# A Memory-aware Performance Optimization of Tensor Programs for Embedded Devices

Sunwoong Joo[*], Attila Dusnoki[#], Martyn Bliss[+], Ben Duckworth[+], Nicolas Scotto Di Perto[+]
Markó Fabó[#], Gábor Lóki[#], Dániel Vince[#], Ákos Kiss[#] and Cheul-hee Hahm[*]

[*]*Samsung Electronics, Visual Display Business Division, Suwon, South Korea*
*{sunwoong.joo, chhahm}@samsung.com*
[+]*Samsung Research United Kingdom, Staines-Upon-Thames, United Kingdom*
*{martyn.bliss, b.duckworth, n.perto}@samsung.com*
[#]*University of Szeged, Department of Software Engineering, Szeged, Hungary*
*{adusnoki, mfabo, loki, vinced, akiss}@inf.u-szeged.hu*

## Abstract

*We present a performance optimization method of tensor programs for embedded devices with memory constraints. Based on the Tensor Virtual Machine, which automatically optimizes tensor programs, we introduce four techniques to reduce memory consumption. First, we improve the search templates for convolution tensor programs to reduce memory consumption. Second, we improve the planner by focusing on a specific target processor, a central processing unit. Third, to avoid unnecessary memory allocations and copies, we improve the runtime. Last, we introduce how to choose the best configuration. Experimental results show that the proposed method gives better performance with respect to both the memory footprint and the inference time when compared to conventional ones.*

**Keywords:** Embedded AI; On-Device AI; Optimization

## 1. Introduction

The optimization of tensor programs is a crucial step in realizing deep learning applications on embedded devices with limited resources. Tensor program libraries, such as Arm Compute Library (ACL) and Microsoft Linear Algebra Subprograms (MLAS), are somewhat optimized for the target architecture. However, because tensor program libraries are implemented to cope with all possible tensor shapes and graph configurations, it is not easy to achieve optimal inference time and memory consumption for a specific model on a specific embedded device.

Manual optimization of tensor programs relies on intuition and experience. It takes a lot of time and effort for each new layer or model. Tensor shapes and graph configurations vary widely across various models, thus multiple transformations can be applied to each tensor program. Moreover, the hardware diversity of embedded devices expands the optimization search space even further. This very large search space makes manual optimization challenging, thus automated optimization methods have been introduced. E.g., in the Tensor Virtual Machine (TVM) [1, 2] deep learning compiler stack, a framework called AutoTVM tries to find the fastest configuration. This framework is designed to compile the tensor program, measure the inference time on a target device, and choose the fastest setting. However, this approach is relatively insensitive to memory consumption.

This paper presents a memory-aware optimization method implemented within the infrastructure of TVM. There are many candidate processors for deep learning processing. But, in terms of memory consumption, CPUs are advantageous in an embedded device with extremely limited resources. So, we assume all of the deep learning applications run on CPUs only.

The rest of this paper is structured as follows: Section 2 explains our proposed memory-aware optimization method, and Section 3 shows experimental results. Section 4 concludes our work.

## 2. Memory-aware Optimizations

The TVM deep learning compiler infrastructure focuses on executing a tensor program as fast as possible, which can lead to not considering memory consumption. This is problematic in devices that use memory-sensitive image-to-image models, such as Style-Transfer or Super-Resolution [3]. This can be addressed by implementing memory-aware optimizations such as those introduced in the following subsections.

### 2.1 Improvements of Convolution Tensor Programs

Convolution is one of the most computationally intensive operations in neural networks. The main idea of TVM's default search for a fast convolution tensor program is to transpose the data into a more compute-friendly layout to reduce the inference time. The main drawback is that these intermediate representations usually require more space.

Our approach is to eliminate most of these allocations by moving the intermediate transformations into the computations. Some cases even eliminate them entirely, while still maintaining the same or even better performance. The potential to save memory depends highly on the convolution size. For example, in the case of a 1×3×1080×1920 (NCHW) input size, if we compute at the H axis, we only need 1080×1920 sized buffers for the intermediate representation. Instead, if the C axis is used for computation, this requires three times bigger memory allocation.

Our solution is to extend TVM's computation and search space with the proper configurations. There is no ultimate configuration which fits for all cases. Inlining everything would give us the best memory usage, but it would result in an overcomplicated direct convolution which loses all the speed improvements made with the transformations. Our optimal case is a slight-or-no runtime loss with a moderate-to-high memory reduction. Allocating inside a computation requires a cache-friendly memory layout to be effective, therefore we had to reorder and tile the computational axis to match that. For target devices which support NEON, we made sure to have the proper format to leverage all the possible vectorization. The biggest drawback is the configuration space size which could explode if we increase the number of parameters above a certain limit. This should be considered while introducing any new option as a parameter.

The transpose convolution, also known as deconvolution, is commonly used in image-to-image models. It up-scales the input data, and with all the intermediate allocations, it is a very memory heavy operation. With a memory-restricted target device, our approach is a bit limited here compared to the convolution. We had to inline all computation and find the optimal re-ordering and tiling which had the best performance. The computation uses dilation for the upscaling, which will introduce branching inside the computation when inlined. To minimize those, we used very specific tiling to avoid branches inside the inner loop of the computations. In addition, we have also done as many loop unrolling actions as it is possible to reduce the overhead of branch penalties.

