A Brief Review of TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Nishant Malpani (IMT2016095)
International Institute of Information Technology, Bangalore (IIIT-B)

Abstract

TVM, an end-to-end compiler stack optimized to solve fundamental challenges for deploying deep learning models, bridges the gap between frameworks (front-end) and their need to deploy the models on a diverse set of hardware devices (back-end) shying away from the current ad-hoc fashion utilized by the current Deep Learning (DL) frameworks. It does so by employing both graph-level optimizations and operator-level optimizations which enables it to provide the required portability. It also uses a learning-based cost function to explore various code optimizations.

This document simply reviews the methodology adopted by the authors of TVM [1] and is an attempt to inculcate interested individuals about the in-depth internals of TVM. I believe for anyone practising deep learning its important for them to understand what's happening behind the curtains when they construct a neural-network in a high-level representation and train it on an edge device; this work is a step towards the same direction.

Keywords – *Deep Learning, Compiler Design, Optimization*

1. The Need

Given the enormous spectrum of hardware systems (CPUs, GPUs, FPGAs and ASICs) in the stage of deployment and their inherent diversity in terms of the memory architecture, the computation primitive, etc. (see Figure 1), mapping DL workloads to such embedded systems is complex and requires manual tuning for each. Current frameworks viz. TensorFlow (1), PyTorch (2), MXNet (3), etc. rely on computational graph intermediate representation to implement optimizations although, according to the authors of TVM (4), these graph-level optimizations are "too high-level to handle hardware back-end-specific operator-level transformations" (4). Many a time these graph optimizations yield new operators with no corresponding hardware primitive within the predefined operator library which obliges us to use unoptimized implementations.

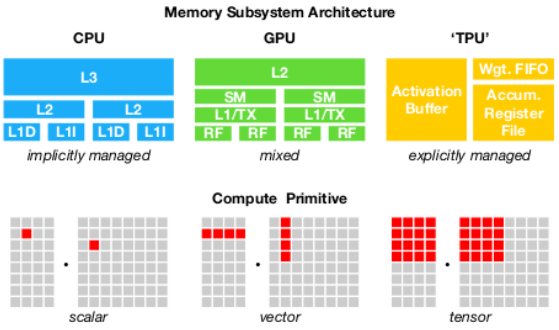

Figure 1. Highlighting divergence in memory architecture and compute primitives across various hardware systems.

Another vital point the authors generate is that DL accelerators like Tensor Processing Unit (TPU) employ static scheduling ("leaner control") rather than dynamic scheduling (5), due to claims indicating power efficiency, which offload most scheduling complexity to the compiler stack. Consequently, the compiler stack is expected to produce code such that pipeline dependencies (Structural, Data and Control) are minimized to hide memory access latency.

TVM also addresses the challenge of searching for the most optimized generated code amongst different versions of the program with various optimizations, without engaging in known approaches such as black box auto-tuning and predefined cost function. Techniques such as loop tiling, loop unrolling, caching, etc. construct a large search space of valid programs for a given operator declaration.

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin and Arvind Krishnamurthy

2. TVM's contribution

- **Data layout transformation**: transforms the computational graph to leverage the hardware features such as the memory architecture.
- **Improved operator fusion**: combines multiple operators into a single kernel to optimize memory access latency by not storing intermediate results.
- **Tensor expression language**: which supports automatic code generation and includes program transormation primitives to generate different versions of the program with various optimization for the same algorithm (attained via. compute/schedule separation).
- **Nested parallelism with cooperation**: brings in "cooperation" amongst threads to reuse shared memory in GPUs and DL accelerators.
- **Tensorization**: introduces novel instructions which exploit the underlying tensorized (not scalar or vector) hardware intrinsics present in TPUs and other DL accelerators.
- **Latency hiding**: hides memory access latency which is realised by appropriate scheduling by the compiler.
- **Machine learning based optimization framework**: explores the large schedule space and return the optimized tensor operators for each layer of a defined neural network.

3. The complete pipeline

Figure 2 displays all the components of TVM. One can note from the figure that the input model is imported in TVM as a computational graph accepted from many frameworks from the likes of TensorFlow, MXNet, PyTorch, etc. TVM, then, rewrites that computational graph to transform it into an optimized version. This is the 'high-level' optimization TVM talks about. The 'low-level' optimization, in the form of operator-level optimization further *generates* the optimal loop program where the operators are declared in TVM's tensor expression language . The low-level program is generated with the assistance provided by the code generator in TVM. Finally, once the back-end is produced, TVM packs it into a deployable module for the target hardware specified.

Tutorials on compiling models written in Python provide a great reference on using TVM's API to deploy models on a rich basket of hardware they support (6) (7).

