Extending a predictable machine learning framework with efficient gemm-based convolution routines

Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, Claire Pagetti

Extending a predictable machine learning framework with efficient GEMM-based convolution routines

Iryna De Albuquerque Silva[1], Thomas Carle[2], Adrien Gauffriau[3], and Claire Pagetti[1]

[1]ONERA, France
[2]IRIT - Université Toulouse 3 - CNRS, France
[3]Airbus, France

Abstract To implement machine learning applications in real-time safety-critical systems, we previously introduced a predictable framework named ACETONE. This framework compiles the detailed description of an off-line trained feed-forward deep neural network into an equivalent C code. In this paper, we improve the performance of the generated C code by including GEMM-based convolutions in ACETONE. The code incorporating the GEMM routines maintains the ACETONE properties of semantics preservation and timing predictability. We compare the proposed method with ACETONE's initial version, KERAS2C and UTVM on a realistic set of machine learning benchmarks and show that the introduced convolution algorithms allow a trade-off between performance and memory footprint.

Keywords: Safety-critical real-time systems, Artificial neural networks implementation, Predictable code generation

1 Introduction

Aeronautics envisions the use of machine learning (ML) algorithms to help and improve such tasks as navigation, predictive maintenance and air traffic control. However, their use in real-life operational safety-critical products raises several issues regarding functional correctness, compliance with normative requirements, formal verification, safety or implementation (Alves et al, 2018; Bhattacharyya et al, 2015).

1.1 Context

In this paper, we focus on the safe real-time implementation of *off-line trained feed-forward deep neural networks* (subsequently referred to as neural networks or DNN) on embedded platforms. The off-line design and training are done using a learning framework such as TENSORFLOW (Abadi et al, 2015) or PYTORCH (Paszke et al, 2019) and produces the *inference model* that is the final neural network with its structure (e.g. number of layers) and its parameters (e.g. weights, biases, activation functions). The implementation – the part we focus on – consists in coding the inference model in a low-level programming language, and then porting the code on the target hardware.

In order to allow the use of ML-based systems in aeronautics, guidelines – namely the AS6983 standard (EUROCAE WG-114/SAE joint group, 2021) – are currently being drafted. They complete the DO 178-C (RTCA/EUROCAE, 2011) – the reference guidance for the implementation process of software items. These two standards identify three main requirements, listed below, to be addressed for the implementation of neural networks. Our purpose is to provide an approach that enables the implementation of the inference model in compliance with avionics requirements.

1.2 General objectives and previous work

In former work (Silva et al, 2022), we have presented a first version of a framework developed according to the following objectives.

Objective 1 (Semantics preservation) *At the end of the design phase, the* inference model *should be described with a formal, unambiguous, readable specification which must be implemented on the target such that* for any input, the output of the code running on the target should be identical (given perfect real numbers representation) to the one observed in the training tool. *However, since we operate with floating-point representation that may vary per processor target or per implementation, some differences are bound to appear, and they must be acknowledged and dealt with.*

This first objective seems a priori easy since operations performed by DNNs can be expressed as linear algebra and their formal definition can be found in the literature. In reality, training frameworks have been designed to ease their use and they propose operations with default configurations that are poorly documented and framework-dependent. To circumvent this issue, we have thus formally defined the semantics of a DNN as the composition of layers associated functions (by extending and formalizing existing works from the literature). Once the semantics is formally defined, we can translate it into C code and confirm that possible discrepancies between training and deployment products are limited to floating-point arithmetic issues.

Objective 2 (Traceability) *The software developer must ensure* traceability *between the requirements and the source / binary code.*

This second objective ensures that the inference model with its requirements is what is really coded and ported on the target. One possible way to ensure traceability is to review the produced code and the specification, relying then on an exhaustive proofreading. Another recognized means of compliance (that also applies to Objective 1) consists in running *intensive testing* to verify the compliance of the implementation to the requirements. To address Objective 2, we have developed ACETONE, a framework that generates C code from any inference model. The generated code is *extremely traceable* since it is humanly possible to trace it back to the original exported DNN model. The compilation of the C code to a binary must also use the flag *-O0* (no compiler optimization), which also favours traceability. Thus, manually achieving the traceability objective is tightly related to the different design choices in the process of translating the inference model into C code.

Objective 3 (Predictability) *The software developer must compute the* WCET (Worst-Case Execution Time) *for each software component.*

Computing the WCET (Wilhelm et al, 2008) for the currently marketed machine learning deployment frameworks (e.g. TensorRT (NVIDIA, 2021), ONNX Runtime (developers, 2021)) is not straightforward. Indeed, most of the implementations are done on GPUs or TPUs with runtime engines, like TENSORFLOW, which interprets the inference model *computation graph*, i.e., a directed graph describing the mathematical structure of the inference model. GPUs and TPUs are not yet accepted in aeronautics, as these devices rely on particular execution models and shared memory, which impair their timing predictability. Moreover, their closed proprietary designs make it even more complicated to develop models and analysis techniques (Perez-Cerrolaza et al, 2022). We instead focus on general purpose multi-core, commercial off-the-shelf (COTS), hardware such as the Coolidge (Kalray, 2021) or the KEYSTONE (Texas Instruments, 2013) (used in the experiments). ML interpreters use non predictable libraries and run on top of complex, from the perspective of WCET analysis, runtime or OS. Thus, there is still a large amount of work and proof to attain the capability of computing WCETs for these software components. This is the reason why we target a more classic static approach which consists in generating a C code equivalent to the inference model and executing it with no interpretation, as proposed in Chichin et al (2020). We used in particular a static WCET analyser from the literature, OTAWA (Ballabriga et al, 2010), developed at the University of

Toulouse, to compute the WCET of the generated code. No parallelization is targeted in this paper, thus the C code is expected to run sequentially on a single core, all the memory allocations are static and the schedule (here the sequence of operations) is also static.

1.3 Contributions

The purpose of this paper is to introduce optimizations to improve the execution times of ACETONE generated programs. Although performance is not the main objective in the avionics domain it deserves to be addressed as we are working in a resource-constrained environment. Therefore, we explore several optimizations in our C code generation that preserve the semantics of the inference model and that are predictable. The first optimization, described in Section 3.4, concerns the memory layout of tensors in order to improve data locality and reduce execution time. The second optimization, presented in Section 4, is the use of GEMM-based convolutions. It concerns the re-implementation of the GEMM (General Matrix Multiplication) routine, which is part of the classic BLAS (Basic Linear Algebra Subprograms) (Dongarra et al, 1990) libraries. Those routines have existed for many years and are used in many domains. However, they do not conform with aeronautics requirements. Indeed, they integrate many compilation optimizations and often use dynamic memory allocation, which is particularly challenging respectively for Objectives 2 and 3. As we did not find any C libraries that meet our needs, we have recoded them in ACETONE to perform convolutions.

To assess the benefits of the optimizations proposed in ACETONE, we made a thorough evaluation of our framework. We have selected a set of real-world use cases and compared our results with KERAS2C (Conlin et al, 2021) and uTVM with static C runtime (Stahl, 2021). In particular, we have ported the binary to an ARM Cortex-A15 of the KEYSTONE (Texas Instruments, 2013) to evaluate the measured execution times. The optimizations introduced do not modify the semantics of the code generated by ACETONE, thus the results presented in Silva et al (2022) are still valid. As in the previous work, we used an ARM-based target that is supported by OTAWA for the WCET analysis, regardless of the fact that it is not representative of real-world application targets. Overall, in terms of performance, the new convolution strategies proposed in this paper are on average 36% faster than the original implementation and the best version shows a speed-up of respectively 1.9× and 5.5× over STATIC uTVM and KERAS2C.

The outline of the paper is as follows. Section 2 recalls briefly the notion of neural networks and part of the semantics we have defined. We particularly focus on the convolution and pooling layers as they are the ones targetted by the optimizations. Section 3 presents ACETONE and its software architecture, including the newly implemented memory layout. Section 4 describes the optimizations concerning GEMM-based convolutions. Section 5 details our testing methodology. Section 6 gives the results of the experiments. Section 7 discusses related work and Section 8 provides concluding remarks.

2 Deep Neural Networks

We focus on the inference of off-line trained feed-forward Deep Neural Networks (DNN). More precisely, we consider convolutional neural networks (CNN) and multi-perceptron (or fully-connected) neural networks. This section is a brief excerpt from Silva et al (2022). There are multiple ways to define DNNs but we chose to express them as mathematical functions. The input of these functions can be seen as a multi-dimensional vector also called *tensor*. Their output is also a tensor. We consider 1D-, 2D- and 3D-tensors but to save space, we only provide definitions for 3D subsequently. We only regard inference with one input (no batch).

Definition 1 (Tensor) *A 3D-tensor T is represented by its* size (n_h, n_w, n_c) *where n_h is the height, n_w the width and n_c the number of channels (or feature maps). We denote by T_{x_1,x_2,x_3}*

the value of T for the indices x_1, x_2, x_3. We denote by $T[s_{11} : s_{21}, ..., s_{1k} : s_{2k}]$ the slice *of T of all the values $T_{s_{11}+x_1,...,s_{1k}+x_k}$ with $i \in [1, k]$ and $x_i \in [1, s_{2i} - s_{1i}]$.*

Definition 2 (Feed-forward Deep Neural Network) *A* feed-forward neural network $N =< l_1, \ldots, l_n >$ *is a succession of* layers l_i *taking as input the output of the previous layer l_{i-1}. The first layer takes the input tensor. A layer can be of* type$(l) \in \{$act, bias, padd, conv, pool, batch norm, flat, dense$\}$ *where* act *is an activation,* padd *is a padding,* bias *is a bias adding,* conv *is a convolution,* pool *is a pooling,* batch norm *is normalization,* flat *is flattening and* dense *is a perceptron. A layer comes with a set of parameters (e.g. weights or stride).*

Definition 3 (Function associated to a DNN) *The function f_N computed by a DNN $N =< l_1, \ldots, l_n >$ is the composition of the functions computed by each layer $f_N = f_{l_n} \circ \ldots \circ f_{l_1}$.*

The semantics of each function is given in the work of The Khronos NNEF Working Group (2018) and also in Silva et al (2022) with mathematical equations. Let us just recall what a convolution and a pooling layer are.

