Journal of Machine Learning and Deep Learning (JMLDL)
Volume 2, Issue 2, Date: 31- December – 2025
ISSN: 2978-4336, DOI: 10.64820/AEPJMLDL.22.19.27.122025

Evaluating DeepMon, Tensor Comprehension, and Glow: Frameworks for Optimizing Convolutional Networks

Poma Panezai [1*], Hania Batool [2], Naseer Ahmed [3], Abdul Qadeer [4], Bushra Qayyum[5]
[1, 2, 3] Department of Computer Science, BUITEMS, Quetta, Pakistan
[4] Balochistan University of Engineering & Technology, Khuzdar
Email: [1] poma.panezai@buitms.edu.pk, [2] haniya.batool003@gmail.com, [3] cs.naseerahmed98@gmail.com, [4] aq.omrani@gmail.com, [5] bushra.qayyum@yahoo.com
*Corresponding Author
poma.panezai@buitms.edu.pk

***Abstract*— Neural networks share some problems with "normal" programs: they need to be executed and used on an always growing number of different architectures. Therefore, we need a way to execute our networks on all those devices efficiently without recreating and retraining our network. Ordinary programs have solved this problem by introducing compilers for each specific architecture to create the machine code for each device from the same codebase, and neural networks can follow the same approach. There is a certain number of different frameworks for optimizing and compiling a trained network with different architectures in the inference phase. In this paper we will compare "DeepMon", a framework specialized in mobile GPUs, "Tensor Comprehension" and "Glow", two popular frameworks for general usage. Based on our evaluations, DeepMon achieves a speedup of up to 5× over basic GPU frameworks on mobile devices, while TC and Glow outperform traditional frameworks like TensorFlow and Caffe2 on server-grade GPUs. Glow achieves inference speeds that are up to 2.7 times faster than those of TensorFlow, and Tensor Comprehension matches or exceeds the performance of CUBLAS in the majority of categories.**

Keywords*— neural networks, compilation, inference phase, optimization frameworks

I. INTRODUCTION

As neural networks gain more popularity today, the demand for good performance also naturally rises. However, Moore's Law is slowly reaching its limit and we have to look for alternatives like parallel computing, scheduling and mapping, instead of just increasing the processing speed of a processor. Fortunately, neural networks, especially Convolutional Neural Networks (CNN) [1], perform many Single Instruction Multiple Data (SIMD) [2] instructions and are therefore especially suited for parallelization and optimization. But not everyone can write code that is fully optimized or is willing to spend some extra time to learn high level languages specialized in performance optimizing like CUDA [3] or OpenMP [4]. Even if they take the time to write that code on their preferred architecture, there is a chance that they miss optimization chances and do not reach peak performance in one or another way.

Another problem that arises is that not every architecture works the same way. As a consequence, transferring the neural network to another architecture leads to the code having to be written anew causing more time to be lost in the process of choosing a different mapping and scheduling strategy for the new architecture. In particular, Machine Learning (ML) researchers would greatly benefit from having more time to work on the network itself instead of worrying about its performance and architecture.

Fortunately, many frameworks for ML already address this problem by outsourcing the data and calculation to the GPU instead of using the CPU or utilizing extra tools like other compilers. This paper aims to take a closer look at Glow [5] and Tensor Comprehension (TC) [6] compilers and explain how they are used and how they optimize the performance of CNNs [7]. On top of that, we will also take a look at DeepMon [8], an ML framework specialized in optimizing CNNs performance on mobile architectures, even though mobiles have a lot less GPU cores and more limitations than desktop machines.

The rest of the paper is structured as follows: Section 2 outlines the related work done in this field and highlights the gaps in the literature that our work aims to cover. Section 3 explores the three frameworks in depth. Section 4 details the compilation flow of each framework. Section 5 specifies the performance of the frameworks in different scenarios. The key benefits and potential issues of the frameworks are shown in Section 6. The comparison of TC, Glow, and DeepMon with regards to other algorithms is presented in Section 7. Section 8 provides a concise conclusion of this paper.

II. RELATED WORK

The increasing computational demand of Deep Neural Networks (DNNs), especially CNNs, has led to extensive research on compiler-based frameworks for optimizing model execution across heterogeneous hardware platforms.

Received: 24-7- 2025
Revised: 24-8-2025
Published: 31-12-2025

A. Compiler-Based Optimization for Deep Learning

The limitations of traditional deep learning libraries, which rely heavily on vendor-specific kernels such as cuBLAS [9], [10] or cuDNN [11], have motivated the development of compiler-based optimization frameworks. These systems aim to automate performance tuning and ensure portability across heterogeneous devices. One notable effort is MLIR, which introduces a multi-level intermediate representation designed to support modular compiler pipelines for diverse machine learning workloads. By offering customizable dialects, MLIR [12] provides flexibility in representing high-level computational graphs while enabling progressive lowering to hardware-specific instructions, thereby bridging the gap between general-purpose frameworks and hardware accelerators.

Similarly, Tiramisu [13] adopts a polyhedral model to separate algorithm specification from scheduling, enabling advanced loop transformations, parallelization, and memory optimizations. Studies show that Tiramisu achieves performance comparable to hand-optimized implementations across CPUs, GPUs, and distributed platforms, demonstrating the potential of compiler-driven scheduling to replace manual low-level programming.