4. High-level Optimizations

High-level optimizations constitute optimizing on the computational graph. A computational graph, commonly used

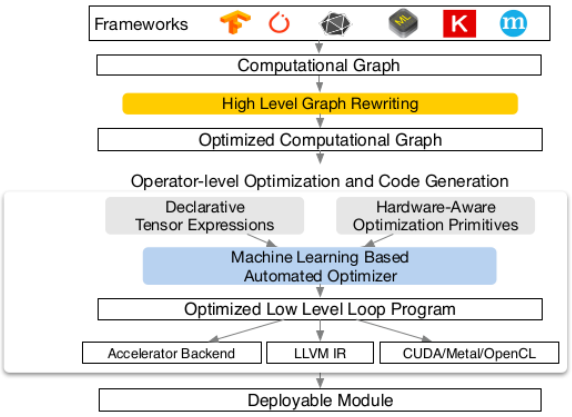

Figure 2. Lens through the TVM stack.

in DL frameworks such as TensorFlow (1), Theano (8), MXNet (3) and so on, is network of connected nodes where the nodes are represented as *operations* which could be as trivial as add to as complex as 2D convolution. These operations operate on tensors whose dataflow is represented by the directed edges of the graph. Ergo, computational graphs used to represent a neural network are Directed Acyclic Graphs (DAG).

"There are quite a few optimizations required by the VM compiler. Each of them is implemented as a pass which is managed by the Relay pass manager." (9)

4.1. Operator Fusion

Operator fusion is simple but effective idea which transforms the computational graph by modifying a cluster of nodes of the graph to fabricate a "super-node" governed by some axioms. Previously, the intermediate results had to be stored in memory causing latencies - with operator fusion, there are no intermediate results! The "super-operator" does the merged operation on its operands without triggering a cycle of memory accesses and stores of the same tensors. With the reduced memory accesses, the execution time of the program, represented as the transformed computational graph, reduces significantly.

Graph operators generally used while representing neural networks are classified in Table 1. The *opaque* operators cannot be fused with any other operators. Rules for fusing operators as adopted by TVM are:

- Multiple injective operators can be fused into another injective operator to produce an fused injective operator.

Table 1. Categories of graph operators.

GRAPH OPERATORS	EXAMPLES
INJECTIVE	ADD
REDUCTION	SUM
COMPLEX-OUT-FUSABLE	CONV2D
OPAQUE	SORT

- A reduction operator can be fused with input injective operator to yield a fused injective operator.
- Complex-out-fusable operator can be fused with element-wise operators to its output.

The above rules are laid out in Figure 3 (10) along with examples to deepen the understanding.

The source code for operator fusion explains (via. comments) the fusing algorithm. The fusion algorithm resorts to dominator trees from Graph Theory and applies *post-dominator analysis*. From the comments in the source code: "*The general algorithm is as follows:*

- *Construct a DAG of dataflow graph for dominator analysis*
- *Construct a post-dominator tree which gives immediate post dominator of each node.*
- *Run fusion algorithm with the given post-dominator information.*

The immediate post-dominator of a node defined by the closest node where all the future path goes into". The above algorithm traverses through each node in the DAG and checks if it needs to be fused to its immediate post-dominator. To comprehend dominator trees deeply and how they can be used in compiler technologies, I recommend go through the examples on its Wikipedia page - Dominator (Graph Theory) - and this YouTube video on how dominator trees are used for incremental updates in LLVM: (11).

4.2. Constant Folding

Constant folding is a popular compiler optimization which involves evaluating **constant** expressions during compile-time rather than their evaluation during run-time, as one would normally expect. Pre-computing graph parts statically saves execution costs. Constant expressions are expressions involving only literals or variables whose values can be computed at compile-time. Constant expressions such as:

$$y = 10 * 2$$
$$x = y + 1$$

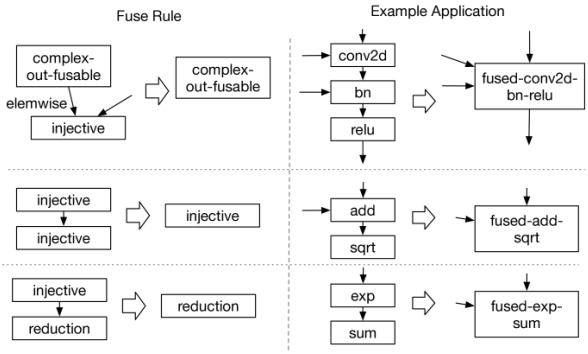

Figure 3. Rules for operator fusion.

can be "folded" statically.

Authors of TVM demonstrate how constant folding is implemented as *passes* in Relay (12) (functional, statically-typed IR) in their developer guide on 'Adding a Compiler Pass to Relay' (13). "*Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code*" (14). More on Relay's pass infrastructure used by TVM can be found in their developer guide (15). The source code on constant folding exhibits how the *ConstantFolder* **mutator** transforms the program by employing the *ContantChecker* **visitor** which traverses nodes in the graph and *checks* for constant nodes.

4.3. Static Memory Planning Pass

Implemented as a *pass*, it pre-allocates memory to hold each intermediate tensor. The source code for *MemoryPlan* pass, a derived class of *ExprMutator* discloses the details on how allocations is executed.

