TVM PAPER REPORT

~Shreyas S (EP22B040)

Question 1. Provide a short (100-word) summary of the paper.

This paper introduces TVM, an automated end-to-end compiler stack designed to make deep learning models run efficiently across different hardware platforms like CPUs, GPUs, and custom accelerators. Instead of relying on vendor-specific, hand-optimized operator libraries, TVM performs both graph-level and operator-level optimizations. It uses a tensor expression language to flexibly describe operations and applies schedule transformations such as tensorization and memory latency hiding to match hardware capabilities. A key contribution is its ML-based cost model that searches large optimization spaces automatically. Overall, TVM achieves competitive or better performance compared to manually tuned libraries across diverse devices & workloads.

Question 2. What did you learn from the paper?

- Most existing DL frameworks depend on libraries like cuDNN, which are manually optimized for specific GPUs. Because of this, deploying the same model to mobile GPUs, CPUs, or FPGAs typically requires hand-written kernels, making portability painful.
- TVM represents an operator using a tensor expression and it separately applies schedules. This separation allows the same mathematical operation to be optimized differently for each hardware backend.
- Hardware like TPUs or certain GPUs have fixed-shape matrix compute instructions (e.g., 8×8 GEMM units). TVM lets us define these intrinsics and automatically replace loop segments with them as it is more flexible than normal vectorization.
- The paper explains how TVM schedules GPU threads to cooperatively load tiles into shared memory and insert barriers.
- On TPU-like designs, memory access and compute are decoupled, so the compiler must insert synchronization instructions to overlap them.

Question 3. What did you like about the paper?

- The paper presents a clear and modular design that separates computation from scheduling and hardware execution. This structure provides a clean abstraction that accommodates diverse back-ends without rewriting core logic. The extension of Halide's concept to tensor computations feels well-motivated and practically useful for supporting new and emerging accelerator architectures.

- The integration of an ML based cost model for schedule search is a strong design choice. Instead of relying on hand crafted heuristics or expensive brute force tuning, the model continually improves as it gathers performance data. This makes the system more scalable and future proof as hardware grows increasingly complex and heterogeneous.
- The evaluation is comprehensive and convincing. By benchmarking TVM across server GPUs, embedded CPUs, mobile GPUs, and FPGA accelerators, the paper demonstrates performance portability rather than optimizing for a single platform. The fact that TVM matches or exceeds manually optimized libraries in many cases highlights the system's practical effectiveness and maturity.

Question 4. What did you dislike about the paper?

Since I am completely new to the compiler space, this paper served more as an introduction to the domain for me and I don't think my knowledge is mature enough to dislike any of the ideas presented here

Question 5. In class, we discussed a few popular deep learning compiler frameworks. What are TVM's advantages and disadvantages when compared with them?

Advantages:

- TVM covers both graph-level fusion and low-level kernel scheduling, unlike MLIR/XLA which separate these concerns.
- TVM's tensor expression + schedule language allows fine control over tiling, memory layout, and tensorization, which is harder to express in XLA or Triton.
- TVM can automatically search efficient schedules across hardware, while Triton often expects the user to write optimized tile level kernels manually.

Disadvantages:

- The scheduling language is complex and can require deep understanding of hardware.
- Autotuning provides portability but requires lengthy search runs, unlike XLA's mostly static compilation.
- MLIR is designed as a shared compiler infrastructure
    - TVM remains more self-contained, limiting interoperability.

Question 6. The TVM infrastructure's development appears inactive now with the lead authors hired by a major company. Let us say you are tasked with the continued improvement of TVM. What are some features you will add to it? Why?

- The original TVM work focuses mainly on CNN-style models. Modern workloads rely heavily on attention, KV-cache management, quantized MatMuls, and dynamic sequence lengths. Adding optimized operator templates and scheduling strategies for transformer blocks would make TVM directly relevant to today's deployment scenarios.
- Current quantization flows require external libraries or manual tuning. A more integrated pipeline: covering calibration, quantization-aware scheduling, and hardware-specific low-bit tensorization would improve inference performance on mobile and edge hardware.
- While the ML based cost model is powerful, tuning is slow in practice. Improvements like knowledge distillation across models/devices, learned schedule priors could significantly reduce tuning time without sacrificing performance.
- MLIR has become the common IR layer across many compilers. Exposing TVM's schedule IR or hardware backend lowering as MLIR dialects would improve interoperability and reduce maintenance effort, while still preserving TVM's flexibility.
- Triton provides a cleaner mental model for writing tile-level GPU kernels. Incorporating similar tile abstractions or a higher-level scheduling DSL could lower the barrier to writing optimized GPU schedules in TVM.