The TVM [14] stack has become another cornerstone in this domain, particularly with the integration of AutoTVM [15] and Ansor [16], which utilize machine-learning-guided search to explore optimization spaces for kernel scheduling. These approaches highlight the importance of combining compiler theory with automated tuning, yielding competitive performance across NVIDIA GPUs, ARM CPUs, and specialized accelerators.

B. Lightweight Inference and Edge Deployment

Parallel to general-purpose compilers, research has focused on deploying CNNs efficiently on edge and mobile devices, where memory bandwidth and power consumption are critical bottlenecks. Frameworks such as TensorFlow Lite [17], PyTorch Mobile [18], and ONNX Runtime Mobile [19] provide lightweight inference engines that integrate quantization, pruning, and operator fusion to reduce model size and computational overhead. Comparative evaluations suggest that while these frameworks deliver reasonable trade-offs between accuracy and efficiency, they often underutilize hardware-specific capabilities such as mobile GPUs or NPUs [20].

Surveys of on-device AI frameworks emphasize that performance gains at the edge increasingly depend on compiler awareness, where model compression techniques must integrate with backend optimizations [21], [22]. For instance, hybrid approaches that combine structured pruning with compiler-guided kernel generation have been shown to significantly reduce latency on mobile devices, suggesting a shift toward co-optimization of models and compilers.

C. Hardware-Software Co-Design

Beyond software compilers, CNN optimization has also been studied in the context of hardware-software co-design. FPGA-based accelerators [23], for example, enable fine-grained customization of convolutional pipelines, with designs exploiting SIMD parallelism, loop tiling, and approximate computing techniques to maximize throughput per watt. Surveys on FPGA acceleration of CNNs report that such platforms offer competitive energy efficiency compared to GPUs, albeit with higher development complexity [24]. Similarly, NPUs and heterogeneous SoCs [25] are increasingly coupled with specialized runtimes such as OpenVINO [26] and Arm NN [27], which target efficient inference on dedicated accelerators.

These efforts highlight the growing convergence between compiler infrastructures and specialized hardware accelerators. The compiler is no longer only a translation tool but an active participant in model adaptation, memory scheduling, and energy-aware optimization.

D. Gaps in the Literature

Despite significant advances, prior studies often evaluate frameworks in isolation—focusing either on general-purpose compiler infrastructures (e.g., MLIR, TVM, Tiramisu) or on specialized inference engines for mobile and embedded platforms (e.g., TensorFlow Lite, PyTorch Mobile, ONNX Runtime Mobile). There is limited work that directly compares compilers spanning both ends of this spectrum: general-purpose optimizers designed for server-grade GPUs and frameworks specialized for resource-constrained mobile devices.

This gap motivates our comparative analysis of DeepMon, Tensor Comprehensions, and Glow. By analyzing their optimization strategies, compilation flows, and performance trade-offs, we position these frameworks within the broader landscape of compiler-driven deep learning acceleration. Our study contributes by bridging general-purpose and mobile-focused approaches, offering a consolidated perspective on how compiler-based techniques can advance efficient deployment of CNNs across diverse architectures.

III. Framework Workflow

Since each of the frameworks have a different approach for the compilation problem, all of them use a different kind of input data. In this section we will describe which kind of data and why they choose to use it for each of them.

A. Tensor Comprehension

First we will introduce TC. It is not really a framework but a domain-specific language. Its goal is to help ML researchers write code that is fast and efficient without having to know how to write CUDA or low-level code for it. There may be alternatives like Halide [28], another domain-specific language, but unlike TC, the scheduling and pipelining has to be done manually in it. All of that is automatically generated in TC while still being general enough to use and integrate into other ML frameworks like PyTorch [29].

1) *The language:* The language is used for element-wise computation of tensors and strongly based on the Einstein notation, a summation convention. Therefore, it is basically just a language describing loops in which ranges of indices are inferred from context. Let us look at a simple example of the language:

$$\text{def}\, mv(float\ (M,K)A, float(K)x) \rightarrow (C)\{C(i)\ +=\ !\ A(i,k) * x(k)\}$$

Fig. 1. Tensor Comprehension without the '!' initialization reduction operator

In figure 1, a function is defined that calculates the product of a matrix A and a vector x. The output is then written into C. All the variables that are not defined in the arguments of the function are automatically considered index variables, which are 'i' and 'k' in this case. Additionally, the repetitive stores into C will be reduced over 'k' with the operator +, since 'k' only appears on the right-hand side of the expression. This is because TC will automatically use reductions if the variable has a associative or commutative operator in order to ensure no change in the computed value.

An extra operation is also utilized here. At first glance, the initialization of C seems to be missing, but TC can reduce an initialization with zero if we just append the '!' to a reduction symbol like '+='. The initialization can also be done manually as indicated in (1), if one does not want to initialize with zero as shown in Figure 1.

There is also an operator 'where' which is used to provide the compiler with more information on the range of some indexes and becomes mandatory if the range cannot be inferred from context.

$$\begin{aligned}&\text{def}\, mvbase(float\ (M,K)A, float(K)x, float(M)\ B) \rightarrow (C)\\ &\{\\ &\quad C(i) = B(i)\ \#initialization\\ &\quad C(i)\ +=\ A(i,k) * x(k)\ \#accumulation\\ &\}\end{aligned} \tag{1}$$

B. Glow

There exists two options to provide Glow with the required data. Option 1 is a node based graph which can currently be in ONNX [19] or the Caffe2 framework, which is part of PyTorch. Option 2 is a C++ based interface that constructs a graph out of code.