4.4. Data Layout Transformations

Data layout expresses the form in which data should be structured in memory and how it should be accessed - row-major order, column- major order, tiled or any other complicated ones. For example, for Graphics Processing Units (GPUs), depending on the number of SIMD processors, number of parallel warps of threads, etc. the data can be laid in a manner to optimize for both spatial and temporal locality and simultaneous execution. Tensorflow's default data layout for convolution operator in NHWC (N - batch size, H - height of a single image sample, W - width of a single image sample, C - number of channels in the image); the data is in 4-dimensions and is laid out in row-major order.

As a more concrete example, Figure 4 (10) displays how

the underlying data layout can be transformed to exploit the 2x2 tensorized operations on its data. Notice how the blue data lines in the top-right corner of the figure differ from the ones in the bottom-right corner; half-word data at 0x20 is now at 0x04 and so on.

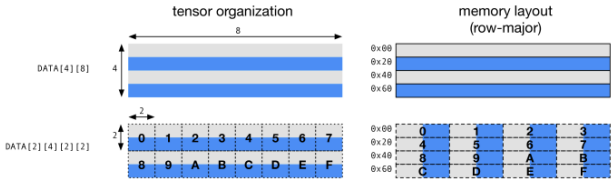

Figure 4. 2x2 tensorized operation dictating a data layout transformation.

To optimize the data layout is to transform the computational graph such that the resultant graph uses its internal data layouts deeply coupled with its target back-end. The data layout should be specified for each operator, supported by the framework and constrained by the memory hierarchy. Layout transformation also becomes critical in cases where the frameworks uses a different layout than the TOPI-supported layout. TOPI - TVM Operator Inventory (16) - provides numpy-style generic operations and schedules with higher abstractions than TVM.

TVM utilizes a Relay *pass*, *ConvertLayout* (17), to do the layout handling. *ConvertLayout* changes the data layout and the kernel (weights) layout of the whole graph rather than doing so on each operator (node) in the graph. *ConvertLayout* is called after the computation graph is parsed into Relay's representation from the framework's by a parser and before building with a *relay.build* call (17). The source code for transform layout establishes the common infrastructure for transforming the layouts. Also, note that not all operators of the computation graph rely on the inherent data layout; which is why the authors of TVM categorize the operators into 3 categories based on their sensitivity to data layouts, as captured in Table 2.

The input data layouts of heavily-layout sensitive operators are transformed while the rest of layout agnostic and lightly-layout sensitive operators adapt to the layout, realised by the *AlterOpLayout* pass in Relay, governed by the output of these heavily-layout operators, to keep the whole computation graph consistent with the same layout, as mentioned earlier. More on how the data layout is transformed can be read from the documentation.

Figure 5 visually displays how the transformation takes place. The TVM blog on 'Automating Optimization of Quantized Deep Learning Models on CUDA' (18), which is also the source of Figure 5, explains this example gracefully.

Table 2. Categories of operators based on their sensitivity to data layout.

Sensitivity	Remarks	Operator examples
Layout agnostic	Neither functionality nor performance is affected	ReLU, pow
Lightly-layout sensitive	Functionally affected but not so much performance-wise	padding, concatenate, reduce operations (sum)
Highly-layout sensitive	Affected both functionally and performance-wise	conv2d, conv2d_transpose

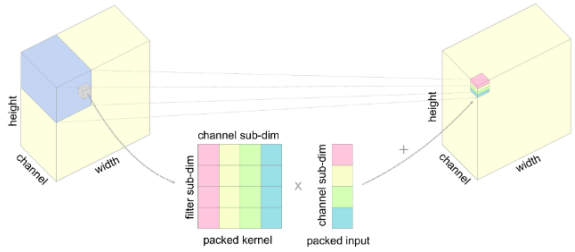

Figure 5. 2D convolution with data layout in NCHW4c and weight layout in OIHW4o4i.

5. Generating Tensor Operations

The operator library imposes limitations on, for example, the novel fused operators produced after operator fusion. The number of possible fused kernels grows dramatically based on the combinations of fused operators, data layout techniques, hardware back-end, etc. As the authors point out, a code generation approach that can generate various possible implementations for a given model's operators is the fitting direction towards the solution.

"*TVM produces efficient code for each operator by generating many valid implementations on each hardware back-end and choosing an optimized implementation*" (4). This is only possible because of the approach TVM adopts from the revolutionary Halide (19) - separating the algorithm from the execution schedule. Decisions involving intermediate storage and the order of computation, which are strictly architecture-specific, constitute the *schedule*, under Halide's nomenclature. Due to this decoupling, experimenting with the schedule to find the most optimal one is possible without modifying the algorithm hence allowing one to express many possible organizations of the same algorithm for a

wide-array of hardware back-ends.

Authors of Halide highlight a simple example to comprehend the role of picking an optimal schedule for an algorithm: "*[...] computing a first stage on the entire image before processing the second stage causes cache misses when storing and loading the intermediate results; instead, an optimized pipeline might transform the organization of computation with tiling and fusion to compute both stages at the granularity of smaller image tiles that fit in cache*" (19). The split representation introduced by Halide is captured in Figure 6.

