT OF 8-BIT INTEGER QUANTIZED CNNS IN BYTES.

CNN Class	CNN Size	TFLM Arena	Σ
small	300568	82156	382724
large	4566744	1721564	6288308

data and read-write section for the runtime buffers are handled by the flash, we implemented an SA-Cache with write-back policy as a trade-off between speed and consistency. The basic algorithm of the cache is shown in Listing 1. As defined by its policy, a hit is logged if a cacheline already contains the required data, in all other cases a miss occurs (Lines $6 - 8$). Moreover, if a cacheline is dirty and a miss occurs, a write-back operation is performed and the related cacheline is loaded (Lines $14 - 15$). Inside the cache component all memory accesses, hits and misses are tracked via counters. In order to use different cache configurations, our implementation is fully configurable by three main parameters:

1) cache size in bytes;
2) cacheline size in bytes;
3) number of association ways.

For our evaluation (see Section V) we used different combinations of these in order to find the well performing configurations.

V. EVALUATION

We have implemented our proposed methodology from Section IV, using the open source RISC-V VP [6] as a foundation. In the following, we first describe our experimental setup (Section V-A), then present our obtained results (Section V-B) and finally provide a more detailed interpretation of the results (Section V-C).

A. Setup

In order to get a general understanding of the abilities of our design we run several configurations of our cache parameters on different CNN architectures using examplary inputs from the ImageNet database [21]. Depending on the input dimensions and number of detectable objects, popular CNN architectures can range from a few kilobytes to hundreds of megabytes [13], [22]. We selected two CNNs for the task of image classification with different memory footprints in order to compare their caching performance. The first one is a person detection network already in the source of TFLM [20]. It takes about 3 KB memory for storage and represents the small to medium size network in our experiments. The second CNN is called MobileNetV1 [23] and is popular in applications running on smartphones or other powerful mobile devices. It is a medium size CNN for multi-class image classification and takes around 4.5 MB of static storage. Although not large compared to high performance CNNs employed for this kind of problem, in the context of TinyML we consider a model of such a size as large. The memory footprint of both CNNs can be seen in Table I. The first column of the table designates the CNN architecture, where *small* refers to the $\approx$ 3 KB network while *large* represents the 4.5 MB network. The second column shows the number of bytes which the particular CNN takes in static storage. The third column shows the runtime memory which TFLM needs for inference while the last column is the sum of the previous two. It can be understood as the overall memory footprint of a CNN. Both nets are compressed using 8-bit integer post-training quantization to further reduce their static memory requirements.

TABLE II
EXPLORED CACHE CONFIGURATIONS.

Cache Size	Cacheline Size
128	32, 64
512	32, 64, 128
1024	32, 64, 128, 256
16384	32, 64, 128, 256
65536	32, 64, 128, 256, 1024
262144	32, 64, 128, 256, 1024

We explored multiple configurations for the cache which are shown in Table II. While there are three parameters defining the configuration of the cache, we focused primarily on the overall cache size and the size of the cachelines. However, for the small CNN at least four different association ways were tested in each parameter configuration. In total, the small CNN runs with 88 different cache-configurations. Because of runtime restrictions, we only used the best performing configurations for the tests of the large CNN. In the end, we conduct 110 runs to fully test the abilities of the cache and suppress low performing configurations. Note that for each parameter setting, the cacheline size is at most half the general cache size, so at least two cachelines are always existing.

B. Results

The plot in Figure 2 visualizes the performance of our cache. In order to reduce the complexity of the visualization we only show the best performing configurations for each cache size. Therefore, the size of the cachelines and the number of cachelines is ignored. We consider these parameters as of minor interest for our further evaluations because we are mainly focusing on the general concept of caching. In the graphic, each bar on the x-axis represents a cache of the shown size while the y-axis distinguishes the CNNs. These results lead to several observations. The first and most obvious one includes the relation between the cache size and the hitrate. Both entities are proportional to each other. A larger cache increases the hitrate. In the case of the small CNN, the largest cache covers $\approx 68.5\%$ of the overall memory consumption. This is a very high proportion and matches a high expected hitrate of $> 99\%$. However, the hitrate of the 16 KB cache is in the same range while it covers only 4.3% of the memory consumption for the small CNN. Additionally, the smallest cache consists of 128 Bytes and leads to a hitrate over 75%. Also, the hitrates of the smallest cache are higher for the large CNN than for the small CNN. The rates up to 1 KB are slightly lower than with the small CNN but in the same area. Looking at the 16 KB cache for the large CNN, the hitrates are close