Definition 4 (2D-convolution associated function) *Let K be a vector of 3D-tensors $[K^1, K^2, \ldots, K^{n_k}]$ representing the* kernels *of the convolution. Each kernel K^i is of size (f_h, f_w, f_c). Let $s = (s_h, s_w)$ be the* stride *parameter with s_h and s_w two integers. The 2D-convolution[1] $\mathcal{C}_{K,s}$ applied to a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{C}_{K,s}(I)$ of size (o_h, o_w, o_c) with $o_h = \left\lfloor \frac{n_h - f_h}{s_h} + 1 \right\rfloor$, $o_w = \left\lfloor \frac{n_w - f_w}{s_w} + 1 \right\rfloor$ and $o_c = n_k$. We have $O_{x,y,z} = \sum_{i=1}^{f_h} \sum_{j=1}^{f_w} \sum_{m=1}^{f_c} K^z_{i,j,m} \cdot I_{s_h \cdot (x-1)+i, s_w \cdot (y-1)+j, m}$ for all $x \leq o_h$, $y \leq o_w$ and $z \leq o_c$. Note that also we must have $f_c = n_c$ thus, convolutions are often applied on 3D-tensors on which* padding *has been applied first to fit the sizes.*

Definition 5 (Pooling layer associated function) *Let $s = (s_h, s_w)$ be the* stride *parameters, let $k = (k_h, k_w)$ be the height and width of the* window *and let $f : \mathbb{R}^{k_h \cdot k_w} \longrightarrow \mathbb{R}$ be a function (e.g.* max *or* average*). The pooling applied on a 3D-tensor I of size (n_h, n_w, n_c) outputs a 3D-tensor $O = \mathcal{P}\text{ool}_{k,s,f}(I)$ of size (o_h, o_w, o_c) with $o_h = \left\lfloor \frac{n_h - k_h}{s_h} + 1 \right\rfloor$, $o_w = \left\lfloor \frac{n_w - k_w}{s_w} + 1 \right\rfloor$ and $o_c = n_c$ with $O_{x,y,z} = f(I[s_h \cdot (x-1) + 1 : s_h \cdot (x-1) + k_h + 1][s_w \cdot (y-1) + 1 : s_w \cdot (y-1) + k_w + 1][z])$.*

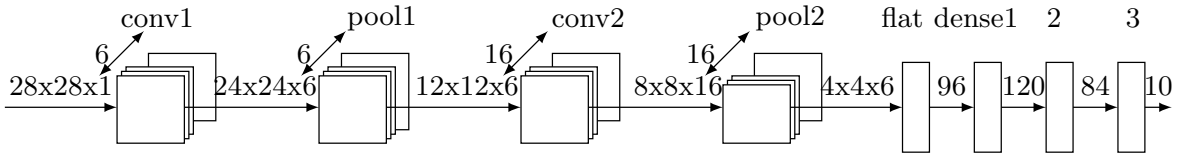


Figure 1: LeNet-5 CNN.

Example 1 (LeNet-5**)** *The* LeNet-5 *(LeCun et al, 1989) model is the basic CNN developed for handwritten digits images recognition. We used the pre-trained* LeNet-5 *from* Keras *which is shown in Figure 1. Such a graphical representation highlights the layers and the number of feature maps.*

The size of the input / output tensors are shown in Figure 1. The first 2D-convolution conv1 *takes inputs of size $28 \times 28 \times 1$, is composed of 6 kernels K^i of size $5 \times 5 \times 1$ and of a stride $s = (1, 1)$. The activation function* tanh *is applied to the outputs. The first pooling layer* pool1 *is an average pooling with stride $s = (2, 2)$ and window $k = (2, 2)$. The second 2D-convolution* conv2

[1]There may be an additional parameter, that is the *dilation* supported by the code generation and not detailed here.

is composed of 16 kernels K^i *of size* $5 \times 5 \times 6$ *and of a stride* $s = (1, 1)$*. The activation function hyperbolic tangent is applied to the outputs. The second pooling layer* pool2 *is an average pooling with stride* $s = (2, 2)$ *and window* $k = (2, 2)$*. The 3D-tensor of size* $6 \times 6 \times 4$ *is flattened in a 1D-tensor of size* 96*. There are three dense layers with respectively* $(n_i, n_o) = (96, 120)$*,* $(n_i, n_o) = (120, 84)$ *and* $(n_i, n_o) = (84, 10)$*. The two first dense layers apply the activation function* tanh *and the last one a* softmax*. Thus the function associated to this* LeNet-5 *is:* $N = \mathcal{A}_{\text{softmax}} \circ f_{\text{dense3}} \circ \mathcal{A}_{\tanh} \circ f_{\text{dense2}} \circ \mathcal{A}_{\tanh} \circ f_{\text{dense1}} \circ f_{\text{flat}} \circ f_{\text{pool2}} \circ \mathcal{A}_{\tanh} \circ f_{\text{conv2}} \circ f_{\text{pool1}} \circ \mathcal{A}_{\tanh} \circ f_{\text{conv1}}$.

3 ACETONE C back-end

We have developed a Python prototype to generate C code. We describe here the back-end of ACETONE and do not detail the front-end, which first imports the inference model description file. We reuse the semantics of Definition 3 considering every layer as an independent programming function for the code generation. The associated C inference code then consists in calling each layer function in the correct order with the expected parameters and inputs.

3.1 Software architecture

The generated C code is composed of some generic initialization functions and some model-dependent functions and files. Indeed, in the Python prototype, we have a hard-coded template library with the definition of the layers' functions, which is instantiated with the expected parameters whenever their presence is identified in the model. In such manner, we have a completely model-dependent C description of the *inference* function. Other model-dependent files refer to the weights, biases and auxiliary parameters that are also written as C files.

We have formalized the software architecture of ACETONE in uml. The main class *NeuralNetwork* contains two variables: *layers* that contains the list of *Layers* (another class defined hereafter) and *user_option* that captures the options chosen by the user for the generation, such as applying semantics-preserving transformations or selecting the algorithm to be used in the convolutional layer. That class defines three methods (in addition to the standard method *init*): *load_model* which imports the json DNN description; *forward_pass* that concatenates the layers to encode the DNN function as the composition of layers and *generate_inference_code* which generates the C code.

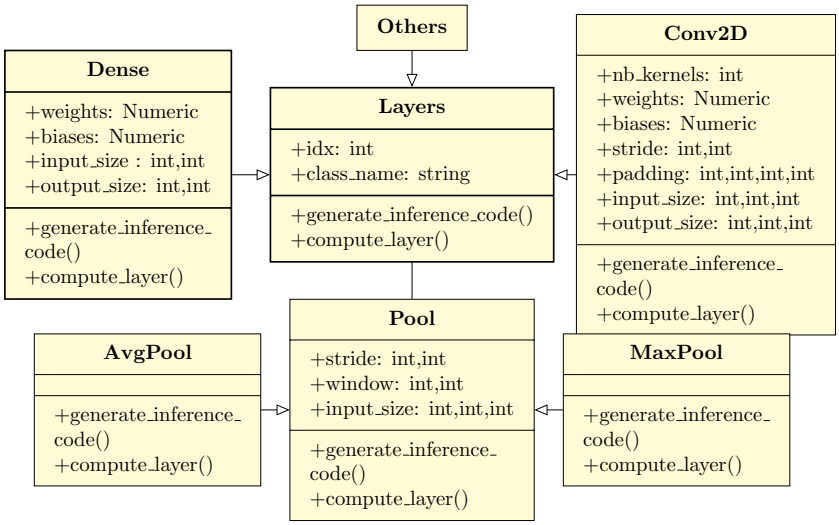


Figure 2: Class diagram of *Layers*.

Figure 2 shows the abstract *Layers* class, which is inherited by all layers types sub-classes. It

defines the parameters common to every layer, such as its *idx* in the network, and also declares two abstract methods. The first – *generate_inference_code* – implements the semantics of the layer in the C language, and the second – *compute_layer* – executes the function of the layer, mainly for debugging and evaluation purposes.

For each type of layer, we then add its particular parameters (e.g. the weights and biases for *dense*) and the methods (*generate_inference_code* and *compute_layer*) are refined. We did not detail all the layers (*others* grouping the missing ones). The prototype supports all the layers listed in Definition 2 and the *ReLu*, *Hyperbolic Tangent*, *Sigmoid* and *Linear* activation functions.

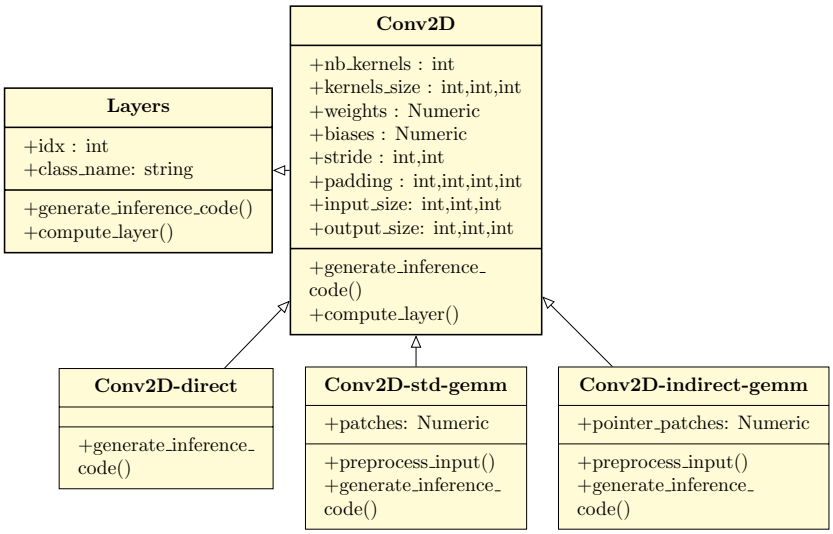


Figure 3: Class diagram of *Conv2D*.

Figure 3 presents how the different encodings of the convolution operation are managed in the software architecture. Three different classes inherit from *Conv2D* class and implement their own method. *Conv2D_direct* was the original coding of the framework while *Conv2D_std_gemm* and *Conv2D_indirect_gemm* are the extensions proposed in this paper.

3.2 Model-dependent inference function

The *inference* function is obtained by inlining the programming functions of every layer - and activation functions if any - of the model, i.e., directly writing their body to the C file. Loops-bounds and any elementary parameter are also hard-coded, deriving an entirely model-dependent *inference* function. The only parameters stored in a header C file are the constant tensors, e.g., weights or biases.