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Figure 6. Highlighting separation of algorithm description of a 3x3 box filter from its schedule.

5.1. Tensor Expression and Schedule Space

The dataflow tensor expression language introduced by TVM is for algorithm description, as shown in Figure 7. The author claims it supports automatic code generation. It's based from languages like Halide (19), Darkroom (20) and Tensor Algebra COmplier (TACO) (21).

```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
          t.sum(A[k, y] * B[k, x], axis=k))
```

computing rule

result shape

Figure 7. Matrix multiplication: C = dot(A.T, B) in TVM's tensor expression language.

Some features of TVM's tensor expression language:

- Supports common arithmetic and math operations.
- Commutative reduction operators (sum, min, max) to schedule them across multiple threads. The official tutorials (22) by TVM features the use of *rfactor* primitive which divides the computation of reduction to be parallelized amongst threads, stores the local reduction result in a temporal array before doing a reduction over the temp array.
- High-order scan operator to describe a symbolic loop. Such operator are lucrative to model 'Recurrent Neural Networks' (RNNs) which relies on, as the name suggests, recurrent computing. The tutorials (23) emphasis with a running example of a scan operator, *cumsum*, on how such operators have an *init* and *update* placeholders and how they're scheduled on TVM.

"*Schedules are the specific rules that lower compute descriptions down to back-end-optimized implementations*" (10). So, with schedule primitives that transform schedules, one can surf the schedule space to provide different ways of generating low-level, platform-dependant code. The authors pictorially represent the various schedule primitives used in TVM, in Figure 8, with the 'schedule tree' representation derived by Halide. From the figure, we can visibly note how the schedule transforms due to the schedule primitives.

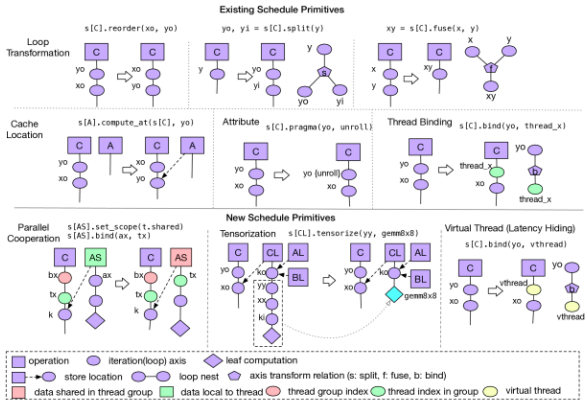

Figure 8. Schedule primitives in TVM.

The tutorial on 'Schedule Primitives in TVM' (24) takes us through the process of defining a schedule and reports stages in scheduling for each operation. An exercise I tried (which I highly recommend you do too) is to run through the tutorial of each schedule primitive and juxtapose with the schedule tree transformation in Figure 8 in order to absorb the understanding of those primitives, visually. The novel schedule primitives introduced by TVM is seen in detail in the next sub-sections.

5.2. Nested Parallelism with Cooperation

GPUs, with their SIMT (Single Instruction Multiple Data (SIMD) + Multi-threading) execution model, offer massive parallelism but they require us to create such parallel

programming models which can make use of the underlying architecture. So, since our separation of program and schedule, our design of schedule transformations should be impeccable to be able to capitalize on the target back-end. 'Shared-nothing nested parallelism', as the authors call, is a *fork-join* type of parallelism. So, if a program can be executed in a parallel manner, each of these parallel tasks can be recursively subdivided into subtasks to exploit the multi-level thread hierarchy on the target architecture (e.g., thread groups or *warps* in GPU). The name of the model comes from the fact that a thread cannot access data of its sibling withing the same parallel computation stage. Hence, the only interaction between sibling threads is during the 'join' stage when their results is merged for the next stage of schedule. The authors address this apparent limitation of "no cooperating of threads withing the same parallel stage" by introducing "cooperation" in their nested parallelism model.

Although cooperation is well-known to GPU programming languages like CUDA, OpenCL, etc., it never has been a schedule primitive, according to the authors. "Memory scopes", the answer to cooperation, is introduced by the authors to the schedule space, so that a *stage* can be marked as shared. So, a group of threads are bound (with the *bind* schedule primitive by TVM) with specified axes, that can cooperatively fetch the data they all need and place it into a shared memory space. The axis relation representation of the new, improved schedule for a program of matrxi multiplication, as shown in Figure 9 (10), shows how the *split* axis is *bound* to 'blockIdx' and 'threadIdx'. The lowered representation of the resulting new schedule in Figure 10 highlight how the compute stages - 'AS' and 'BS' - are shared. Also, note in Figure 10, the need for compiler support to append memory synchronization barriers to guarantee visibility of shared data across the consumers.