These graphs are strongly typed and contain a set of high-level and target independent instructions which will be compiled to target specific machine code and are divided further to provide more information to the compiler. The subdivision of this graph is modeled visually in Figure 2. At first the input graph is divided into sub graphs called modules (marked white). These modules contain a set of functions (sub sub graphs, marked red) and storage nodes (marked black), which act like global storage for the functions. Every other node is part of a function sub-graph and represents a high level operation (marked blue) of the neural network like 'MaxPool' or 'MatrixMultiply'. A node of a function can access each storage node of its respective module. Some nodes can be a 'placeholder' node which can be replaced by anything later. Usually they are used to model an input or output node or will be replaced by the trained weight. Since the specific content is unknown during compile time, they cannot be optimized.

Besides the input graph, Glow also needs a back end for the target machine. Unlike other frameworks there is no need for a large set of kernels which implement many high level functions. Instead, a collection of small functions like 'Add', 'Subtract' or 'Max' are needed. Later these simple operations will be used to construct the complex ones, the details of which are provided in Section 3.2.

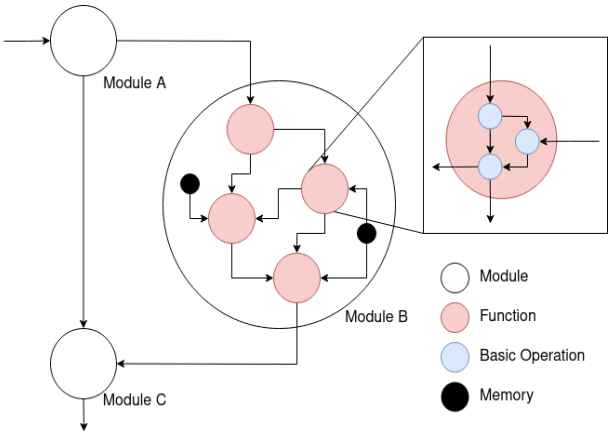

Fig. 2. Graph model of Glows high level graph

At the end Glow produces a compiled set of sub-networks which will be executed in parallel on manycore devices like GPUs.

1) DeepMon: To generate the machine code for a mobile GPU, DeepMon uses a normal model in Caffe [30], MatConvNet [31] or YOLO *[32]* format. These input models are then analyzed and a proper memory layout will be created. Since mobile GPUs have less memory bandwidth than server grade GPUs, the amount of read and write operations has to be reduced a lot to minimize latencies between the computation of different frames.

The kernel code for the GPU itself may be written in OpenCL [33] or Vulkan [34]. Based on which has been chosen, DeepMon needs to handle the kernel differently. OpenCL kernels need to be compiled first, whereas Vulkan kernels can be loaded directly.

IV. Compilation Flow

To produce memory and performance wise efficient code, there are different concepts and ideas in use. We will describe some of those ideas in detail with TC, Glow and DeepMon as example.

A. Tensor Comprehension

1) Compiler: The compiler for TC uses a polyhedral just in time compiler. That means the compiler first has to somehow lower its high-level logical layout to C99 arrays which the polyhedral code generator expects. To do that, the compiler has several lowering steps and options. It analyses the ranges and access relations of the inferred tensors, emits the declaration of the tensors and the reorder expressions and utilizes mirroring, clipping, forward substitution of convolution expressions and padding with zeros. All these lowering steps ensure no out-of-bounds access and that the data matches to the C99 arrays in row-major arrays without having to rely on specific data layouts.

But the compiler does not directly lower TC to an Polyhedral Immediate Representation (IR). The compiler

lowers TC to Halide-IR first. After that the Halide-IR is then finally lowered into the polyhedral-IR which then optimizes, generates the CUDA code and then executes it.

The reason for using Halide is that it already has all the features one may need for writing high-performance code. However, that does not mean TC is unnecessary, because the pipelines Halide provides can be potentially used for every domain while TC is specialized for tensors. Additionally, in Halide, all the mapping and scheduling has to be done manually as mentioned in Section 3.1. Therefore, TC uses tools from Halide whenever it can.

2) Schedule Trees: TC uses schedule trees [35] for scheduling and mapping. The benefit of the schedule trees is that it can relay properties or target-specific information from its high-level to the downstream optimizer.

Schedule trees introduce several nodes that define the scheduling and mapping of a TC. The nodes include; a band node that describes a partial execution order of one or multiple functions in an iteration domain, a filter node that constrains its subtree to a subset of the iteration domain, and a context node which provides extra information for variables or parameters that are constant in a subtree, such as GPU grid and block sizes. There are more node types but these are the most important ones.

The schedule trees are built by defining an outer sequence node. After the sequence node, a filter node is added for each statement in the TC. Then for each loop iterator in the statement a band node is added. After that, the schedule tree is optimized and transformed in four steps for locality and parallelism by fusing, tiling and sinking loops.

3) Scheduling: The first step to optimize the schedule tree is handled by the isl scheduler [36]. It builds a data dependence graph in which nodes represent the statements and edges represent the dependencies. Then the isl scheduler iteratively builds clusters between strongly connected nodes in the data dependence graph for fusing the loops in the schedule tree. TC also extends the isl scheduler to influence the clustering by giving the autotuner different scheduling strategies (see Section 4.1.5).