Listing 1 gives part of the *inference function* for the first *convolutional* layer of Example 1.

Listing 1: Convolutional layer inlined code of the Example 1 inference function.

```
for (int f = 0; f < 6; ++f){
  for (int i = 0; i < 24; ++i){
    for (int j = 0; j < 24; ++j){
      sum = 0;
      for (int c = 0; c < 1; ++c){
        for (int m = 0; m < 5; ++m){
          for (int n = 0; n < 5; ++n){
            int ii = i*1 + m*1 - 0;
            int jj = j*1 + n*1 - 0;

            if (ii >= 0 && ii < 28 && jj >= 0 && jj < 28){
              sum += output_pre[jj + 28*(ii + 28*c)] *
                     weights_Conv2D_01[n + 5*(m + 5*(c + 1*f))];
            }}}}
      sum += biases_Conv2D_01[f];
      output_cur[j+24*(i+24*f)] = (exp(sum)-exp(-sum))/(exp(sum)+exp(-sum));
    }}}
```

Using inlined and hard-coded functions saves function calls and the model structure parsing overheads. However, this also comes at the cost of using more instruction space, as we duplicate code, producing larger source files, which can be prohibitive in an embedded environment. Nonetheless, Otawa produces rather precise estimation for the WCET since we are able to provide the correct context in which layers are executed with no risk of pessimism in the determination of loop bounds.

3.3 Memory layout strategy

With ACETONE we proposed a solution for the deployment of the inference model in resource constrained applications, thus we were also interested in designing the inference code regarding its memory usage without however performing very fine optimizations. To that end, we studied and determined the desired memory layout of the binary corresponding to the code generated by ACETONE.

The memory space is segmented into discrete blocks with specific purposes. We mainly focus on the stack, data, BSS and text segments. We pay special attention to the stack segment because its use penalizes execution time and adds uncertainty in WCET estimation as the base address of the stack may be unknown before run time, which increases the pessimism of the WCET analysis. In our implementation the stack segment essentially contains the input and output tensors to the inference function, together with its local variables. The data segment contains the global and local statically initialized variables present in the setup functions and its size does not change at run time. Uninitialized variable data such as the intermediate tensors exchanged between layers are stored in the BSS segment, as well as the local variables used to store intermediate results within the layers functions computations. In our work, we favoured storing inference model constant parameters, in particular weight and bias tensors, as constants to statically allocate all memory at compile time, using then the text segment. Storing model parameters as read-only data also prevents mistaken access during run time.

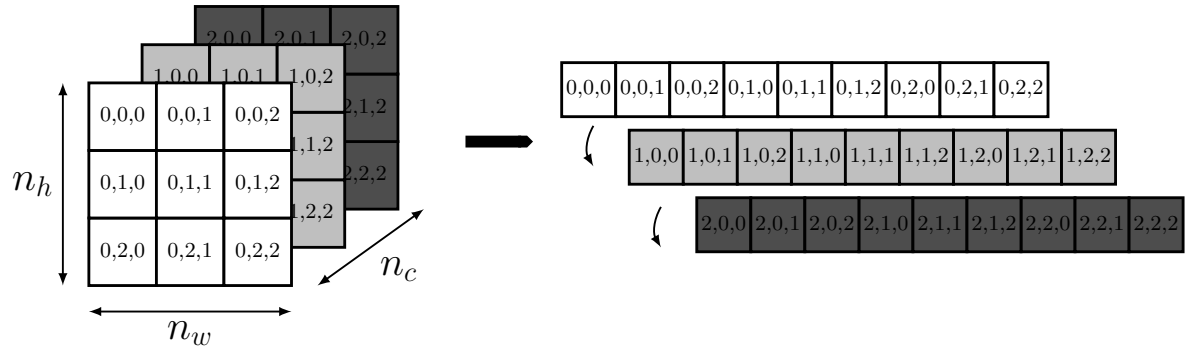
3.4 Memory layout of tensors

Since our former work, we carried a precise study on the memory layout of data and memory access patterns within the framework. We aimed to find ways to improve the memory efficiency of ACETONE. The multidimensional tensors defined in Section 2 are directly represented as one-dimensional arrays in the C code, which allows storing them in a linear memory address space.

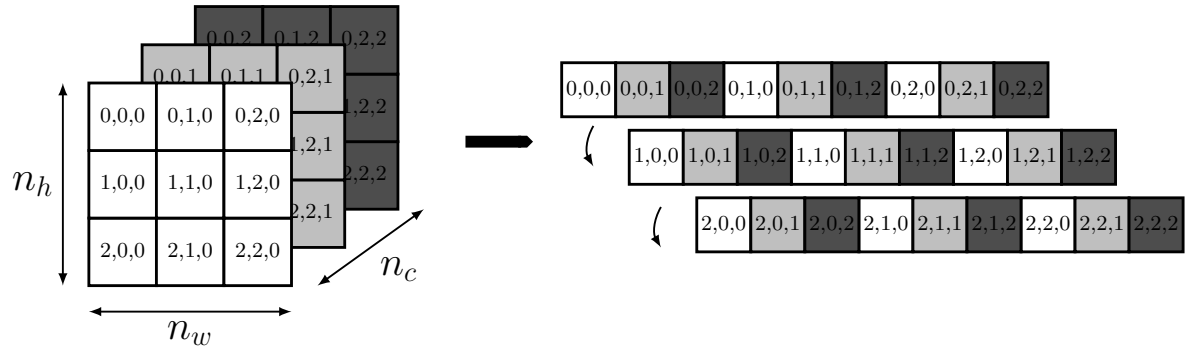
Definition 6 *The order in which the tensors values are laid out in memory is called* data layout *or* data format*.*

The data layout of 3D tensors can follow six combinations, but only two are of interest for the operations performed by neural networks: *channels-first* or *channels-last*. In the channels-first convention, the elements along the width dimension are stored consecutively in memory, the elements along the height have a stride of n_w and the elements along the channels are stored with a stride of $(n_h \cdot n_w)$. The opposite is done when the data layout follows the channels-last convention: the corresponding elements of different channels are stored consecutively in memory and, within a channel, the elements along the width have a stride of n_c and the elements along the height have a stride of $(n_w \cdot n_c)$. Figure 4a illustrates the data layout with the *channels-first* format and Figure 4b shows the data layout with the *channels-last* format, both for $n_c = n_h = n_w = 3$.

The most suitable memory layout between *channels-first* and *channels-last* depends on the layer and its parameters. For instance, the pooling operation (see Definition 5) is carried out on elements within a window of one channel at a time, before going through the remaining



(a) Channels-first data layout.



(b) Channels-last data layout.

Figure 4: Illustration of two different data layout of 3D tensors.

channels. Thus, it is intuitive to see that this operation takes advantage of the spatial locality of the channels-first data layout. Table 1 presents the number of processor cycles taken to execute a pooling layer on an ARM Cortex-A15 processor with the channels-first convention and the percentage of increase when using the channels-last format. The chosen pooling configurations are only a fraction of all those that are possible, but are representative enough of the type of tensors found in the literature. We observe that the channels-first layout outperforms its counterpart in almost every input configuration tested, except for the configuration in third column of Table 1, where $I = (27 \times 27 \times 192)$. We explain this by the fact that in this case, the tensor dimensions are rather small so the entire tensor fits in the cache. It follows that the data layout convention does not influence the execution time.

Table 1: Comparison between channels-first and channels-last layouts in pooling layer for various input tensors ($n_h \times n_w \times n_c$).

	Number of cycles			
Data layout	$(24 \times 24 \times 6)$ $k_h = k_w = 2$ $s_h = s_w = 2$	$(224 \times 224 \times 64)$ $k_h = k_w = 2$ $s_h = s_w = 2$	$(27 \times 27 \times 192)$ $k_h = k_w = 3$ $s_h = s_w = 2$	$(224 \times 224 \times 224)$ $k_h = k_w = 2$ $s_h = s_w = 1$
channels-first	141 627	165 283 135	10 679 316	1 291 051 435
channels-last	+3,49%	+1,45%	−0,09%	+7,25%

In the case of convolutional layers, as detailed in Definition 4, each kernel requires a set of elements from every channel of the input tensor. Hence, it is not straightforward to determine which memory layout will outperform the other when applying a direct convolution algorithm

like the one of Listing 1. However, our second optimization (see next section) consists in mapping the 3D input tensor to a 2D tensor, unfolding the convolutional operation into a matrix multiplication. In this case where height and width dimensions are combined, the channels-first data layout is more appropriate (Li et al, 2016).

4 GEMM-based convolution

We propose a second category of optimizations targeting the convolutional layers, while respecting the three objectives (semantics preservation, traceability and predictability) listed in the introduction. We observed in our previous work (Silva et al, 2022) that those layers were the most resource-consuming and that our implementation was ineffective. Indeed, although the *direct convolution* algorithm with its nested-loop approach presented in Listing 1 is simple, it has poor memory access pattern due to the need of replicating the input tensor for the many convolution filters. The literature shows that great effort has been made to improve the performance of convolutional layers in terms of time and memory consumption, as we discuss in Section 7. In this work we decided to explore GEMM-based convolutions for the reason that we explain next.

4.1 Context

The GEMM-based convolution algorithm first introduced by Chellapilla et al (2006) consists in translating the convolution operation as a matrix-matrix multiplication. Such a translation allows using Basic Linear Algebra Subprograms (BLAS) (Dongarra et al, 1990) libraries – low-level routines to perform common linear algebra operations – which benefit from years of research in linear algebra computation (Goto and Geijn, 2008), providing highly optimized routines.

The GEMM routine as defined in BLAS is of the form $\mathbf{C} \leftarrow \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$. For our application, we consider $\alpha = 1$ and $\beta = 0$, initially setting $\mathbf{C} = 0$, a zero matrix of the appropriate size. So $\mathbf{C}$ is obtained by multiplying element-wise the entries of the rows of $\mathbf{A}$ and the columns of $\mathbf{B}$.