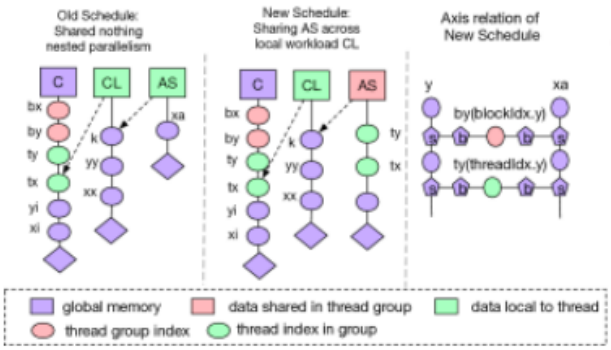

Figure 9. Schedule tree for cooperative nested parallelism.

```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
      memory_barrier_among_threads()
      for yi in range(8):
        for xi in range(8):
          CL[yi][xi] += AS[yi] * BS[xi]
      for yi in range(8):
        for xi in range(8):
          C[yo*8+yi][xo*8+xi] = CL[yi][xi]
```

All threads cooperatively load AS and BS in different parallel patterns

Barrier inserted automatically by compiler

Figure 10. Lowered code of schedule with cooperative nested parallelism.

5.3. Tensorization

Just like *vectorization* is to be realised explicitly through architecture-specific instructions for SIMD architectures, an extension of that problem is *tensorization* for specialized DL accelerators. By leveraging hardware intrinsics, one can achieve a significant performance boost for quantized operators, example is the *dp4a* instruction in CUDA which makes possible efficient computation of dot-product between two 4-element 8-bit integer vectors (18). So, with this, we can implement high-level operators such as 2D convolution (which are backed by dot-products) efficiently by using these hardware intrinsics.

The authors separate the hardware interface from the schedule for the schedule to scale to newer DL accelerators with their own tensor instructions, in the future. They introduce a tensor intrinsic "declaration mechanism" in the tensor expression language along with *tensorize* schedule primitive to replace (*lower*) a unit of computation with the corresponding tensor intrinsics. The schedule must use these primitives to benefit from the acceleration.

As seen in top part of Figure 11, "*the tensor expression language describes both the users' intended compute description, and the abstractions that the hardware exposes*" (10). The bottom part of Figure 11 shows how the schedule utilizes the just defined hardware (tensor) instrinsic, in the bottom *lowered* code. The complimentary Figure 12 shows the corresponding transformation once the schedule equips the *tensorize* schedule primitive.

The tutorial 'Use Tensorize to Leverage Hardware Intrinsics' (25) beautifully explains how matrix multiplication can benefit on an accelerator that supports matrix-vector multiplication (GEMV) as a hardware primitive by *splitting* the *matmul* loops with a factor the hardware accelerator can tensorize over. The tutorial demonstrates the defining of this GEMV tensorization intrinsic.

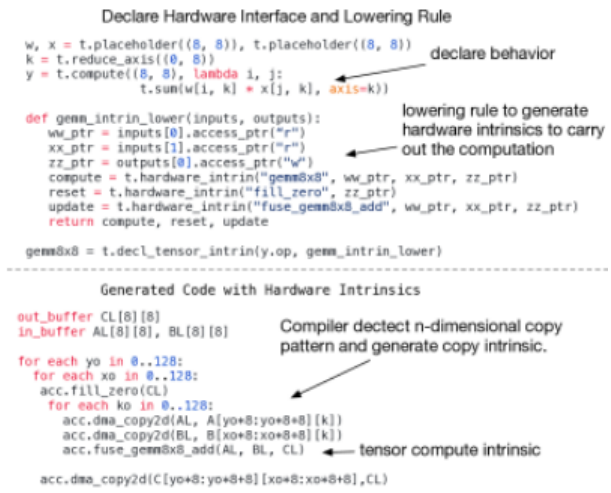

Figure 11. Using tensor expression language to describe hardware intrinsics. The code below is the *lowered* code.

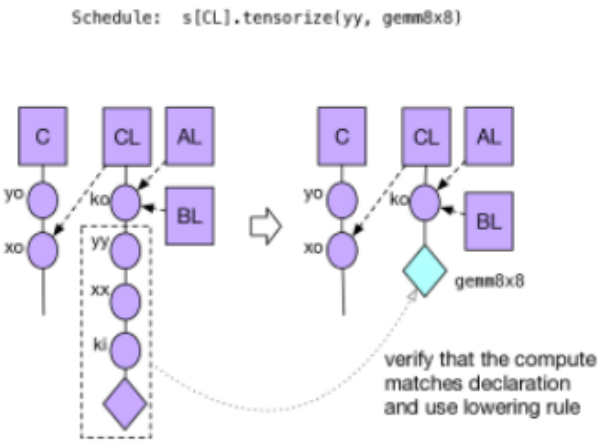

Figure 12. Transformation of the schedule tree brought in by tensorization.