The second step for optimizing the schedule tree is by tiling imperfectly nested loops and transforming the schedule tree. Here each band is tiled separately and after the tiling is done, the parallel loops are then sunk in the tree. This completes the imperfectly nested loop tiling.

4) Mapping: As mentioned in Section 4.1.2 the schedule tree is also used for mapping. That means the schedule tree can also represent how something is mapped to an accelerator, in this case a GPU with multiple blocks and threads. This is the next step for the optimization. While TCs has its own algorithm for the mapping, it is derived from Polyhedral Parallel Code Generator (PPCG) [37]. In PPCG, the grid and block sizes are independently specified from the tile size.

While the outermost band is mapped to the GPU block, TC requires the schedule tree to have at least one outermost band with outer parallel dimensions. The innermost band is mapped to GPU threads, but the number of mapped dimensions there has to be the same on all branches. Single bands are mapped to blocks to create a single kernel in the end, since ML frameworks only accept it that way. Multiple bands on the other hand can be mapped to threads.

To reflect the mapping, special names are then inserted on the schedule tree. This is done by using a context node to deliver the additional information for the GPU grid or block size. These context nodes are then associated to band dimensions in filter nodes.

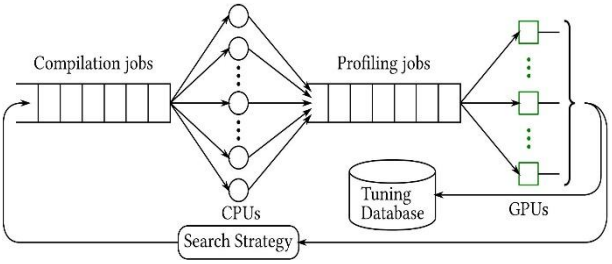

Fig. 3. The autotuning procedure of TC [6]

5) Memory Promotion: The last step performed is memory promotion. It is based on array tiles and used to promote parts of tensors into the shared or private memory of a GPU. For that purpose, TC also uses the schedule tree to attach memory-related information. To implement this step, TC extends PPCG's support for memory promotion.

There are requirements for directly accessed arrays to be promoted to shared memory. The first requirement is that the tile exists and is of a fixed size, and that some elements are accessed multiple times but at least one of these accesses do not use memory coalescing. The last requirement does not apply to indirectly accessed arrays, because they also have long memory dependencies. In the end, since the shared memory is fixed, the promotion is decided by applying a simple greedy heuristic. It is refused when the needed shared memory is larger than the available memory.

6) Optimization by autotuning: TC also uses its own autotune framework, which utilizes a compilation cache. In this compilation cache, the generated CUDA code is saved with the key being the TC it was derived from, the input shapes and the target architecture it was built for. Additionally, the performance data is also saved for all versions in the cache.

The autotune works by selecting the same starting setups of similar TCs from the compilation cache and using that as a base to search for the best configuration setup. If none are found meaning a cache miss, TC will begin with its normal compilation flow mentioned a few sections before. The algorithm applied for the setup generation is the genetic algorithm. Because the autotune has to evaluate 100s to 1000s versions for each kernel, this process is also parallelized. The many selected configurations are queued in a compilation job queue and then compiled by multiple CPUs. After that, each result is queued in a profiling job queue and gets evaluated on an available GPU. These results update the database used to generate the next configuration with the genetic algorithm. This whole process can be seen in Figure 3.

It is also possible to modify this autotuning process to a certain degree by changing some of the parameters like

number of threads, the mutation rate of the genetic algorithm or the number of generations.

B. Glow

1) High level IR: As mentioned in Section 3.2, Glow uses a strongly typed node based graph as input and converts it into the final machine code. To perform this task the input graph is transformed into several levels of IRs at which different level of optimizations are applied. The input graph is in the form of a high level IR and contains only device independent instructions. During the first stages Glow uses the graph structure to detect possibilities for optimizations like copy propagation, removing redundant or unused constants, redundant operations and more.

Glow also performs node lowering, transforming complex functions into a set of more simple linear algebra operations. This creates the potential for further optimizations, but it also is the reason for the flexibility of Glow. Since the back end does not need to implement a large set of complex and individual functions, it is relatively easy to support Glow for specific hardware. In addition to that, the back end still has the opportunity to implement specialized functions for a sequence of these low-level operations. The scheduler may also benefit from this by creating a more efficient and parallelizable schedule. Node lowering can only be applied after the differentiation phase of the network because the semantics of the graph will be changed after the nodes have been lowered.

2) Low level IR: After all high-level optimizations are finished the graph is transformed into a strongly typed low-level IR, which is described in Low Level Virtual Machine (LLVM) code [38].

All operations are now replaced by a sequence of instructions. Some low level LLVM code can be found in Figure 4. With the additional information about the actual instructions, Glow can now ap- ply memory specific optimizations to improve, for example, the amount of memory accesses or general access time. To improve these optimizations the memory may be annotated. The compiler interprets buffers with an '@in' annotation as an input buffer, therefore data is only read from it. The '@out' annotation marks an output buffer; data may only write to it. A buffer annotated with '@inout' is a combination of both. These are only some of the available annotations.

Glow also has to deal with one disadvantage of node lowering. If a complex function iterates over a buffer, it only needs to access each element once. But since we transformed this one operator into many, each of them accesses each element, resulting in an increase of memory accesses. To prevent this, Glow performs operator stacking. It detects these kinds of operations and stacks the instructions on top of each other. With that, the back end still only needs to implement the simple operations and this stack of operators only needs to iterate once over the buffer. The alternative would be to fuse the operations back together, but that would result in a new function for each combination of operations which needs to be implemented by the back end. That would be the opposite of the concept behind Glow and is impractical in general.