In order to perform the GEMM-based algorithm, convolution inputs need to be adapted. First, the input tensor is transformed into a Toeplitz-like matrix, constructed by conveniently storing in columns the set of elements, or *patches*, that will be multiplied by each convolution kernel. This algorithm is named *im2col* in Chellapilla et al (2006), as in image-to-columns, because in most of convolution applications the input tensor is an image (3D tensor). It is illustrated in Figure 5. We observe that each column (patch) contains $f_h \cdot f_w$ elements from each channel, totalizing $f_h \cdot f_w \cdot f_c$ entries. The spatial dimensions of the output tensor (see Section 4) determine the numbers of columns in the patch matrix, i.e., $o_h \cdot o_w$. The resulting tensor of patches is then of shape $f_h \cdot f_w \cdot f_c \times o_h \cdot o_w$ and is stored in $\mathbf{B}$. Then, the kernels tensor is virtually represented as a 2D tensor (a matrix), with dimensions $n_k \times f_h \cdot f_w \cdot f_c$. This matrix of kernels is stored in $\mathbf{A}$. In Figure 5 we observe that each line of $\mathbf{A}$, illustrated with an unique pattern filling, corresponds to a different kernel of convolution. As for the input tensor, the different nuances of gray represent the multiple channels of the original tensor.

Property 1 *The* GEMM*-based convolution preserves the semantics of the direct convolution.*

Indeed, we notice that GEMM-based algorithm performs the exact same mathematical operations as in the direct convolution defined in Definition 4, only the input tensors are now organized differently in memory.

4.2 GEMM routine implementation

There are many BLAS implementations in the literature. Examples of CPU-based open-source initiatives include OpenBLAS (Xianyi et al, 2011), ATLAS (Karmani et al, 2011) and BLIS

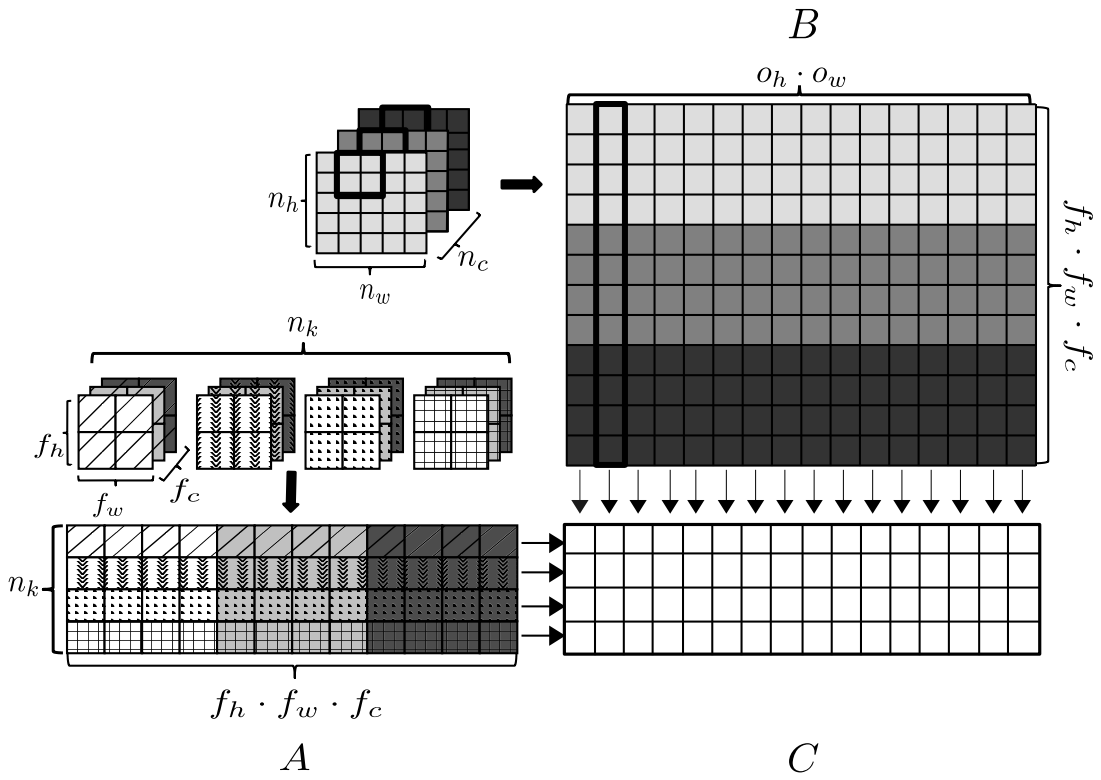


Figure 5: Example of input matrices for a GEMM-based convolution algorithm, considering a stride of one and no padding. Matrix A corresponds to the kernels tensor conventionally flattened and matrix B corresponds to the matrix of patches, created by copying and arranging elements of the original input tensor.

(Van Zee and van de Geijn, 2015). Others are proprietary, such as Intel Math Kernel Library (MKL) and Arm Performance Libraries. These libraries essentially make use of matrix blocking techniques and advanced versions also use hardware intrinsics and assembler code to improve performance. However, these projects were not developed in a manner aware of the needs of certification, hindering in particular the WCET analysis task. To overcome this limitation while still using principles presented in the BLAS, we proceeded to develop our own GEMM function for ACETONE.

In fact there are other possibilities for restructuring the data before calling the matrix multiplication routine. The original definition of the GEMM routine allows any of the three matrices to be transposed and we decided to follow the same approach to explore the different memory access patterns. In this work we developed the following four variants of the GEMM algorithm:

- GEMM_NN: matrices A and B are both accessed non-transposed ($\mathbf{A} \cdot \mathbf{B}$);
- GEMM_NT: matrix A is non-transposed and matrix B is transposed ($\mathbf{A} \cdot \mathbf{B}^\intercal$);
- GEMM_TN: matrix A is transposed and matrix B is non-transposed ($\mathbf{A}^\intercal \cdot \mathbf{B}$);
- GEMM_TT: matrices A and B are both transposed ($\mathbf{A}^\intercal \cdot \mathbf{B}^\intercal$);

Afterwards we tested these variants with a certain number of convolution configurations to analyse how they compare to each other and to the initial direct convolution algorithm. Figure 6 shows the measured execution times on an ARM Cortex-A15 expressed in number of processor cycles. Note that the results for GEMM-based algorithms take into account the patch-building

algorithm. When considering matrix **B** transposed we actually call a different patch-building algorithm, popularly known as *im2row*, whereas when matrix **A** is expected to be transposed, the transposition is done in the ACETONE back-end, with no increase in execution time. We can observe that the GEMM-based convolutions (first four bars), which exploit a better memory access pattern, always outperform the direct algorithm. However, convolution parameters, e.g, size of input tensor, number of kernels or stride, which influence the structure of the matrices, impact the performance of the different GEMM-based algorithms. Inspecting Figure 6 we conclude that no routine consistently achieves the best results across the whole network. Owing to page-space limitations, we chose to explore variants GEMM_NN and GEMM_NT in the remainder of the paper.

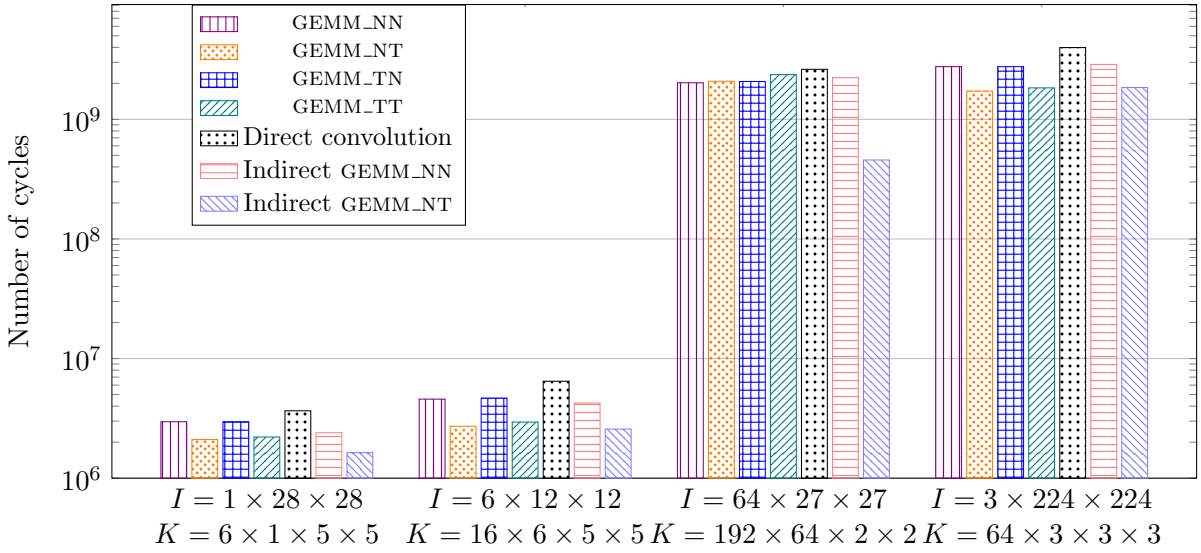


Figure 6: Comparison between GEMM-based and direct convolutions.

4.3 Optimization on the patch matrix construction

There are many axes of improvement in the presented GEMM-based convolution implementation. The first addresses the required pre-processing, i.e. the patch-building algorithm. Indeed, reshaping and duplicating elements of the input tensor before each convolutional layer is a costly job that incurs a non-trivial time penalty during inference. To address this, we developed a variant that performs the patch-building algorithm off-line. This strategy shares the same principle as the idea presented in Dukhan (2019). Essentially, the code generated by ACETONE contains a matrix of *pointers* to the elements of the input tensor that compose the patches, instead of explicitly creating the matrix of patches. Figure 7 illustrates how the matrix of pointers works. We name this algorithm *indirect* GEMM*-based convolution*, in opposition to the *standard* GEMM*-based convolution* explained before. Still in Figure 6 we examine how two algorithms using this indirect version, indirect GEMM_NN and indirect GEMM_NT, compare to their equivalent in standard version and to the direct convolution in terms of execution time. We observe a reduction in the number of cycles measured, whose intensity depends on the convolution configuration, and is explained by the fact that the patch-building algorithm is done *before* run time.

Besides that, translating the 3D input tensor into a matrix of patches requires extra memory compared to the *direct convolution*, since every element of the input is replicated up to $f_h \cdot f_w$ times. Considering this, Anderson et al (2017) proposed a design space of patch-building algorithms to translate convolution to GEMM, focusing on the memory consumption problem. We plan on studying it in future work.