5.4. Explicit Memory Latency Hiding

"*Latency hiding refers to the process of overlapping memory operations with computation to maximize memory and compute utilization*" (4). Since memory accesses are usually the most expensive in terms of latency, while an instruction is out fetching data from memory, the processor can use that idle time to compute other non-memory instructions in parallel, ergo "hiding the latency" of memory instructions. Many techniques can be utilized to hide latencies, as the authors of TVM cover - CPUs achieve memory latency hiding with simultaneous multithreading (SMT) (26) or hardware prefetching (27) (28). GPUs, with their pool of SIMT units, exercise context switching of many warps (29). Specialized DL accelerators such as TPU have a very different approach of favoring "learner" control (as discussed previously) with a *decoupled access-execute* (DAE) architecture (30). DAE provides two separate instruction streams for access operands (memory instructions) and execute operands (execution instructions) that communicate via queues. The challenge of synchronizing the two streams is handed to the programmer, i.e. the software. Figure 13 shows a DAE *hardware* pipeline, in comparison with a monolithic one, that hides memory access latencies and hence reduces the total run-time latency. To resolve dependencies between the memory instructions and execution instructions, synchronization operations, which "*sends a signal between pipeline stages to indicate when a task is completed so that the next dependent stage can start tp consume or overwrite data*" (10), must be augmented in the instruction stream and as said previously, the compiler is handed this task! Figure (30) shows how the compiler inserts such synchronization instructions in the form of dependence token enqueuing/dequeuing actions for a DAE hardware pipeline.

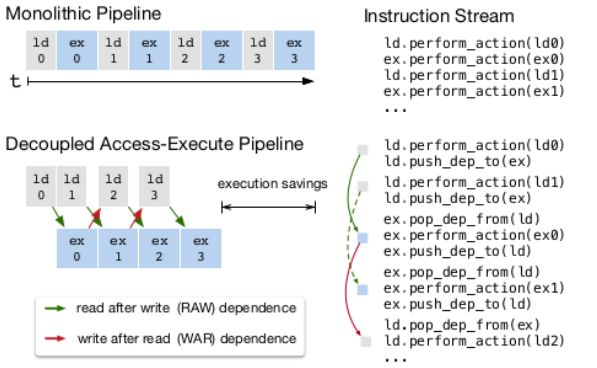

Figure 13. Example of DAE which hides most of the memory access latency.

The authors express how programming with the low-level synchronization primitives exposed by the hardware is a demanding job. In order to reduce the burden on the programmer, they introduce a **virtual** threading schedule primitive in TVM that "*lets the programmer specify a high-level data parallel program that TVM automatically lowers to a low-level explicit data dependence program*" (10). The programmer writes the high-level data parallel program thinking he's programming for a hardware back-end with a support for multithreading. TVM then lowers this program to a **single instruction stream** with low-level explicit synchronization to ensure correct no violation of the execution order within

each virtual thread. Figure 14 hints at how TVM realises virtual threading. One can see the algorithms represented with a high-level multi-threaded program schedule. The lowering process, which takes this high-level representation, "*maps the instructions of these virtual instruction streams into the limited physical instruction streams*" (10), depending on the hardware back-end. The lowering is bound to rules which ensure its error-free in terms of preserving ordering of the instructions. The appendix section A describes the necessary and sufficient condition for the correctness of the lowering process (10).

6. Automating Optimizations

The previous section introduces the rich set of schedule primitives used in TVM; now what's left is to find the optimal operator implementation using those primitives available in TVM for each layer of a DL model. The task of choosing ideal schedule-specific parameters such as the tiling size, loop unrolling factors, etc. which forms is a massive search space depending on the target back-end, needs the element of automation. To address this challenge, the authors build an *automated schedule optimizer* constituting of:

- a schedule explorer that *proposes* new parameters by surfing through the search space
- a ML-based model that *predicts* the performance by trying on those parameters for a specific hardware

6.1. ML-Based Cost Model

Ironic as it seems, the authors turn to ML to solve challenges for ML/DL. They engage in this statistical approach to model the cost which predicts the *rank* of distinct configurations based on the relative order of run-time costs. Unlike the other automation methods viz. blackbox tuning and finding the cost with a predefined cost model, ML-based cost modelling doesn't require running all configurations and measuring their performance to identify a good one (like blackbox auto-tuning) and it doesn't expect one to carefully fabricate a cost model by considering factors such as: memory access patterns, pipeline dependencies, threading patterns and so on, for each hardware target (like predefined cost model methodology). Such observations on differences amongst the automation methods are summarized in Table 3.

The gradient tree boosting model (based on XGBoost (31)), equipped by the authors, is trained using run-time measurement data collected during exploration. During the inference stage, it makes predictions based on features such as memory access count, reuse ratio of each memory buffer at each loop level and one-hot encodings to recognize loop annotations such as "vectorize", "unroll" and "parallel", extracted from the loop program (see Figure 15). Apart from utilizing the loop program to a gradient-based ML model, the authors also try an another approach of feeding the Abstract Syntax Tree (AST) of the loop program to TreeRNN (32). This DL appraoch eliminates manual feature engineering, as seen in Figure 15, however the authors resort to the former method due to lesser inference time and lesser training time. The inference time, as the authors point out, is pivotal to be under a threshold because the schedule explorer queries the ML model frequently and this overhead should be smaller than the time it takes to measure performance on real hardware, else we would not gain anything from our efforts to go along with ML-based cost modelling rather than blackbox auto-tuning.

Table 3. Comparison of automation methods.