### 2.2 Improvement of the Graph Planner

TVM creates the blueprint of the model via the Graph Planner. The output of this phase contains the actions that have to be done, the parameters of these actions, and the dependencies of them. These actions are the computation layers in the models. Since one layer can depend on another, it requires the output of the parent layer. These are temporary buffers in the TVM terminology.

To store the results of intermediate computations, TVM uses these buffers which are managed during each layer execution. TVM can support multiple backends and the temporary buffers have to be managed separately for every backend. Since we are focusing on the CPU backend, the memory allocations happen on the CPU side only andin this case the management of the temporary buffers can be computed ahead. Based on the dependencies and the shapes stored in the blueprint, it is possible to precompute the minimal memory space area which could hold all the temporary buffers required and to reuse obsolete areas as well. In addition, in the case of a single CPU backend, we can also combine the input and the output buffers with the temporary ones, which is not possible in the case of multiple backends.

### 2.3 Improvement of the Runtime

In the initialization phase the runtime program makes allocations, allocating those memory chunks which will be used later by layers' computations. In addition, most of these allocated spaces will be filled with static data. Since our target devices are embedded devices running Linux-based OS, the memory optimization has to consider this. In a Linux environment the most frequent method to measure memory consumption is to read the peak memory consumption number of a process from the kernel, which is stable and works quite straightforwardly. On the other hand, the Linux OS applies many optimizations in its own memory manager while allocating and deallocating physical CPU memory. One of the most painful operations in a memory-aware embedded system is the unloading of an allocated memory space. In this case the OS will only mark the area as a free memory space, but does not detach it from its current parent process. The actual unloading, detaching will be done later controlled by other conditions, such as back references, urge of allocating additional memory, etc.

The approach officially suggested by TVM is to load the weights of a model from a separate file. This leads to mapping the whole file to the process's memory region which increases the memory consumption. After that the initialization phase does a memory copy from this pre-allocated space to a special tensor data memory region. In the general case, this is necessary because the runtime doesn't know anything about the target architecture, and this special tensor data region can be at the CPU or GPU or any other memory region. In our case, it is the CPUs, and there is no need for an extra copy. Additionally, in this environment, it is not optimal to load all the static data, allocate free space for them, and keep them until the end of the process. One solution is to load only small chunks for the next allocated space instead of loading everything together. With these we can achieve an implementation that never increases the maximum resident set size above the minimum that is required during initialization.

### 2.4 Configuration Selection

In the default AutoTVM implementation, the cost function measures the inference time. In most of the cases it is a good approach, however in case of embedded devices, the memory consumption should also be considered. Our solution is a manual fine-tuning of this approach.

It is possible to rewrite the measurement method of the evaluation of the generated tensor programs. The memory consumption can be retrieved instead of inference time. With this change the AutoTVM framework searches for a

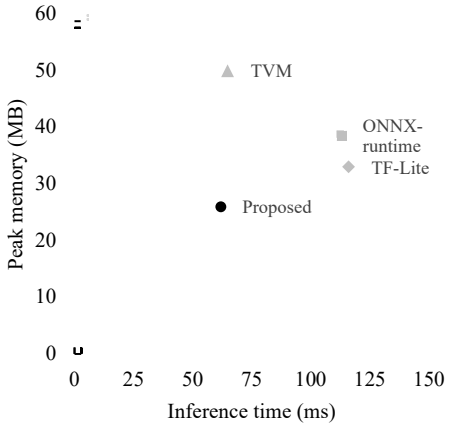


**Figure 1: Peak memory and inference time comparison for mobilenet-v1-224-1.0 model**

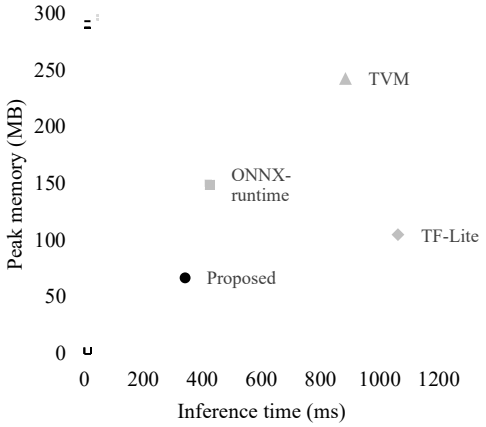


**Figure 2: Peak memory and inference time comparison for super-resolution model**

**Table 1: Inference time comparison for convolutions of mobilenet-v1-224-1.0 model (ms)**

| | TVM | Proposed |
|---|---|---|
| Convolutions | 54 | 52 |
| Depthwise-Convolutions | 10 | 6 |

**Table 2: Inference time comparison for convolutions of super-resolution model (ms)**

| | TVM | Proposed |
|---|---|---|
| Convolutions | 651 | 305 |
| Transpose-Convolution | 326 | 21 |

new configuration set which reduces the memory consumption. However, with this approach, the inference times may increase considerably. To get the best out of both approaches it is possible to limit parameters that have a large impact on memory consumption in the search templates to a certain value which leads to a better memory consumption in general.