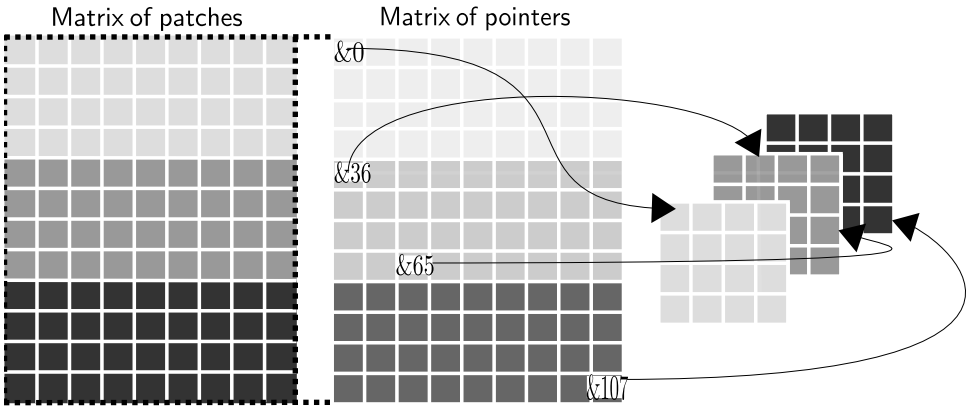


Figure 7: Indirect GEMM-based convolution. A matrix of pointers only containing the addresses of input elements is created before runtime and is used instead of the matrix of patches.

5 Comparison methodology

In order to test the practical advantages and limitations of our optimizations, as well as their behaviour compared to the other frameworks in the literature, we have defined the following methodology. We have selected a set of representative benchmarks (Section 5.1) from the state of the art of the machine learning domain and coherent with our restrictions (i.e. feed-forward DNN with restricted types of layers). The idea was to carry out a large test campaign by varying several parameters (e.g. number and type of layers, data type of parameters, type of activation function). In our test campaign we considered the memory limitations of the target used for experiments (ARM Cortex-A15), but ACETONE has no limitations on the size of models for which it can generate a semantically equivalent and predictable C code. We then introduce the two code generation frameworks selected for comparison (see Section 5.2). Finally, we define four criteria to assess the quality of implementation in accordance to the requirements listed in the introduction (see Section 5.3). In particular, not all criteria require the same level of testing: computing the WCET needs to be done once whereas the measurements need to be repeated several times.

5.1 Benchmark description

LeNet-5 The LeNet-5 model (LeCun et al, 1989) refers to the feed-forward convolutional neural network introduced in Example 1. It is one of the earliest models of this type, known for promoting the development of deep learning with the introduction of the back-propagation algorithm. While simple, this model contains the main basic layers: convolution, pooling and dense layers. All the layers have the same hyperbolic tangent activation function, except for the last one, where a softmax is performed. Overall, it has 44,426 trainable parameters to stock and an inference pass executes 572,504 floating-point operations (FLOPs).

CifarNet CifarNet was first introduced in Krizhevsky (2009) and was for a long time the state-of-the-art model used to solve the object classification problem on the Cifar-10 dataset, which consists of 32 x 32 RGB images of 10 classes. CifarNet is composed of three convolutional layers, interleaved with pooling layers, followed by two dense layers (see Figure 8). The ReLu activation function is applied to all the layers. The main difference with LeNet-5 is that it has a three-dimensional input and the convolutional layers have additional parameters such as padding and stride different from 1, which adds some complexity in terms of computation. With this configuration the number of trainable parameters increases to 122,570 alongside with 9,18

million FLOPs for inference.

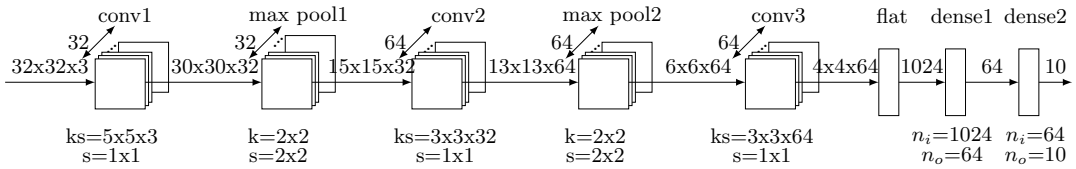


Figure 8: CifarNet CNN

AudioNet To diversify our set of test benchmarks, we built a basic audio recognition model based on an online Tensorflow tutorial (TensorFlow, 2022). Our model was trained on a speech commands dataset (Warden, 2018), which contains short audio clips with eight simple commands, such as *yes* or *stop*. In order to do so, we converted the dataset that is originally represented in the time domain to spectrograms (2D tensors). The AudioNet architecture has two convolutional layers, one pooling layer and two dense layers, with a ReLu activation function applied to every layers except for the last one (see Figure 9). The model has 1,6 million trainable parameters and performs 32,7 million FLOPs for one inference. Although real-world speech recognition systems are more complex than this architecture, it gives an insight on some other applications of DNNs.

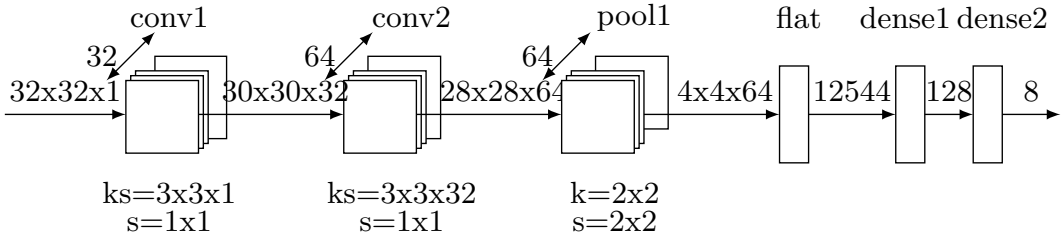


Figure 9: AudioNet CNN

Time series prediction model Time series prediction is another example where machine learning algorithms succeed. For this benchmark, we implemented the *2D-CNNpred* model presented in Hoseinzade and Haratizadeh (2019). It strives to solve a binary classification problem applied to financial time series for stock market prediction. This model uses multiple technical market indicators with different time steps, i.e. a 2D tensor, to predict the direction of a given market (or time series). The model contains three convolutional layers with a ReLu activation function, two max-pooling layers and one dense layer with the sigmoid function (see Figure 10). Another particularity of this architecture is that the spatial dimensions of convolution kernels and pooling windows are not of the same size, being instead adapted to the characteristics of financial data.

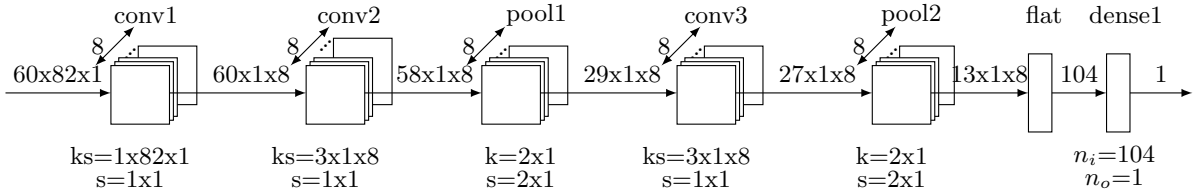


Figure 10: Time series CNN

5.2 Other C back-end frameworks

We chose two open-source frameworks from the TinyML (Ray, 2022) domain that were developed with nearly the same objectives as ACETONE.

Keras2C It is a framework to generate C code from KERAS ML models. As explained in Conlin et al (2021), the KERAS2C back-end was developed to address real-time applications and not to optimize the code for speed. In the generated C code the programming functions describing the layers are generic and all the mutable data are passed in and out of each function during the inference execution. Thus, in terms of timing analysis, KERAS2C presents a downside that is an overestimated WCET due to the inability of passing the context in which a function is called when there are multiple occurrences of it. Also, because it has a strategy of declaring all the parameters of layers as local variables initialized in the core of the function, using the stack, KERAS2C uses dynamic memory allocation when working with large neural networks. This induces additional certification challenges in terms of verification and WCET analysis. Finally, the current available version of KERAS2C implements the convolution operation using the direct algorithm, without any GEMM-based optimization.

UTVM **with static C runtime** The work of Stahl (2021) consists of a patch to TVM (Chen et al, 2018a), or more precisely UTVM, that relies on a static scheduling and static memory allocation. Its goal is to replace the original graph executor logic that is not convenient for resource-constrained devices. UTVM *with static C runtime*, or STATIC UTVM subsequently, uses the relay module produced by TVM and generates a dedicated C source code that calls the generated operator implementations directly and which is able to execute the model statically. By doing minor changes to STATIC UTVM, we were able to proceed with a timing analysis of the inference model and we could observe that the generated code when analysed with OTAWA is very similar to ACETONE code. The kernel sources generated by UTVM for bare-metal targets do not apply BLAS libraries automatically, but some other optimizations are done mainly at graph-level, such as layout optimizations, operators fusion and constant folding. When calling TVM with external (BLAS) libraries STATIC UTVM is unable to generate an entirely static code, particularly in the functions of convolution layers.

5.3 Criteria of comparison

We have identified four comparison criteria to assess our results against the requirements presented in Section 1 and the performance.

Criterion 1 (Semantics preservation evaluation) *Let $T = (T_1, T_2, ..., T_n)$ be the output tensor provided by the training framework for a given input tensor I and $\tilde{T} = (\tilde{T}_1, \tilde{T}_2, ..., \tilde{T}_n)$ be the output tensor of the C code execution for the same input. We define the absolute error as*

$$\|\tilde{T} - T\|_\infty = max_{0 \leq i \leq n} \|(\tilde{T} - T)_i\|$$

The first criterion concerns the *semantics preservation* of Objective 1, that is the ability to reproduce on the target the result of the inference pass observed in the training tool. As mentioned, we formally defined the semantics of DNNs in Section 2, we coded them in C language, through ACETONE, and now we want to verify that training and deployment outputs are close enough, i.e., verify that we correctly understood and implemented the DNN semantics used in machine learning frameworks.

When comparing outputs of PYTHON (training framework) and C (deployment framework) programs, we expect differences related to floating-point arithmetic operations, including rounding, truncation and accumulation errors added to the fact that computations are potentially done in different orders in the two implementations. The gap observed when comparing both results

should however lay in a sufficiently narrow range, which depends on the type of application the inference model is used for. In this paper we do not provide numerical values for the accepted range, as it is responsibility of the certification authorities. The norm of the criterion asserts a maximum bound on the error observed when comparing both implementations for a given input tensor and it is requested to be minimal, considering that the numerical floating-point precision in decimal digits according to the IEEE-754 standard (IEEE, 2019) is $\approx 10^{-7}$ in single-precision and $\approx 10^{-16}$ in double-precision.

