CSE 598 Assignment 4: TVM and GLOW

Sreenivasan Ramesh
sreenivasan.ramesh@asu.edu

TVM: An Automated Optimizing Compiler for Deep Learning

Motivation behind developing TVM

Most deep learning framework operations are optimized for a narrow range of server-class GPUs, relying on vendor specific libraries. Another challenge is building new operators and experimenting with new fused operators that take advantage of the underlying architecture require significant engineering resources. Deep Learning frameworks rely on the computation graph intermediate representation (IR) for graph level optimizations, which are too high level to be easily converted to handle hardware specific operators. Even if we have a supported hardware backend, frameworks must choose between avoiding graph-level operations that yield new operators or using unoptimized implementations of new operators.

These vendor specific libraries are too specialized and opaque to be ported across different hardware backends. Getting these frameworks to work efficiently on mobile devices, custom hardware, IoT, etc. and designing new accelerators require a lot of engineering effort - manually writing kernels to take advantages of architectural differences as different hardware backends have different memory architectures and compute primitives.

Summary of TVM

TVM is a compiler optimization stack that exposes graph and operator level optimizations across different hardware backends. TVM takes a high-level graph specification of the program from deep learning frameworks and generates low-level optimized code which utilizes hardware specific implementations of operations. To do this TVM provides:

1. A computation graph optimization layer to rewrite high-level dataflow – operator fusion and data layout transformation.
2. Generating Tensor Operations: The *Tensor Expression Language* – A DSL to optimize tensor operations by supporting automatic code generation. The DSL supports operators like commutative reduction and high-order scan. *Schedule Space:* Since it is difficult to generate optimized implementation for a diverse backend, TVM decouples the compute descriptions from schedule optimizations (like Halide) and provides schedule rules that lowers compute rule descriptions on hardware optimized implementations. TVM also introduces concepts such as *memory scopes* (for tagging memory buffers and creating lowering rules to target specialized accelerators) and *tensorization* (to generalize the hardware interface). TVM also has support for *latency hiding* (which require different strategies for different backends) by providing primitives for virtual thread scheduling, allowing higher data parallelism.
3. Automatic Optimizations: TVM used an ML based cost optimizer, where a scheduler proposed different configurations for an operator, and an ML model takes uses the lowered loop program to predict the runtime on the given hardware. The authors evaluate the results and show their cost model outperforming a cuDNN.

Strengths of TVM

1. **Portability and Reduced engineering effort** for designing new accelerators – By providing a DSL, the Tensor Expression Language the hardware designer can declare a tensor instruction interface,

and TVM provides a system to automatically tensorize these instructions to the target hardware platform.

2. **Automatic Code Generation** – By decoupling descriptions from computation or schedule rules and extending support for new optimizations such as nested parallelism, tensorization and latency hiding, TVM can produce efficient code for any operator by generating many valid implementations on different devices.
3. **Better performance on emerging workloads** compared to cuDNN, such as Transformer related workloads, probably due to hand-tuned optimizations not yet being available for these workloads.
4. **Accurate Cost Models** for designing new accelerators – Versatile Tensor Accelerator (VTA) provides a RISC-like abstraction to expose tensor operations, compute and memory arbitrations for the hardware. TSIM is a simulation environment, that takes the hardware description from VTA to provide more accurate performance costs models, helping in the development process of new accelerators.
5. **Support for non-static computation graphs** (WIP as of 2020) such passing trees, sequences, etc. Using the relay virtual machine, extensions can be added to the runtime environment to tackle programs containing control flow and other advanced operations, by providing support for runtime data structures allowing addition of new data structures without having to change the system or the compiler

Weaknesses of TVM

1. **Poor performance against vendor specific libraries** such as TensorRT on commonly used workloads: TVM still cannot outperform TensorRT which converts subgraphs to single TensorRT operations which work on hand-tuned cuDNN libraries.
2. **Cross Language Support** - TVM currently has a non-unified runtime object protocol for objects like AST or IR nodes, NDArrays, Tuples, Closures, etc. where each of them have their own runtime data structure, limiting the amount of cross-language support.
3. **No Support for Dynamic Shapes** – TVM does not have support for computations with dynamic shapes (tensor shapes unknown at runtime), which are some of the most common cases with RNNs. Further, there is currently no documentation/examples for RNN models.