Quantization is another step supported by Glow, during which the floating point-based arithmetic is replaced by an integer based one. To check whether it is even possible and

```
declare {
 %input = weight float<8 x 28 x 28 x 1>,
     broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>,
     xavier, 25.0
 %filter0 = weight float<16>, broadcast,
     0.100
 %weights = weight float<10 x 144>, xavier,
     144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo,
     @in %input, @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in
     %allo0, @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

which size of the integers is even needed, the compiler adds some profiling nodes to potential variables, tensors and more to monitor their minimal and maximal value during inference based on findings [39]. After that, the floating point-

Fig. 4. LLVM code of Glows low level IR [5]

operations are replaced by integer-based ones if possible. Glow may even generate an integer-based sub graph in a floating point-based function, depending on profiling results. But to minimize performance loss, the amount of conversion is kept as low as possible.

The back end may implement additional levels of IR or additional compile passes and after that the actual machine code is generated with the help of a small target independent standard library.

C. DeepMon

A DeepMon model is processed in two main phases. In the Model Conversion phase, a model is converted to fit the restrains of a mobile GPU and a proper memory layout is created. Furthermore, the memory for input and output data will be allocated. During the inference [40] phase the actual stream of images is processed. A frame dispatcher will select important frames and uses them as input for the neural network.

1) Convolutional layer caching: An important feature of DeepMon is its ability to cache the result of a processed image. Images of, for example, a video do not change a lot in a small amount of time. While background objects do not move at all, only the objects in the foreground change during that time. DeepMon uses this characteristic to save the limited computational power of a mobile GPU. But it would not

suffice to use the total result of the last image. If we would do that a cache hit would be stale (the objects in the foreground tend to move too much) or the strong restrictions for a hit would result in a large amount of cache misses. But we could still use immediate results of certain layers if their input data has not changed. To find those layers, the image is divided into a grid. Now each block of the grid can be compared with its counterpart of the last image. If both blocks are similar enough we could just reuse the results for this block. An example is shown in Figure 5. It contains two consecutive images and shows the cached blocks in the third one. To save memory, DeepMon only saves the state of the first N layers since later layers tend to contain functions that are easy to calculate which would result in a large cache overhead [41].

The real problem now is how to determine whether two blocks are similar enough in a reasonable amount of time. Since 'normal' algorithms are too slow, DeepMon uses a modified algorithm to calculate the histogram of the colour distribution for each block and decides based on that. To do that in an efficient way the number of bins needs to be well chosen. A large number of bins results in a high cache accuracy but a low hit rate and the other way around. DeepMon usually uses 16 bins. We also need to determine a good value for the distance threshold, so that if the distance between two histograms is lower than the threshold it is a cache hit.

One edge case are filters with a size larger than 1. Assuming a 3x3 filter, if the filter is applied to a pixel at an edge it uses potentially uncached data from a neighboring block. Therefore, the pixels at the edge are recalculated based on the applied filter size. But if the neighbor block is a cache hit, DeepMon actually skips this recalculation for the pixels between these two cached blocks, since the data can now be reused.

2) *Convolutional layer decomposing:* Another more common optimizations are the decomposition of convolutional layers [41]. During that, the calculation of one layer is split into multiple lightweight layers. This allows additional optimizations and, in case of mobile GPUs, supports the caching mechanism. A tensor is mostly represented as [NxCxDxD], where N and C are the size of the input / output elements and D is the size of a filter. By using the Tucker-2 algorithm this can be split into the 3 tensors [C'xCx1x1], [N'xC'xDxD] and [NxN'x1x1]. This results in two tensors with a filter size of 1, which removes the need to handle the edge case for the edge pixels and therefore results in more cache hits.

3) Adaptation to mobile GPUs: To fit the restrictions of mobile GPUs, the amount of memory accesses and memory usage needs to be optimized. The first step is to choose a proper memory layout. A good layout would allow the kernel to read lots of data with few memory accesses. DeepMon usually uses a [NxDxDxC] layout where the input looks like [HxWxC]. This allows OpenCL to read more data with a single operation. This improves performance of most mobile devices, but not all. Some of them already have a sufficient memory bandwidth to support common optimization methods for server grade GPUs, like modelling the inputs as one big matrix and applying matrix operations on that.

Fig. 5. Example of DeepMons convolutional layer caching [8]