Criterion 2 (Performance estimation) *For a C code corresponding to one inference run, we define the performance as the average observed time of* 50 *executions on the target, being here an* ARM *Cortex-A15 of the* KEYSTONE *(Texas Instruments, 2013).*

Our optimizations were introduced to reduce as much as possible the measured execution time. We take the average of only 50 iterations as the inference function is a deterministic algorithm being executed on a bare-metal target, i.e., variations in execution times are very limited. The minimum, maximum and standard deviation statistics presented in Table 3 support our claim.

Criterion 3 (Worst-case execution time estimation) *For a C code corresponding to one inference run, the WCET estimation consists in providing a guaranteed upper bound of the execution time of the program on a given target trough a static analysis.*

In order to determine the WCET as required by Objective 3, we compile each C code for a lpc2138 ARM-based target, and compute its WCET with OTAWA (Ballabriga et al, 2010).

We chose to use static WCET analysis as measurement-based methods may not guarantee that the actual worst-case execution time of the studied software is found. To that end, we selected the OTAWA tool because it features an ample set of techniques in terms of analysis. OTAWA supports the ARM instruction set but does not contain the micro-architecture model of the ARM Cortex-A15. Our experiments with OTAWA do not deliver a WCET bound for this target, hence we do not link WCET and performance (measured execution time) results. Such a comparison would not allow us to determine how much of the discrepancy between WCET bounds and measured execution time results comes from the code itself or from the accuracy of the analyser.

Our goal here is not to measure the accuracy of the bounds provided by OTAWA or to compare them with those obtained by a commercial analyser, but to ensure that the code generated by ACETONE is statically analysable and to compare the different variants of the generated code with the code produced by state-of-the-art frameworks, by means of a single analysis tool that provides a common basis for comparison.

Criterion 4 (Memory layout estimation) *For a C code corresponding to one inference run, the criterion requires detailing the organization of the executable's contents in the different memory segments.*

Our objective here is to reduce the size of the compiled code in general with a particular focus on stack usage.

6 Experiments

This section summarizes the results obtained following the evaluation and comparison methodology presented in Section 5.

Semantics preservation. We use Criterion 1 to compute the maximal observed error over 1000 test inputs when the generated codes were executed on a x86 target. We experimented with a consequent number of inputs to understand how the observed errors were related to the input tensors numerical values. For ACETONE and for KERAS2C, the reference training framework was KERAS and for STATIC UTVM it was TENSORFLOW LITE.

Table 2 illustrates the bounds of errors in semantics preservation both for single and double-precision floating-point arithmetic. All the frameworks produce very similar results with maximum errors of the order of 10^{-6} for all the benchmarks tested in single-precision floating-point. We were also able to test ACETONE with double-precision, for which the semantics preservation assessment indicated errors of the order of 10^{-15}. As explained in Section 5.3, we now do not have a numerical value for the maximum error tolerated to assure the semantics preservation but we observe that only one significant digit is lost in both cases given the numerical floating-point precision stated in the IEEE-754 standard (IEEE, 2019). The observed errors can be considered acceptable seeing the number of calculations performed in the inference function and we conclude that the semantics preservation is met by all methods.

Table 2: Results for the semantics preservation both in FP32 and FP64 precision.

	Maximum error	
Output tensor range	FP32 (all)	FP64 (ACETONE only)
$]-1,1[$	$\simeq 10^{-6}$	$\simeq 10^{-15}$

Performance estimation. Addressing Criterion 2, we measured the inference time on an ARM Cortex-A15 (implementing the ARMv7 architecture) of the KEYSTONE. For all experiments, caches were activated and we put data (.data and .BSS) and code (.text) sections in the DDR. Another alternative would be to place all or some of those segments in the SRAM, which is faster but has limited space (6MB against 2GB). We chose to place all the sections in DDR in order to evaluate the performance of the generated inference code itself, avoiding the complexity associated with the transfer of data between DDR and SRAM. We used the flag *mfloat-abi=hard* in order to use the *neon floating point unit* of the processor. C codes were compiled without any optimization level (-O0). Table 3 shows the results for the average (i.e., performance estimation), minimum, maximum and standard deviation values for the measured execution times (MET) for each benchmark.

We can observe a great improvement in the MET when applying a GEMM-based convolution algorithm in ACETONE. Furthermore, convolution with indirect GEMM_NT strategy is the one that performs best, as besides eliminating the patch-building overhead it has an optimal memory access pattern since input and weight tensors are accessed in the order they are laid out in memory. Also, the optimizations on operators performed by STATIC UTVM have a greater influence on smaller models, like LENET-5 and TIMESERIES but they are outperformed by GEMM-based convolution on bigger models. Finally, KERAS2C has the worst MET of all benchmarks: we attribute that to the strategy of allocating weight tensors on the stack which adds a *memcpy* overhead for each layer (copy the weights from text to stack segments).

The additional statistics on the measured execution times presented in Table 3 show that there is little variation between successive tests, with small standard deviation values, meaning that the measured execution times are close to the average of the 50 measures. Since we use the same input for consecutive measurements, the exact same operations are performed every time and small differences in execution times may be related to the initial state of the target processor.

Worst-Case Execution Time estimation. To estimate WCET, according to Criterion 3, OTAWA requires flow-fact information, that is information about the control flow: loop bounds and addresses of targets for indirect function calls (function pointers). Obtaining this information for our generated code was easy (and making this process automatic is part of future

Table 3: Average, minimum, maximum, and standard deviation values for measured execution times (50 executions) on an ARM Cortex-A15 with -O0 flag.

Framework \ Architecture		MET [cycles]			
		LeNet-5	CifarNet	AudioNet	TimeSeries
ACETONE direct convolution	avg	12 703 056	242 876 559	594 530 189	1 904 968
	min	12 702 928	242 875 774	594 033 534	1 904 564
	max	12 704 161	242 877 227	595 796 878	1 918 758
	std dev	182	199	144 771	1 386
ACETONE standard GEMM_NN	avg	10 413 463	107 685 957	373 666 343	2 359 273
	min	10 413 175	107 673 640	373 431 005	2 358 751
	max	10 413 595	107 948 273	374 130 784	2 380 949
	std dev	61	26 621	80 670	2 203
ACETONE standard GEMM_NT	avg	7 846 088	86 432 662	293 342 053	1 623 689
	min	7 845 879	86 429 874	293 028 691	1 623 437
	max	7 847 155	86 623 575	293 806 837	1 636 555
	std dev	213	19 188	62 906	1 298
ACETONE indirect GEMM_NN	avg	10 170 036	101 310 891	339 604 963	1 883 804
	min	10 169 742	101 302 717	339 429 572	1 883 416
	max	10 189 748	101 622 125	340 121 039	1 888 056
	std dev	2 027	31 354	84 270	654
ACETONE indirect GEMM_NT	avg	7 693 964	77 525 969	268 068 119	1 153 821
	min	7 693 575	77 524 834	267 927 035	1 153 807
	max	7 694 248	77 530 350	268 654 573	1 154 085
	std dev	73	663	95 841	34
static uTVM	avg	10 178 445	193 599 362	744 591 139	1 267 106
	min	10 178 413	193 563 415	741 476 588	1 266 867
	max	10 179 534	193 833 073	748 143 911	1 267 926
	std dev	135	42 230	2 671 191	177
Keras2c	avg	25 767 758	642 390 830	1 542 805 783	5 360 812
	min	25 767 366	642 345 451	1 525 852 550	5 360 386
	max	25 768 707	643 387 595	1 545 161 133	5 378 789
	std dev	216	144 170	1 925 150	1 823

work). For Keras2c and static uTVM, we had to first modify the generated code to analyse only the inference code (as we did for our code), and to leave the initialization functions out of the WCET.

Looking at Table 4, we observe that the design choices applied in the C code have a significant impact on the WCET bound. This is not simply a matter of performance optimizations, but also of the ability to provide precise flow-fact information to the analyser. We can observe that the C code generated by Keras2c is highly penalized by the Otawa analysis. Indeed, since it employs function pointers we are unable to provide contextual information about the layers function calls. Otawa thus assumes that each call to a layer function is a call to the most expensive layer of this type. Hence, the numerical results of this WCET analysis are excessively pessimistic and inaccurate.

In ACETONE, the layers are implemented as a sequence of independent loops, and in static uTVM as a sequence of separated instructions calling the layer functions. Consequently, Otawa is able to benefit from the detailed flow-fact information for these versions.

The optimizations presented in this paper aimed to improve the performance (MET) of

Table 4: WCET given by Otawa for the different frameworks.

Framework \ Architecture	WCET [cycles] LeNet-5	CifarNet	AudioNet	TimeSeries
ACETONE direct convolution	121 832 112	617 483 465	2 138 965 680	8 170 861
ACETONE standard gemm_nn	110 725 811	403 192 446	1 403 775 479	6 703 008
ACETONE standard gemm_nt	109 745 297	299 646 957	1 023 605 044	5 693 944
ACETONE indirect gemm_nn	108 054 355	393 630 348	1 377 599 430	5 404 820
ACETONE indirect gemm_nt	104 701 978	289 307 084	1 007 460 909	4 381 980
static uTVM	113 449 651	997 882 377	2 178 743 225	4 299 639
Keras2c	1 160 385 934	33 913 150 451	3 260 034 878	46 645 295 230

ACETONE generated programs in priority, essentially by making better use of the cache. We can however note in Table 4 that the WCET bounds produced by Otawa were also reduced with gemm-based convolutions compared to the direct convolution. This behaviour is consistent as the MET indeed decreased and leads us to believe that Otawa was able to recognize in its analysis the improved cache usage.

Memory layout of executable. Following Criterion 4 we analysed the memory layout of the generated codes when compiled to arm Cortex-A15. For the sake of simplicity, we only present the results obtained for the LeNet-5 model as the same trend is observed in all models.

Table 5: Memory layout of the executable generated for LeNet-5.