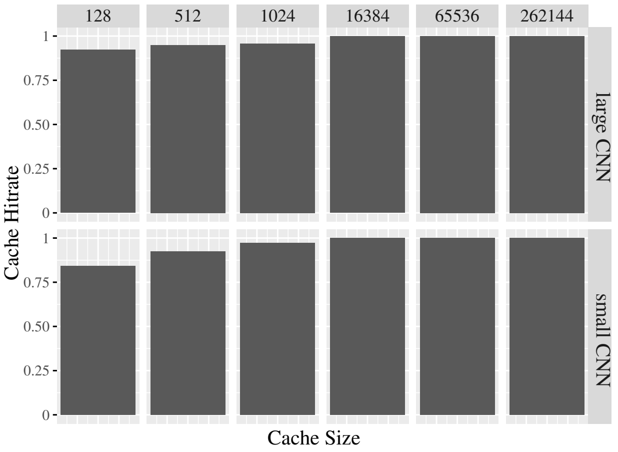

Fig. 2. Experimental results. The x-axis shows the hitrate of the cache in relation to all related memory accesses. On the y-axis the explored cache sizes are visible. The first row of bars is related to the large CNN with ≈ 4.5 MB of size while the second row shows the results of the small CNN with ≈ 300 KB.

TABLE III
OFFSETS BETWEEN ADJACENT MEMORY ACCESS INSIDE TFLM CONVOLUTIONAL OPERATION OF THE SMALL CNN. THE SOURCE COLUMN INDICATES IF THE ACCESSED VALUE BELONGED TO THE LAYER'S INPUT OR THE KERNEL.

Source	**Offset**	**Count**
input	-255	2296
	-127	25155
	-63	13644
	-31	26928
	-15	17856
	-7	34560
	1	6073211
$\sum$		**6193650**
filter	-65535	8
	-32767	8
	-16383	175
	-8191	35
	-4095	143
	-2047	143
	-1023	575
	-511	575
	-127	2303
	1	6189685
$\sum$		**6193650**

to perfect ($> 99\%$) although the cache covers only 0.02% of the required memory.

C. Interpretation

Our results suggest that caching the memory access for TLFM inference works well. A small cache in front of the slow storage- and execution memory can produce high hitrates. Even a cache of 1 KB is able to hit the majority of memory accesses for both networks. Moreover, the size of the network is not a significant factor for the success of the caching. This phenomenon can be explained by the basic architecture of the CNNs. We used the same input dimensions for both nets while the larger one is in fact deeper and leverages more architectural features, e.g. skip connections. Logically, the depth of a network does not affect the caching performance because it only makes the list of operations for execution deeper, but the operations itself are not more complex. Another important fact is that all operations inside a CNN are performed on multidimensional data fields. Those are stored in tensors consisting of ongoing memory. The metadata of an operation is previously loaded by TLFM in the main memory and therefore not related to the cache performance. Although different types of layers can have multiple input operators, most of them can be reduced to a sequence of binary operations. Hence, only three ongoing memory sequences are required for each binary operation: one for the input, one for the parameters, and one for the output. As a result, each of them is caught by a separate cacheline. This leads to the state where the majority of operands is always available by cachelines previous to their access. In order to confirm this, we logged the memory locations of all memory access regarding the input and filter of each convolutional layer. Those two data origins represent the majority of memory access because for each simple convolutional layer of kernel-size $W_K \times H_K \times C_K$ there are $W_K * H_K$ memory accesses for each output. Based on these locations we computed offset in bytes between adjacent memory access operations. This is visualized by Table III. The first column indicates if the memory access belongs to input data or filter data. The second column shows all existing offsets in bytes between two adjacent memory access operations. For each input and category of offset bytes, the number of access is shown in the last column. We can observe, that by far the most accesses are sequential ($\mathit{offset} = 1$). The largest absolute offset to access the input feature maps of the layer is 255. Therefore, a cacheline of 256 or larger can handle all I/O for each input almost perfectly.