The last problem to solve is the variety of available mobile GPUs. Currently there exist many GPUs with different memory architectures and features. Some of them contain no local memory on the GPU itself, some contain a lot of it. To use the potential available memory, DeepMon chooses dynamically which kernel to use for certain calculations. If for example the filter of a layer fits in the local memory, a memory using kernel will be compiled and used, otherwise DeepMon will use a kernel which does not use local memory. This also works for GPUs with no local memory at all

V. PERFORMANCE

In this section we will now go over the performance of each framework. We will address how each framework performs in comparison to other frameworks like PyTorch or TensorFlow, but we cannot offer an extensive comparison of the introduced frameworks as we lack the right equipment and each of them has a specific and different purpose which they pursue (see Section 7). Nonetheless, we still can go over the performance data the original paper of each framework provided with them.

A. Tensor Comprehension

TC compares its performance with Caffe2 [42] that is now part of PyTorch and Aten [43]. Both of these frameworks are used with CUBLAS for computation. The three frameworks are compared in four different categories, the first one being for Transposed matrix multiplication. While TC is marginally faster with small matrix sizes, this is the only category TC is significantly slower than the other frameworks (up to 4.2 times slower than Caffe2 with CUBLAS) with large matrix sizes. TCs paper justifies this by stating it would be the result of CUBLAS being extremely hand tuned to operate at peak performance and utilizing register tiling while TCs performance is bound by shared memory bandwidth, because TC does not use register mapping yet.

The other categories are transposed batched matrix multiplication, grouped convolutions and productions modules consisting of lookup tables and Multi-Layer Perceptron. In each of these, TC performs better or matches the other two frameworks in performance.

B. Glow

There is close to no performance data we found on Glow except the one its paper presented. It compared Glow with TensorFlow and TVM, another compiler like TC and Glow. One has to note that TVM was compared without its autotuning capabilities. While using two popular CNN

(Resnet50 and VGG19) as the network, Glow performed 2.7 times faster than TensorFlow and 1.3 times faster than TVM.

C. DeepMon

After extensive comparisons, the authors of DeepMon concluded that it is one of the best performing frameworks for processing CNN locally on mobile architectures currently. The comparisons consist of multiple tests between the mobile frameworks basic-GPU, DeepMon and DeepX on different categories using various deep learning Models. All tests clearly show that DeepMon is faster than the other two frameworks. Against basic-GPU it is up to 5 times faster and against DeepX the state-of-the-art framework twice as fast. But these better performance results come at the cost of an accuracy loss of up to 6%, which is only 1% more than DeepX.

There are more performance tests against cloud-based approaches like remote-strong, remote-weak and edge-strong , indicating all but remote-weak being up to 2.7 times better with a stable connection. Another test showed that DeepMon also is more efficient in power consumption using 5 times less power than basic-GPU but 3 times more than remote-strong as remote-strong does not need to compute as much on the mobile devices.

The last test was to conclude if the Vulkan implementation would change DeepMon's performance, but it was equal to the implementation in OpenCL.

VI. Key Benefits and Potential Issues

Now that we have explained how each of the framework works and performs, we will now list some advantages and disadvantages each framework brings with it. The summary of the comparison of the frameworks have been presented in table 1.

TC's biggest advantage is its simplicity. Thanks to its simplicity, it is a very powerful tool to do all sorts of computation on tensors without even having to optimize the process for each architecture manually. While TC is simple to use and automates the optimization, it still allows the freedom to modify this optimization process to a certain degree. Another advantage of TC is that it can be integrated into many popular ML frameworks without enforcing a user to work with TCs implementation.

The only disadvantage is that it is only useable with Nvidia hardware since it only generates code in CUDA.

Glow on the other side is flexible and does not limit itself just to Nvidia, which is a big advantage compared to TC. If there is a chosen architecture that is not supported by Glow, one could easily implement the necessary back end for Glow. Like TC, Glow can achieve specific device optimization that were done in the back end to later use it as a reference or just as it is.

One disadvantage of Glow is that it may become hard to make use of specific features some devices provide. Since Glow lowers a lot of functions to low level instructions, it may be difficult to reconstruct or even detect these features if the device also has some special hardware for these functions.

If we look at DeepMon, we can also see some advantages. One of them for instance would be its powerful memory saving optimizations which lets the neural network perform on a useable level without relying on remote hardware. Another advantage would be its good caching mechanism that saves computational power for the computation of highly repetitive data like video streams. This also leads to the smaller power consumption mentioned in Section 5.3. DeepMon is also flexible like Glow, because it uses OpenCL.