## 3. Evaluation

In our experiment, we compared our optimization methods against the unmodified version of TVM, TensorFlow-Lite (TF-Lite), and the Open Neural Network Exchange (ONNX) runtime. TF-Lite is a framework for embedded devices, which can run converted TensorFlow (TF) [4] models. The tool is widely used, thus suitable as a baseline in our experiment. We used version 1.15, compiled with ARM NEON support. The ONNX runtime supports a variety of tensor program libraries, such as ACL and MLAS, and it is also a de-facto standard format for model exchange in various deep learning frameworks. The used version is 0.5, which supports MLAS. The baseline TVM version was 6.0, where its auto-tuning logs were used for optimization.

The evaluation platform of the experiments was an embedded device equipped with an ARM Cortex A-72 quad-core processor.

In our experiment, we have used two applications. To measure the inference time, we ran each model 51 times using randomly generated inputs. The measurement of the first warm-up run was thrown away, and the remaining 50 were averaged. To measure memory consumption, we have used peak resident set size information retrieved from the operating system.

### 3.1 MobileNet

MobileNet [5] is suitable as a reference model because it is a well-known model for embedded devices. Figure 1 shows that the tensor program optimized by TVM reduces inference time compared to TF-Lite by 43%, but memory consumption increases by 34%. In contrast, the tensor program optimized by our proposed method reduces the inference time by 46%, and also reduces memory consumption by 21% compared to TF-Lite. The inference time difference between TVM and our proposed method is not significant, but the memory consumption of the proposed method is close to half that of TVM's.

Table 1 shows the inference time for each type of convolution in the MobileNet model. Despite the significant difference in memory consumption between the baseline TVM and the proposed method, the inference time is similar. Because the default search templates provided by TVM already have a search space fitted for the MobileNet, the extended search space of the proposed method has a lesser effect on the inference. In this MobileNet experiment, most of the reduction in peak memory consumption comes from improvements in the runtime and memory planner.

### 3.2 Super-Resolution

In Super-Resolution models, unlike in classification models, the output tensor is also in an image form. Thus, the memory consumption is usually higher. For this experiment, we measured the inference time and memory

consumption of a lightweight Super-Resolution model consisting of a combination of multiple 1×1, 3×3 convolutions, and one 3×3 transpose convolution. In this model, the resolution of the input image is 250×250, and the resolution of the output image is 500×500. Figure 2 shows the inference time and memory consumption of the Super-Resolution model described above. One important observation is that the tensor program of TVM and the tensor program of ONNX-runtime show different characteristics compared to Figure 1. This is because the Super-Resolution model used in the experiment consists of convolutions with a relatively small number of channels and a large feature map size compared to MobileNet. The proposed method has extended the search space to explore tensor programs suitable for each model shape characteristic, thereby minimizing both inference time and memory consumption in models with different characteristics.

Table 2 shows the inference time for each type of convolution in the Super-Resolution model. Although both the baseline TVM and the proposed method are using optimized tensor programs for target device and model through auto-tuning, we can observe that there is a significant difference in the inference time between TVM and the proposed method, unlike the MobileNet results in Table 1. This is because the templates of the proposed method have an extended search space and so can generate a tensor program suitable for image-to-image models such as Super-Resolution.

## 4. Conclusion

In this paper, we proposed a memory-aware optimization method that significantly reduces the memory consumption and also reduces the inference time of tensor programs for embedded devices. Four techniques have been introduced, implemented, and evaluated in embedded TV devices. The first one was the improvement of convolution and transpose convolution algorithms using target specialization and addition of new parameters for the optimal search space. The second was the reuse of memory regions in the blueprint of the model to reduce the memory consumption. The third was the improvement of the initialization phase in case of targeting the CPU backend, which avoids unnecessary allocations and copies. The last one was a method to choose the best configuration while focusing on performance in a memory-aware embedded system.

Through experiments, we have shown that the proposed methods significantly reduce memory consumption in classification and in memory-sensitive image-to-image models, while also reducing inference times. The evaluation reveals that our proposed tensor program optimization methods produce the best performance and memory consumption compared to other tensor programs.

## References

[1] T. Chen, et al., “TVM: An automated end-to-end optimizing compiler for deep learning,” in Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI), Carlsbad, CA, USA, pp. 578-594, 2018.

[2] T. Chen, L. Zhang, E. Wang, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Learning to optimize tensor programs,” arXiv:1805.08166, 2018.

[3] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for real-time style transfer and super-resolution,” in Proc. 14th European Conf. Computer Vision, Amsterdam, The Netherlands, pp. 694-711, 2016.

[4] M. Abadi, et al., “TensorFlow: A system for large-scale machine learning,” in Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI), vol. 16, pp. 265-283, 2016.

[5] A. G. Howard, et al. “MobileNets: Efficient convolutional neural networks for mobile vision applications,” arXiv:1704.04861, 2017.