Glow: Graph Lowering Compiler Techniques for Neural Networks

Motivation behind developing Glow

Similar to the motivations behind TVM, Glow was designed to better utilize hardware backends as most deep learning frameworks are not optimized to fully take advantage of heterogeneous hardware. Most frameworks iterate over the computational graph node after node, and this inefficient use of hardware architecture has led to the creation of ML compilers such as XLA and TVM.

Supporting heterogeneous architectures and new operators is not scalable as device specific implementation is required. Glow is designed to take a computational graph, optimize the graph and auto generate device specific code for heterogeneous devices. Unlike other ML compilers, Glow includes target independent optimizations that are performed lowering the graph on high-level IR, prior to hardware specific code generation. This allows for additional graph-level optimizations that cannot be performed by compilers such as TVM. An added advantage is reducing the number of device optimized linear algebra operators that need to be implemented. Glow also works on the ONNX representation for graphs, allowing for optimization across other frameworks such as mxNet and CNTK.

Summary of Glow

Glow generates optimized backed specific code, by first lowering the computational graph into a two phase, strongly typed IR. The graph is differentiated, and the high-level IR is optimized, and then the nodes are decomposed from high level IR to linear algebra operators. The main advantage of doing this is to limit the number of specialized operators that need to be implemented. More optimization is done after this, both target specific and independent. The graph is then scheduled into a linear sequence of nodes to minimize the memory, and using *IRGen* low-level IR is generated, after which low-level IR optimizations are done. Finally, the backend specific optimizations are performed, and the optimized code is generated.

Glow's optimizations provide support for profile guided quantization, compiler optimization for quantization and operator stacking. Glow also provides a runtime environment that executes models across multipole devices. A *partitioner* partitions the model into smaller sub-graphs, a *provisioner* assigns subgraphs to devices, a *device manager* compiles the sub-graphs and an *executor* which handles the execution of the network.

Strengths of Glow

1. **Reduced engineering effort for new operators** - Since Glow decomposes complex operators, the hardware designers can focus only on efficient implementations for a limited set of core operators.
2. **Multiple optimizations** at different IR levels – Since Glow performs optimizations on the highlevel IR, it is able to **perform domain-specific optimizations** which are not done by XLA or TVM.
3. **Support for multiple frameworks** - since Glow uses ONNX representation for the dataflow graph, it can work for other frameworks such as mxNet, CNTK, etc. It can also be made to work with TensorFlow models after converting them to ONNX representations.

Weaknesses of Glow

1. **Performance against TVM** - Though Glow outperforms, XLA, it is only able to match TVM's performance when autotuning and improved schedules are turned off for TVM.
2. **No support for asynchronous model execution** - The glow runtime does not support concurrent execution, pipelining, batching, and so on. This requires modifying the current execution engine and implementing an asynchronous mechanism which can execute code on multiple accelerators.

3. **No support for non-static graphs and control flows** – Currently RNNs and LSTMs are supported by unrolling the network. To truly support dynamism, Glow's internal data structures need to be extended, and control-flow aware automatic differentiation must be implemented.

References

[1] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. ArXiv 1802.04799, 2018.

[2] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. ArXiv:1805.00907, 2018.

[3] Control Flows, Github Issue, https://github.com/pytorch/glow/issues/2738.

[4] TVM and Deep Learning Compilation Conference, https://sampl.cs.washington.edu/tvmconf.

[5] Glow Runtime, Github Issue, https://github.com/pytorch/glow/issues/2045