A disadvantage DeepMon has is its unsuitability for server grade GPUs. As unique and powerful the optimizations of DeepMon are, they are nearly obsolete in server grade computation tasks. Mobile GPUs lack memory and bandwidth in a way server grade GPUs do not and the optimizations are specialized for exactly that problem. Therefore, these optimizations are just not necessary and would reduce the overall performance of server grade GPUs, but of course for these server grade GPUs, using TC or Glow would be better than DeepMon for good performance.

VII. Comparison with Other Frameworks

Apart from DeepMon, Tensor Comprehension, and Glow, it is important to have a brief look at TVM, XLA, and ONNX Runtime, among other optimization frameworks. TVM is an additional compiler-based system that, similar to Glow and TC, reduces models into many intermediate representations for optimization. Nevertheless, TVM is hardware-agnostic and incorporates autotuning, which brings it closer in scope to Glow, in contrast to TC, which is limited to CUDA. XLA (Accelerated Linear Algebra) [44] has characteristics similar to Glow's node lowering, but it focuses on operation fusion and kernel specialization. Despite this, XLA is inextricably linked to the TensorFlow ecosystem. It integrates with TensorFlow and Just After eXecution (JAX) [45].

Lastly, ONNX Runtime is a cross-platform inference engine rather than a compiler. It prioritizes portability and deployment by utilizing pre-optimized kernels, as opposed to deep graph-level compilation techniques, such as those employed in Glow or TC. These comparisons collectively underline that XLA is a leader in TensorFlow integration, ONNX Runtime prioritizes deployment flexibility, and TVM and Glow are direct competitors in compiler-based optimization.

VIII. Conclusion

In this paper we presented and briefly compared the CNN frameworks Tensor Comprehension, Glow and DeepMon. We showed that DeepMon is a promising candidate to make neural networks viable on hardware for mobile devices without the need of any remote hardware. Even if it cannot compete with Glow or TC on server grade devices it is the only one of them which performs well enough on mobiles.

Tensor Comprehension on the other hand is a suitable solution to write efficient code relatively fast since its syntax is simple and the compiler still modifiable enough to adapt it to the specific use case. However, TC only supports CUDA at the moment. Therefore, it is only usable with Nvidia GPUs.

Our last candidate Glow is a valid alternative for any GPU. Its approach is similar to LLVM and it even adopts its

syntax at lower optimization levels. Therefore, it can be easily adopted to new GPUs without Glow support, especially through node lowering. But to produce code of maximal efficiency, a lot of work has to be put into the device specific back end.

TABLE I Comparative Summary Of Tc, Glow, And Deepmon Frameworks In Terms Of Performance, Optimizations, And Limitations.

Fram-ework	Target Hard-ware	Key Optimiz at-ions	Perform -ance Highligh ts	Limitati ons
Tensor Compre hensi-on	Nvidia GPUs (CUDA)	Polyhedr al compiler , schedule trees, autotuni ng	Matches/ exceeds CUBLA S (except large matrix multiply)	CUDA-only (Nvidia only)
Glow	General GPUs/ CPUs	Multi-level IR, node lowering , quantizat i-on	2.7× faster than TensorFl -ow, 1.3× faster than TVM	Hard to exploit device-specific features
Deep-Mon	Mobile GPUs	Convolut i-on caching, layer decompo si-tion, dynamic kernels	Up to 5× faster than basic-GPU, 2× faster than DeepX	Accurac y loss (~6%), not suited for server GPUs

REFERENCES

[1] P. H. S. Panahi, A. H. Jalilvand, and A. Diyanat, "Enhancing Quality of Experience in Telecommunication Networks: A Review of Frameworks and Machine Learning Algorithms," 2024, *arXiv*. doi: 10.48550/ARXIV.2404.16787.

[2] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, in SIGMOD '02. New York, NY, USA: Association for Computing Machinery, June 2002, pp. 145–156. doi: 10.1145/564691.564709.

[3] R. Farber, *CUDA Application Design and Development*. Elsevier, 2011.

[4] R. Chandra, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[5] N. Rotem *et al.*, "Glow: Graph Lowering Compiler Techniques for Neural Networks," 2018, *arXiv*. doi: 10.48550/ARXIV.1805.00907.

[6] N. Vasilache *et al.*, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," 2018, *arXiv*. doi: 10.48550/ARXIV.1802.04730.

[7] Md. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review," *Proc. IEEE*, vol. 111, no. 1, pp. 42–91, Jan. 2023, doi: 10.1109/JPROC.2022.3226481.

[8] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, Niagara Falls New York USA: ACM, June 2017, pp. 82–95. doi: 10.1145/3081333.3081360.

[9] L. A. Torres, C. J. B. H, and Y. Denneulin, "Evaluation of computational and energy performance in matrix multiplication algorithms on CPU and GPU using MKL, cuBLAS and SYCL," May 27, 2024, *arXiv*: arXiv:2405.17322. doi: 10.48550/arXiv.2405.17322.

[10] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti, "Evaluation and tuning of the Level 3 CUBLAS for graphics processors," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2008, pp. 1–8. doi: 10.1109/IPDPS.2008.4536485.

[11] "Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs | IEEE Journals & Magazine | IEEE Xplore." Accessed: Aug. 24, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8721631

[12] N. B. Agostini *et al.*, "An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, in ICCAD '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 1–9. doi: 10.1145/3508352.3549424.

[13] R. Baghdadi *et al.*, "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2019, pp. 193–205. doi: 10.1109/CGO.2019.8661197.

[14] T. Chen *et al.*, "TVM: End-to-End Optimization Stack for Deep Learning".

[15] "DLOOPT: An Optimization Assistant on AutoTVM for Deep Learning Operators | Journal of Signal Processing Systems." Accessed: Aug. 24, 2025. [Online]. Available: https://link.springer.com/article/10.1007/s11265-022-01804-0