VI. Discussion and Future Work

Our experiments show that the usage of a dedicated cache enhanced memory especially for the inference of CNNs can be highly effective. Small caches in the category of 1 KB are able to achieve very high hitrates and can balance the runtime delays produced by cheaper and slower flash memory. This is mainly due to the efficient memory planning of the TFLM. Because the majority of relevant memory access related to CNNs has an adjacent character, the overall performance is boosted. In contrast to the work of [7], [9], [11], [14], [15] we did not modify the architecture of our CNN to lower its memory footprint. However, the application of those specialized architectures could further reduce the memory requirements and therefore complement our work. Our approach is strongly related to the work of [16]. Both approaches leveraged flash memory to be able to deploy larger networks. Because of emerging low-power embedded flash technologies the general idea does not necessarily violate the definition of TinyML, which restricts the energy consumption of such devices to 1 mW. Our research differs from [16] because we used reactive hardware instead of software based swapping techniques. As a consequence, our approach does not produce high level runtime overhead, and moreover it is reactive. Given that our system caches only data access of the inference of a CNN, and that inference itself is a deterministic task, we suggest to following ways to further improve our design in future work. This should include the design of a dedicated hardware component which is able to determine the required data of future operations and can prefetch those data in an asynchronous way. Furthermore, our approach to caching works well because TFLM uses incremental memory access which supports the performance of cachelines. Future research should also focus on finding more effective ways to store the static and runtime data of neural network layer to further improve the I/O rates. We took advantage of the freedom, flexibility, and system view of VPs for our design. Because of this, we were able to confirm the effectiveness of our design without much overhead in development. To further support knowledge transfer to industrial applications we suggest more experiments with real hardware prototypes, including an analysis of the throughput of CNNs with and without cache enhancement. Another important aspect to consider by future investigations is that our design is not optimized for memory bus traffic. In some cases, data is loaded from the flash/cache into CPU registers and then into the main memory and back; this can clearly be optimized. Another direction for follow-up research could focus on the integration of our design in dedicated hardware accelerators to further increase the throughput of the network. In summary, we consider our work to be an important first step, leading to more research in memory optimization architectures for the inference of DL models on edge devices.

VII. Conclusion

We have shown that using a dedicated memory for the inference of CNNs on tiny edge devices together with caches can be highly effective. To achieve this, we have extended a VP with all necessary components and run inference of two different CNNs on it. We used caching as a proven technique to increase I/O access of slower flash memory. Moreover, we showed that the memory planning of TFLM supports caching very effectively and can therefore be tightly coupled with general purpose caches in the TinyML context. To the best of our knowledge, dedicating both static and runtime memory to a cache enhanced memory is a novel concept. Our work has shown this is promising avenue for future research. In particular, we plan to improve upon our work by also considering accurate timing information and perform an evaluation at the register-transfer level.

References

[1] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, sep 2018. [Online]. Available: https://doi.org/10.1145/3234150

[2] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "Moving deep learning to the edge," *Algorithms*, vol. 13, no. 5, 2020. [Online]. Available: https://www.mdpi.com/1999-4893/13/5/125

[3] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. A. Patterson, D. Pau, J. Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav, "Benchmarking tinyml systems: Challenges and direction," *CoRR*, vol. abs/2003.04821, 2020. [Online]. Available: https://arxiv.org/abs/2003.04821

[4] W. Banerjee, "Challenges and applications of emerging nonvolatile memory devices," *Electronics*, vol. 9, no. 6, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/6/1029

[5] H. Liu, D. Chen, H. Jin, X. Liao, B. He, K. Hu, and Y. Zhang, "A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions," *CoRR*, vol. abs/2010.04406, 2020. [Online]. Available: https://arxiv.org/abs/2010.04406

[6] "RISC-V virtual prototype," https://github.com/agra-uni-bremen/riscv-vp, 2022.

[7] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[8] J. Pedoeem and R. Huang, "YOLO-LITE: A real-time object detection algorithm optimized for non-gpu computers," *CoRR*, vol. abs/1811.05588, 2018. [Online]. Available: http://arxiv.org/abs/1811.05588

[9] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," p. 9031–9042, 2018.

[10] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015. [Online]. Available: http://arxiv.org/abs/1504.08083

[11] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *International Conference on Learning Representations (ICLR)*, 2016.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[13] K. Simonyan and Z. Andrew, "Very Deep Convolutional Networks For Large-Scale Image Recognition," pp. 1–14, 2015.

[14] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference : A whitepaper," 2018.

[15] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *CoRR*, vol. abs/1609.07061, 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[16] H. Miao and F. X. Lin, "Enabling large nns on tiny mcus with swapping," *CoRR*, vol. abs/2101.08744, 2021. [Online]. Available: https://arxiv.org/abs/2101.08744

[17] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018, p. 129–144.

[18] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[19] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[20] 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," 2020. [Online]. Available: http://arxiv.org/abs/2010.08678

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[22] C. R. Banbury, C. Zhou, I. Fedorov, R. M. Navarro, U. Thakker, D. Gope, V. J. Reddi, M. Mattina, and P. N. Whatmough, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," *CoRR*, vol. abs/2010.11267, 2020. [Online]. Available: https://arxiv.org/abs/2010.11267

[23] 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," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861