MODEL CATEGORY	DATA COST	MODEL BIAS	NEED HARDWARE INFO	LEARN FROM HISTORY
BLACKBOX AUTO-TUNING	HIGH	NONE	NO	NO
PREDEFINED COST MODEL	NONE	HIGH	YES	NO
ML-BASED COST MODEL	LOW	LOW	NO	YES

6.2. Schedule Exploration

During the initial runs, when no prior training data exists, the schedule explorer picks random configurations for the ML model to predict on. The ML model, after a few iterations of training in an *online* manner, is now capable to deliver its predictions to the explorer which iteratively selects another batch of potential candidates (configurations) to run the measurements on. The authors employ a parallel simulated annealing algorithm (33) to explore the schedule space instead of enumerating and running through every configuration through the ML model. "*The explorer starts with random configurations, and, at each step, randomly walks to a nearby configuration. This transition is successful if cost decreases as predicted by the cost model. It is likely to fail (reject) if the target configuration has a higher cost. The random walk tends to converge on configurations that have lower costs as predicted by the cost model*" (4).

Figure 6.2 shows the complete pipeline.

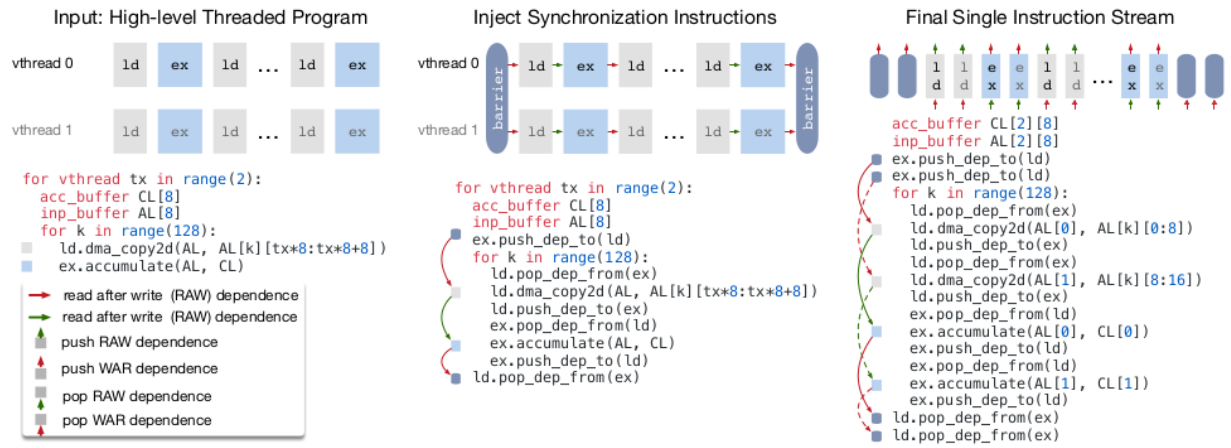

Figure 14. Virtual threading.

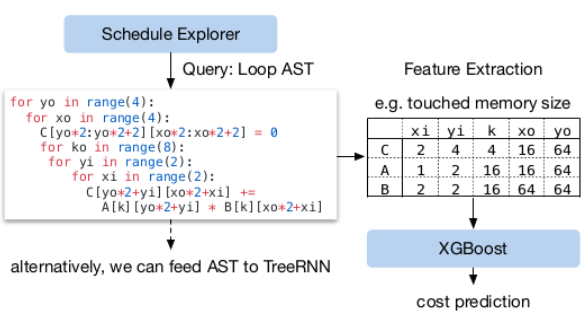

Figure 15. Example workflow of XGBoost and TreeRNN models.

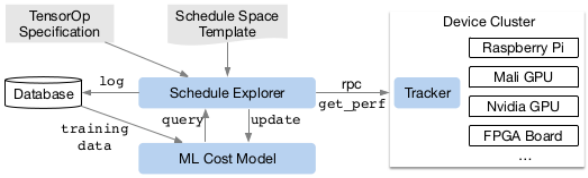

Figure 16. Automated optimization framework's workflow.

7. Evaluation

The evaluation methodology and their results are beyond the scope of this review. Please refer to the original paper (4) where the authors explore components of TVM individually to capture their performance gain over multiple platforms (server-class GPU, embedded GPU, embedded CPU and DL accelerator) in comparison to the existing frameworks (MXNet and TensorFlow).

8. Conclusion

The raison d'être of this review document was to bring in key elements of TVM together and absorb the technical concepts used to be able to reason the incredible success of TVM, an end-to-end compilation stack, on tackling optimization challenges for deep learning across a diverse set of hardware back-ends. I hope this work encourages additional such studies to get a deeper understanding of how compilation stacks, such as TVM, tap out the best from their target back-end hardware and how they evolve the domain of compiler design.

9. Future Work

- Design tutorials about topics which are not already covered.
- Participate in the community discussions.
- Improve TVM's documentation.
- Contribute to *Dive into Deep Learning Compiler* project.
- Contribute to TVM.