Framework \ Segment	Size of memory segments [bytes] stack	.data	.bss	.text	total
ACETONE direct convolution	3 424	2 792	27 960	494 732	528 908
ACETONE standard gemm	3 480	2 792	115 512	495 524	617 308
ACETONE indirect gemm	3 400	98 792	27 960	494 116	624 268
Keras2c	374 248	2 816	312	811 712	1 189 088
static uTVM	5 112	2 784	24 732	501 308	533 936

Table 5 shows how different the memory usage of the different frameworks is. We also note the increase of memory usage when using gemm-based convolution. In the case of *standard* gemm the extra space needed for the patches is allocated in the BSS segment since it is only initialized during run time by the patch-building algorithm, contrarily, for *indirect* gemm the buffer of pointer to patches is present in the initialized data segment.

Following the discussion of Section 3.3, we note that because Keras2c allocates all the weight tensors inside the inference function code, its binary stack size is higher than for the other frameworks. Moreover, weights shall also be present in the text segment, together with its static C library. static uTVM also stores the model parameters as constants, which results in a memory distribution very similar to ACETONE. The former however writes parameters as byte arrays, which is not favourable for traceability requirements and uses the same memory size as arrays of reals. Our stack usage measurement performed on the target is coherent with the stack usage estimation given by *gcc*.

Keras2c aside, we conclude that ACETONE's implementation of the direct convolution algorithm has the best memory footprint but the poorest MET and WCET. Indeed, the different

convolution strategies implemented in ACETONE allow a trade-off between optimizing for either performance or code size.

7 Related Work

There are plenty of frameworks and techniques aiming to enable the deployment of neural networks in resource-constrained embedded systems. However, most of them rely on an inference engine that dynamically explores a computation graph. In this section we focus on works that were built for or that can be adapted to the avionics domain.

7.1 Frameworks generating code or executable for DNN

Generic C code generator frameworks. The first work in this line (Chichin et al, 2020) is guided by avionics constraints as well and, in order to provide an efficient implementation of DNN inference models, the authors developed an automatic code generator that allows preserving the semantics of the trained machine learning model. However, their work scope was limited to fully-connected neural networks, with a particular focus on the definition of efficient execution models on the target and only partially discusses the feasibility of WCET analysis of feed-forward inference models.

The second is Keras2c (Conlin et al, 2021). This method consists in a library to convert Keras models into real-time compatible C code, supporting a wide range of layers and relying only on C standard library functions. Differently from this approach we aim to be agnostic of the training framework in the accepted input for the code generation tool. In Section 5 we discussed how Keras2c behaves from an avionics requirements perspective and in Section 6 we have extensively compared our results with those of Keras2c. The study of Pearce et al (2020) also investigates a predictable implementation of neural networks for safety-critical cyber-physical systems. They embed the Keras2c code on Patmos, a time-predictable processor, which is part of the larger T-CREST (Schoeberl et al, 2015) project. The software tool-chain of the latter includes a LLVM-based compiler and the Platin tool for WCET analysis. We wish, however, to rely on commercial off-the-shelf processors.

uTVM (ApacheTVM, 2021) is an extension of TVM, an optimizing compiler for machine learning models, which provides an implementation of TVM for bare metal devices. The objective of uTVM is to remove OS dependencies and abstractions, it does, however, still depend on a graph parsing within a C runtime. The adaptation of uTVM with a static C runtime (Stahl, 2021) has been extensively compared with our results in Section 6.

N2D2 (Sentieys et al, 2021) is an end-to-end framework from the creation of neural network models downs to their implementation, and including the training. On the code generation side, the authors explore how approximation techniques can improve the performance and energy efficiency of hardware accelerators in machine learning applications. This framework supports the import of ONNX models and can generate deployable solutions for different targets, including plain C code and OpenCL optimized code for DSP and GPU. We will evaluate these tools as future work to understand if the generated C code can be applied in our work scope.

LLVM front-end frameworks. TVM (Chen et al, 2018a) is a tool capable of compiling machine learning models from different popular frameworks and generating specific low-level optimized code for a diverse set of hardware back-ends. The workflow of TVM consists in first translating the inference model imported from ML frameworks in a high-level intermediate representation called Relay, performing a set of high-level and low-level optimizations and finally generating code for different compiler back-ends, including LLVM. However TVM still relies on a dynamic interpreter for the scheduling, which cannot be statically time-analysed.

MLIR (Multi-Level Intermediate Representation Overview) (Lattner et al, 2021) is a LLVM intermediate representation which was developed with the idea to use the same IR for all compiler optimizations (hence the "Multi-Level"). It contains particular features that target machine learning applications, in particular it is possible to represent computation graphs in MLIR. MLIR can be instantiated into dialects that allow putting the focus on particular aspects of the code, to specify constraints or apply specific optimizations. An example of MLIR dialect that is particularly relevant to critical embedded applications, such as the ones we target, is described in Pompougnac et al (2020): it enables the semantics of synchronous reactive applications inside an MLIR description. Nonetheless, we chose to adhere to the industrial workflow and thus generate C code for reasons of readability and traceability of the code.

7.2 Optimizations for convolutional layers

Optimizations exploiting algebraic structure of convolutions. The Fast Fourier Transform (FFT)-based convolution proposed by Mathieu et al (2014), then implemented by Chetlur et al (2014) and also used in Lin et al (2018) was proven to accelerate convolutions, however it has a large memory overhead when weight tensors are much smaller than the input tensor, because both tensors have to be mapped to the same size. Likewise, Cong and Xiao (2014) proposed using the Strassen algorithm for fast matrix multiplication in order to reduce the computational workload in convolutional layers. Indeed, the Strassen algorithm for matrix multiplication performs seven multiplications instead of eight, but many extra additions are required and extra memory to store the intermediate matrices is also needed. In the work of Lavin and Gray (2016) as well as in Park et al (2016), the authors propose speeding CNNs using Winograd's minimal filtering algorithms, which performs well for small convolutional kernels and in particular on GPU. In this work we preferred leveraging GEMM-based convolutions as we wanted first to improve convolution performance relying on a rather generic algorithm, that is not tightly dependent on convolution parameters or on the deployment hardware.

Graph-level optimizations. Graph-level optimizations aim at optimizing the computation graph of neural networks and in particular the nodes, or operators, seen as the smallest components of the computational graph. It contrasts with operation-level optimization, where the implementation of the mathematical functions themselves are targeted. Chen et al (2018a) proposes a joint operation- and graph-level optimization process to optimize end-to-end the inference model and that aims to be framework-agnostic. Their work consists of an initial high-level dataflow rewriting to generate an optimized graph and then a ML- based model finds operator-level code optimizations for a given hardware target. Common optimizations at graph level include data layout transformations and operator fusions. Liu et al (2018) extends Chen et al (2018a) with new operation-level optimizations on ARM CPUs for convolutions. Jia et al (2019) introduces a method to automatically generate equivalent graph substitutions and formally verifies their compatibility with the original graph. Furthermore, graph and data layout transformations are treated as a joint optimization problem, exposing more optimization possibilities. Graph-level optimizations alone don't impact the semantics preservation and can be performed ahead of ACETONE code generation, being complementary to our approach.

Target-dependent optimizations. Additionally, Zhang et al (2018), Amiri and Shahbahrami (2017), Pujol et al (2022) have proposed methods for improving both direct and GEMM-based convolutions using vector extensions and thus requiring processor-specific layouts. The authors of Chen et al (2018b) and also Zheng et al (2020) go further and present a method for improving the performance of convolution layers based on autotuning. It relies on a search-based approach that looks for loop transformations in a hierarchical search space constructed for a given computational graph. Recently, Tollenaere et al (2022) showed that it was possible to reduce the complexity

of these works by using only a random sampling of candidates, it prunes the search space while increasing the density of efficient implementations. We plan on studying the aforementioned hardware-dependent strategies as future work.

Optimizations through model compression. Other strategies propose reducing the complexity of DNNs through quantization, such as in Gong et al (2014), approximation or pruning within convolutional layers, like in Han et al (2016). These approaches don't preserve the semantics of the model, if only performed during inference (post-training), and are thus considered out of the scope of our work. We highlight, however, that the methodology presented in this paper can be easily adapted to support compressed inference models: once the semantics of compressed models is defined, we can generate a compliant C code and perform the same experiments described in Section 5 with adapted use cases. It is indeed part of future work objectives. In particular, for quantized models, both inference with integer-only quantization and inference with simulated quantization can be considered (Gholami et al, 2021). In the first case, minimal changes would be required in the framework to support operations in low-precision arithmetic while in the latter case additional steps would be required to dequantize model parameters, perform computations in floating-point arithmetic then requantize results. Neither severely alter the presented workflow.

8 Conclusions

Machine learning applications are widely used in many domains, however, most of them are not built with focus on avionics constraints or more generally on resource-constrained devices operating under real-time constraints. In previous work we introduced ACETONE, a framework capable of automatically generating functionally equivalent and time-predictable C code from feed-forward neural networks. In this work, we presented an extension of ACETONE to implement a new memory layout and include GEMM-based convolution algorithms, targeting performance improvement. We evaluated these new optimizations and compared them to the previous version of ACETONE and with the state of the art of C code generators for machine learning models. We proved them to be superior to Keras2c and comparable to static uTVM for the evaluated criteria. The different implementations for the convolution operator allow trading memory footprint versus performance objectives.

As future work, we plan on improving inference code from ACETONE even more. In particular, we intent to enable the use of loop tiling strategies to optimize matrix-vector and matrix-matrix multiplications. Indeed, loop tiling allows working with tailored smaller tensors which can make a better use of the cache memory and also exposes different parallelization combinations (Goto and Geijn, 2008). To that end, we will explore different existing methods for determining optimal blocking sizes, such as autotuning (Whaley et al, 2001) and analytical approaches (Low et al, 2016). Additionally, we aim to study the use of vector extensions to further exploit hardware capabilities when performing the different layer's functions, as was done in Zhang et al (2018) and also in Pujol et al (2022).