[16] "Ansor: Generating High-Performance Tensor Programs for Deep Learning | USENIX." Accessed: Aug. 24, 2025. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zheng

[17] "Real-time and accurate object detection on edge device with TensorFlow Lite - IOPscience." Accessed: Aug. 24, 2025. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/1651/1/012114/meta

[18] R. C. Castanyer, S. Martínez-Fernández, and X. Franch, "Integration of Convolutional Neural Networks in Mobile Applications," in *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, May 2021, pp. 27–34. doi: 10.1109/WAIN52551.2021.00010.

[19] *onnx/onnx*. (Aug. 23, 2025). Python. Open Neural Network Exchange. Accessed: Aug. 23, 2025. [Online]. Available: https://github.com/onnx/onnx

[20] Z. B. Alawi, "A Comparative Survey of PyTorch vs TensorFlow for Deep Learning: Usability, Performance, and Deployment Trade-offs," Aug. 06, 2025, *arXiv*: arXiv:2508.04035. doi: 10.48550/arXiv.2508.04035.

[21] X. Wang *et al.*, "Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models," *ACM Comput Surv*, vol. 57, no. 9, p. 228:1-228:39, Apr. 2025, doi: 10.1145/3724420.

[22] V. Shankar, "Edge AI: A Comprehensive Survey of Technologies, Applications, and Challenges," in *2024 1st International Conference on Advanced Computing and Emerging Technologies (ACET)*, Aug. 2024, pp. 1–6. doi: 10.1109/ACET61898.2024.10730112.

[23] "A survey of FPGA-based accelerators for convolutional neural networks | Neural Computing and Applications." Accessed: Aug. 24, 2025. [Online]. Available: https://link.springer.com/article/10.1007/s00521-018-3761-1

[24] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019, doi: 10.1109/ACCESS.2018.2890150.

[25] "On-Chip Training NPU - Algorithm, Architecture and SoC Design | SpringerLink." Accessed: Aug. 24, 2025. [Online]. Available: https://link.springer.com/book/10.1007/978-3-031-34237-0

[26] "ICCV 2019 Open Access Repository." Accessed: Aug. 24, 2025. [Online]. Available: https://openaccess.thecvf.com/content_ICCVW_2019/html/SDL-CV/Gorbachev_OpenVINO_Deep_Learning_Workbench_Comprehensive_Analysis_and_Tuning_of_Neural_ICCVW_2019_paper.html

[27] J. Meng *et al.*, "Automatic Generation of High-Performance Convolution Kernels on ARM CPUs for Deep Learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2885–2899, Nov. 2022, doi: 10.1109/TPDS.2022.3146257.

[28] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Not.*, vol. 48, no. 6, pp. 519–530, June 2013, doi: 10.1145/2499370.2462176.

[29] S. Imambi, K. B. Prakash, and G. R. Kanagachidambaresan, "PyTorch," in *Programming with TensorFlow*, K. B. Prakash and G. R. Kanagachidambaresan, Eds., in EAI/Springer Innovations in Communication and Computing. , Cham: Springer International Publishing, 2021, pp. 87–104. doi: 10.1007/978-3-030-57077-4_10.

[30] "Caffe | Proceedings of the 22nd ACM international conference on Multimedia." Accessed: Aug. 24, 2025. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2647868.2654889?casa_token=ddpY1R-qrfAAAAAA:idQBFzwLOAWvwmkaQREv2dJblad0gGX7aJh4j4_LPWygE101J3bWLc4g6AfX6BTkAe9tqgxqSD2EqAw

[31] A. Vedaldi and K. Lenc, "MatConvNet: Convolutional Neural Networks for MATLAB," in *Proceedings of the 23rd ACM international conference on Multimedia*, in MM '15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 689–692. doi: 10.1145/2733373.2807412.

[32] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, "A Review of Yolo Algorithm Developments," *Procedia Comput. Sci.*, vol. 199, pp. 1066–1073, Jan. 2022, doi: 10.1016/j.procs.2022.01.135.

[33] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," May 16, 2011, *arXiv*: arXiv:1005.2581. doi: 10.48550/arXiv.1005.2581.

[34] P. Plebański, A. Kelm, and M. Hajder, "Efficiency and Development Effort of OpenCL Interoperability in Vulkan and OpenGL Environments: A Comparative Case Study:," in *Proceedings of the 20th International Conference on Software Technologies*, Bilbao, Spain: SCITEPRESS - Science and Technology Publications, 2025, pp. 111–119. doi: 10.5220/0013529000003964.

[35] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule Trees," presented at the IMPACT - 4th Workshop on Polyhedral Compilation Techniques, associated with HiPEAC, ACM, Jan. 2014. Accessed: Aug. 23, 2025. [Online]. Available: https://inria.hal.science/hal-00911894

[36] S. Verdoolaege and G. Janssens, "Scheduling for PPCG," 2017, doi: 10.13140/RG.2.2.28998.68169.

[37] S. c, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 1–23, Jan. 2013, doi: 10.1145/2400682.2400713.

[38] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86. doi: 10.1109/CGO.2004.1281665.

[39] Q. Zhang *et al.*, "Intelligence Inference on IoT Devices," in *Learning Techniques for the Internet of Things*, P. K. Donta, A. Hazra, and L. Lovén, Eds., Cham: Springer Nature Switzerland, 2024, pp. 171–195. doi: 10.1007/978-3-031-50514-0_9.

[40] Z. C. Tan and A. S. Meyer, "The structure is the message: Preserving experimental context through tensor decomposition," *Cell Syst.*, vol. 15, no. 8, pp. 679–693, Aug. 2024, doi: 10.1016/j.cels.2024.07.004.

[41] X. Zhao, L. Wang, Y. Zhang, X. Han, M. Deveci, and M. Parmar, "A review of convolutional neural networks in computer vision," *Artif. Intell. Rev.*, vol. 57, no. 4, p. 99, Mar. 2024, doi: 10.1007/s10462-024-10721-6.

[42] K. Hazelwood *et al.*, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 620–629. doi: 10.1109/HPCA.2018.00059.

[43] X. Li *et al.*, "LongTail-Bench: A Benchmark Suite for Domain-Specific Operators in Deep Learning," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2022, pp. 282–295. doi: 10.1109/IISWC55918.2022.00032.

[44] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Modeling and Simulation for Defense Systems and Applications V*, SPIE, Apr. 2010, pp. 9–15. doi: 10.1117/12.850538.

[45] G. Sapunov, *Deep Learning with JAX*. Simon and Schuster, 2024.