(The above non-exhaustive list is intended to provide direction to ones familiar with the basics of TVM, as covered in this document. It does not entail any future work on this document itself.)

Acknowledgements

I would like to pay gratitude to my professors - Prof. K K Subramaniam and Prof. Vinod Veera Reddy - for mentoring me during this credit-work which is tagged as 'Project Elective' for my 8^{th} semester at IIIT-Bangalore.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[3] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015.

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," 2018.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–12. [Online]. Available: https://doi.org/10.1145/3079856.3080246

[6] T. T. Yao Wang, "Quick start tutorial for compiling deep learning models." TVM Tutorials. [Online]. Available: https://docs.tvm.ai/tutorials/relay_quick_start.html

[7] "Compile deep learning models." TVM Tutorials. [Online]. Available: https://docs.tvm.ai/tutorials/index.html#compile-deep-learning-models

[8] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," 2012.

[9] "Putting the vm in tvm: The relay virtual machine." TVM Docs and Developer Guide. [Online]. Available: https://tvm.apache.org/docs/dev/virtual_machine.html

[10] "Tvm: End-to-end optimization stack for deep learning."

[11] J. Kuderski, "Dominator trees and incremental updates that transcend time." YouTube -LLVM Developers' Meeting, 2017. [Online]. Available: https://www.youtube.com/watch?v=bNV18Wy-J0U&t=689s

[12] J. Roesch, S. Lyubomirsky, M. Kirisame, L. Weber, J. Pollock, L. Vega, Z. Jiang, T. Chen, T. Moreau, and Z. Tatlock, "Relay: A high-level compiler for deep learning," 2019.

[13] "Adding a compiler pass to relay." TVM Docs and Developer Guide. [Online]. Available: https://docs.tvm.ai/dev/relay_add_pass.html#example-constant-folding

[14] "What is a pass?" LLVM docs. [Online]. Available: https://llvm.org/docs/WritingAnLLVMPass.html#introduction-what-is-a-pass

[15] "Relay pass infrastructure." TVM Docs and Developer Guide. [Online]. Available: https://docs.tvm.ai/dev/relay_add_pass.html#example-constant-folding

[16] "Introduction to topi." TVM Tutorials. [Online]. Available: https://docs.tvm.ai/tutorials/topi/intro_topi.html

[17] "Convert layout pass." TVM Docs and Developer Guide. [Online]. Available: https://docs.tvm.ai/dev/convert_layout.html

[18] W. Lin, "Automating optimization of quantized deep learning models on cuda." TVM Blogs. [Online]. Available: https://tvm.apache.org/2019/04/29/opt-cuda-quantized

[19] 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," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[20] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2601097.2601174

[21] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133901

[22] "Reduction." TVM Tutorials. [Online]. Available: https://docs.tvm.ai/tutorials/language/reduction.html

[23] "Scan and recurrent kernel." TVM Tutorials. [Online]. Available: https://docs.tvm.ai/tutorials/language/scan.html

[24] "Schedule primitives in tvm." TVM Tutorials. [Online]. Available: https://docs.tvm.ai/tutorials/language/schedule_primitives.html

[25] "Use tensorize to leverage hardware intrinsics." TVM Tutorials. [Online]. Available: https://tvm.apache.org/docs//tutorials/language/tensorize.html

[26] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, p. 12–19, Sep. 1997. [Online]. Available: https://doi.org/10.1109/40.621209

[27] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, p. 609–623, May 1995. [Online]. Available: https://doi.org/10.1109/12.381947

[28] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 364–373. [Online]. Available: https://doi.org/10.1145/325164.325162

[29] V. Volkov, "Understanding latency hiding on gpus." UC Berkley, 2016. [Online]. Available: https://escholarship.org/uc/item/1wb7f3h4#main

[30] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Washington, DC, USA: IEEE Computer Society Press, 1982, p. 112–119.

[31] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[32] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," 2015.

[33] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: https://science.sciencemag.org/content/220/4598/671

A. Correctness of Virtual Threading Lowering

Theorem 1. *Let $<$ be the partial order of the instructions after lowering, $S = \{(x_i, y_i)\}$ be set of all pairs of push-pop instructions before lowering. Every push message sent by the sender gets received by the corresponding pop instruction (lowering is correct), if and only if*

$$(x' > x) \implies (y' > y)$$
$$Similarly, (x' < x) \implies (y' < y)$$
$$\forall (x, y), (x', y') \in S$$

In other words, the relative order of receiver (y, y') of the synchronization message need to be the same as its sender (x, x') for each send–receive pair.

Proof. Proof by contradiction. Let a be the first sender in a physical queue to send its message to wrong receiver d. Then $\exists (a, b), (c, d) \in S$.

- $a < c$ since a is the **first** sender who sent the wrong message.
- $b < d$ because of the theorem condition.
- The above statement means b pops a message from the queue before d from some sender h (it ideally should have received from sender a), and $h < a$ due to the FIFO property of message queue.
- This contradicts the fact that a is the first sender in the queue to send to the wrong receiver (h is the first wrong receiver!).

□