References

Abadi M, Agarwal A, Barham P, et al (2015) TensorFlow: Large-scale machine learning on heterogeneous systems. URL `https://www.tensorflow.org/`, software available from tensorflow.org

Alves E, Bhatt D, Hall B, et al (2018) Considerations in assuring safety of increasingly autonomous systems. NASA

Amiri H, Shahbahrami A (2017) High performance implementation of 2D convolution using Intel's advanced vector extensions. In: 2017 Artificial Intelligence and Signal Processing Conference (AISP), pp 25–30, https://doi.org/10.1109/AISP.2017.8324097

Anderson A, Vasudevan A, Keane C, et al (2017) Low-memory GEMM-based convolution algorithms for deep neural networks. https://doi.org/10.48550/arXiv.1709.03395, URL http://arxiv.org/abs/1709.03395, arXiv:1709.03395 [cs]

ApacheTVM (2021) microTVM: TVM on bare-metal. URL https://tvm.apache.org/docs/topic/microtvm/index.html

Ballabriga C, Cassé H, Rochange C, et al (2010) OTAWA: an open toolbox for adaptive WCET analysis (regular paper). In: IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)

Bhattacharyya S, Cofer D, Musliner D, et al (2015) Certification considerations for adaptive systems. 2015 International Conference on Unmanned Aircraft Systems, ICUAS 2015 pp 270–279. https://doi.org/10.1109/ICUAS.2015.7152300

Chellapilla K, Puri S, Simard P (2006) High Performance Convolutional Neural Networks for Document Processing. In: Lorette G (ed) Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1. Suvisoft, La Baule (France), URL https://hal.inria.fr/inria-00112631, http://www.suvisoft.com

Chen T, Moreau T, Jiang Z, et al (2018a) TVM: end-to-end optimization stack for deep learning. CoRR abs/1802.04799

Chen T, Zheng L, Yan E, et al (2018b) Learning to optimize tensor programs. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA, NIPS'18, p 3393–3404

Chetlur S, Woolley C, Vandermersch P, et al (2014) cuDNN: Efficient Primitives for Deep Learning. CoRR abs/1410.0759. URL http://arxiv.org/abs/1410.0759, https://arxiv.org/abs/1410.0759

Chichin S, Portes D, Blunder M, et al (2020) Capability to Embed Deep Neural Networks: Study on CPU Processor in Avionics Context. In: 10th European Congress Embedded Real Time Systems (ERTS 2020)

Cong J, Xiao B (2014) Minimizing computation in convolutional neural networks. In: Wermter S, Weber C, Duch W, et al (eds) Artificial Neural Networks and Machine Learning – ICANN 2014. Springer International Publishing, Cham, pp 281–290

Conlin R, Erickson K, Abbate J, et al (2021) Keras2c: A library for converting keras neural networks to real-time compatible C. Eng Appl Artif Intell 100:104,182

developers OR (2021) Onnx runtime. https://onnxruntime.ai/

Dongarra JJ, Du Croz J, Hammarling S, et al (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1):1–17. https://doi.org/10.1145/77626.79170, URL https://doi.org/10.1145/77626.79170

Dukhan M (2019) The indirect convolution algorithm. CoRR abs/1907.02129. URL http://arxiv.org/abs/1907.02129, https://arxiv.org/abs/1907.02129

EUROCAE WG-114/SAE joint group (2021) Certification/approval of aeronautical systems based on AI. On going standardization.

Gholami A, Kim S, Dong Z, et al (2021) A survey of quantization methods for efficient neural network inference. CoRR abs/2103.13630. URL https://arxiv.org/abs/2103.13630, https://arxiv.org/abs/2103.13630

Gong Y, Liu L, Yang M, et al (2014) Compressing deep convolutional networks using vector quantization. CoRR abs/1412.6115. URL http://arxiv.org/abs/1412.6115, https://arxiv.org/abs/1412.6115

Goto K, Geijn RAvd (2008) Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software 34(3):1–25. https://doi.org/10.1145/1356052.1356053, URL https://dl.acm.org/doi/10.1145/1356052.1356053

Han S, Mao H, Dally WJ (2016) Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In: Bengio Y, LeCun Y (eds) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, URL http://arxiv.org/abs/1510.00149

Hoseinzade E, Haratizadeh S (2019) CNNpred: CNN-based stock market prediction using a diverse set of variables. Expert Systems with Applications 129:273–285

IEEE (2019) IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) pp 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

Jia Z, Padon O, Thomas J, et al (2019) TASO. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. ACM, https://doi.org/10.1145/3341301.3359630, URL https://doi.org/10.1145/3341301.3359630

Kalray (2021) MPPA® Coolidge™ Processor - White Paper. https://www.kalrayinc.com/documentation/

Karmani RK, Agha G, Squillante MS, et al (2011) ATLAS (Automatically Tuned Linear Algebra Software). In: Encyclopedia of Parallel Computing. Springer US, p 95–101, https://doi.org/10.1007/978-0-387-09766-4_85, URL https://doi.org/10.1007/978-0-387-09766-4_85

Krizhevsky A (2009) Learning multiple layers of features from tiny images. Tech. Rep. 0, University of Toronto

Lattner C, Amini M, Bondhugula U, et al (2021) MLIR: scaling compiler infrastructure for domain specific computation. In: Lee JW, Soffa ML, Zaks A (eds) International Symposium on Code Generation and Optimization, (CGO), pp 2–14

Lavin A, Gray S (2016) Fast algorithms for convolutional neural networks. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp 4013–4021, https://doi.org/10.1109/CVPR.2016.435

LeCun Y, Boser BE, Denker JS, et al (1989) Backpropagation applied to handwritten zip code recognition. Neural Comput 1(4):541–551

Li C, Yang Y, Feng M, et al (2016) Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016

Lin S, Liu N, Nazemi M, et al (2018) FFT-based deep learning deployment in embedded systems. In: 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE, pp 1045–1050, https://doi.org/10.23919/DATE.2018.8342166

Liu Y, Wang Y, Yu R, et al (2018) Optimizing CNN Model Inference on CPUs. arXiv https://doi.org/10.48550/ARXIV.1809.02697, URL https://arxiv.org/abs/1809.02697

Low TM, Igual FD, Smith TM, et al (2016) Analytical modeling is enough for high-performance BLIS. ACM Transactions on Mathematical Software 43(2):1–18. https://doi.org/10.1145/2925987, URL https://doi.org/10.1145/2925987

Mathieu M, Henaff M, LeCun Y (2014) Fast training of convolutional networks through FFTS: International conference on learning representations (ICLR2014), cbls, april 2014. 2nd International Conference on Learning Representations, ICLR 2014 ; Conference date: 14-04-2014 Through 16-04-2014

NVIDIA (2021) Tensorrt documentation

Park H, Kim D, Ahn J, et al (2016) Zero and Data Reuse-Aware Fast Convolution for Deep Neural Networks on GPU. In: Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. Association for Computing Machinery, New York, NY, USA, CODES '16, https://doi.org/10.1145/2968456.2968476, URL https://doi.org/10.1145/2968456.2968476

Paszke A, Gross S, Massa F, et al (2019) PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach H, Larochelle H, Beygelzimer A, et al (eds) Advances in Neural Information Processing Systems 32. p 8024–8035

Pearce H, Yang X, Roop PS, et al (2020) Designing neural networks for real-time systems. IEEE Embedded Systems Letters p 1–1

Perez-Cerrolaza J, Abella J, Kosmidis L, et al (2022) GPU Devices for Safety-Critical Systems: A Survey. ACM Comput Surv 55(7). https://doi.org/10.1145/3549526, URL https://doi.org/10.1145/3549526

Pompougnac H, Beaugnon U, Cohen A, et al (2020) From SSA to Synchronous Concurrency and Back. Research Report RR-9380, INRIA Sophia Antipolis - Méditerranée (France), URL https://hal.inria.fr/hal-03043623

Pujol R, Jorba J, Tabani H, et al (2022) Vector extensions in cots processors to increase guaranteed performance in real-time systems. ACM Trans Embed Comput Syst

Ray PP (2022) A review on tinyml: State-of-the-art and prospects. Journal of King Saud University - Computer and Information Sciences 34(4):1595–1623

RTCA/EUROCAE (2011) DO-178C/ED-12C - Software Considerations in Airborne Systems and Equipment Certification

Schoeberl M, Abbaspour S, Akesson B, et al (2015) T-crest: Time-predictable multi-core architecture for embedded systems. Journal of Systems Architecture 61(9):449–471

Sentieys O, Filip S, Briand D, et al (2021) Adequatedl: Approximating deep learning accelerators. In: 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS 21)

Silva IDA, Carle T, Gauffriau A, et al (2022) ACETONE: predictable programming framework for ML applications in safety-critical systems. In: 34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy, pp 3:1–3:19

Stahl R (2021) µTVM StaticRT CodeGen. URL https://github.com/tum-ei-eda/utvm_staticrt_codegen

TensorFlow (2022) Simple audio recognition: recognizing keywords. URL https://www.tensorflow.org/tutorials/audio/simple_audio

Texas Instruments (2013) TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip. Tech. Rep. SPRS893E, Texas Instruments Incorporated

The Khronos NNEF Working Group (2018) Neural Network Exchange Format

Tollenaere N, Iooss G, Pouget S, et al (2022) Autotuning convolutions is easier than you think. ACM Transactions on Architecture and Code Optimization https://doi.org/10.1145/3570641, URL https://doi.org/10.1145/3570641

Van Zee FG, van de Geijn RA (2015) BLIS: A Framework for Rapidly Instantiating BLAS Functionality. ACM Transactions on Mathematical Software 41(3):14:1–14:33. https://doi.org/10.1145/2764454, URL https://doi.org/10.1145/2764454

Warden P (2018) Speech commands: A dataset for limited-vocabulary speech recognition. CoRR abs/1804.03209. URL http://arxiv.org/abs/1804.03209, https://arxiv.org/abs/1804.03209

Whaley RC, Petitet A, Dongarra JJ (2001) Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27(1-2):3–35. https://doi.org/10.1016/s0167-8191(00)00087-9, URL https://doi.org/10.1016/s0167-8191(00)00087-9

Wilhelm R, Engblom J, Ermedahl A, et al (2008) The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. ACM Trans Embed Comput Syst

Xianyi Z, Qian W, Saar W (2011) Openblas: An optimized blas library. URL https://www.openblas.net/

Zhang J, Franchetti F, Low TM (2018) High performance zero-memory overhead direct convolutions. In: Dy J, Krause A (eds) Proceedings of the 35th International Conference on Machine Learning, pp 5776–5785, URL https://proceedings.mlr.press/v80/zhang18d.html

Zheng L, Jia C, Sun M, et al (2020) Ansor : Generating high-performance tensor programs for deep learning. https://doi.org/10.48550/ARXIV.2006.06762, URL https://arxiv.org/abs/2006.06762