CSE-234: Data Systems for Machine Learning, Winter 2025

LLM deployment across cloud and edge with ML Compilation

Lecturer: Tianqi Chen

Scribe: Brandon Le, David Min, Jeffrey Liu, Chengcheng Zhang, Kaiyuan Hu, Yue Pan, Mrinaal Dogra, Yifei Shao, Lakshya Goyal, Jerry Thomas John, Thanh-Nha Tran, Rami Altai, Justin Washabaugh

1 History of Machine Learning Revolutions

Machine learning has undergone several transformative waves, each introducing new capabilities that have shaped the field:

1.1 Big Data (2010)

The first major wave in 2010 was driven by big data. Competitions such as the Netflix Prize challenged participants to build models predicting user preferences based on large-scale datasets. This initiated the era of data analytics and recommendation systems, which are still highly relevant today.

At this time, various machine learning systems emerged to support these analytical tasks, including:

- **Apache Spark** – A large-scale data processing framework.
- **XGBoost** – An efficient implementation of gradient boosting for structured data.
- **GraphLab** – A machine learning framework for scalable graph-based computations.

1.2 Deep Learning (2013)

The second major wave arrived in 2013 with AlexNet, marking the rise of deep learning. While neural networks had existed since the 1990s, this period saw their large-scale deployment, significantly advancing pattern recognition.

Deep learning models were now capable of:

- Recognizing complex visual patterns in images.
- Understanding and generating speech from audio data.
- Handling multi-class classification beyond binary outputs.

Key machine learning systems enabling deep learning during this period included:

- **TensorFlow** – Google's deep learning framework.

- **PyTorch** – A widely adopted deep learning library known for its flexibility.
- **TVM** – A deep learning compiler for optimizing model execution across various hardware platforms.

1.3 Generative AI (2023)

The most recent revolution is the advent of **Generative AI**, characterized by:

- Open-ended content generation, including text, images, and audio.
- Generalist AI models capable of performing multiple tasks.
- A growing demand for optimized acceleration and computational efficiency.

Notable models in this era include OpenAI's GPT-based models, Google's Bard, and Meta's Llama 2. These models have high demands for memory and compute, and ML systems design plays a key role in their deployment and maintenance now more than ever.

2 Systems for Generative AI: Challenges and Opportunities

Generative AI introduces several key system-level challenges:

2.1 Memory

Large-scale models such as **Llama-70B** require significant memory, consuming up to **320GB of VRAM** to store parameters in `fp32` precision.

2.2 Compute

Unlike earlier ML models, generative AI demands highly specialized computing resources. The **post-Moore era** has led to:

- Increased reliance on hardware accelerators (e.g., TPUs, GPUs).
- The need for extensive software optimizations to efficiently leverage available hardware.

2.3 Integration

Modern AI applications extend beyond single chat models. They now encompass multimodal capabilities, integrating vision, audio, and text processing. Coordinating multiple models and system components remains a challenge.

2.4 Evolutions and Co-Design

To keep pace with rapidly evolving AI models, continuous innovation in hardware and software co-design is necessary, and systems need to be able to quickly adapt to these changes.

3 Democratization of Generative AI: Cloud and Edge

3.1 Current Generative AI Paradigm

Today, most generative AI models operate in the cloud. Users send data to centralized providers, where models run on large-scale computing clusters and send back the results. These clusters usually run on a single hardware backend.

3.2 Typical Engineering Approach

In the current ML ecosystem, machine learning systems rely on specialized stacks such as NVIDIA, AMD, and TPUs. These stacks require:

- Specialized libraries and systems for each backend, making optimization labor-intensive.
- Manual, non-automatic optimizations, requiring significant engineering effort.

To streamline this process, ML compilers are designed with multiple layers of abstraction to efficiently map high-level models onto hardware.

3.3 Universal Deployment Runtime

To support diverse hardware backends, modern ML compilers focus on:

- Code generation and library dispatching, enabling efficient execution across different platforms.
- Composable optimizations, such as operator fusion, parameter sharding, and memory planning.
- Backend-aware execution, which incorporates layout transformations and low-level optimizations.

A universal deployment runtime aims to bridge the gap between ML models and hardware, ensuring efficiency and portability.

3.4 ML Compilation and System Stack

When building ML compilers, a key design principle is constructing multiple layers of abstraction to represent ML computation. The system stack typically consists of:

- **Computational Graphs:** At the highest level, ML models are represented as computational graphs. Each node in the graph represents high-dimensional tensor computations, and these nodes can be composed together to form neural networks.
- **Tensor Programs:** At a lower level, tensor programs define parallelization strategies for loop execution in kernels. This is particularly useful for layout transformation and loop optimization.
- **Libraries and Runtimes:** Built by vendors and engineers, these libraries optimize key operators of interest and help accelerate computations.
- **Hardware Primitives:** At the lowest level, specialized hardware primitives, such as NVIDIA Tensor Cores, expose low-level hardware features required for acceleration.

3.5 Challenges and Continuous Evolution

One of the biggest challenges in ML compilation is ensuring continuous improvement as new models and system innovations emerge. Unlike traditional compilers, ML compilers must be designed for rapid evolution.

- **Scalability:** As models grow in size and complexity, maintaining efficiency across different hardware stacks becomes increasingly difficult.
- **Optimization Bottlenecks:** Techniques such as sparse weights, quantized kernels, paged attention, and layout optimization require significant engineering effort.
- **ML Engineering and Systems Integration:** ML engineering now plays a crucial role alongside ML modeling. The goal is to enable continuous innovation.

3.6 Future: Bringing Generative AI to Consumer Devices

A key research question is whether generative AI models can be efficiently deployed on personal devices, similar to how personal computers evolved from mainframes. This shift would reduce reliance on cloud providers and enable:

- Privacy-preserving AI models running locally.
- Optimized AI inference on consumer hardware (e.g., smartphones, laptops).
- Lower latency by eliminating cloud-based processing delays.

4 TVM Unity Introduction

TVM Unity allows users the opportunity to optimize machine learning models and deploy them efficiently across various hardware backends. By providing a flexible and modular compilation framework, TVM enables seamless integration and performance optimization.

- **Python-First Compilation:** TVM adopts a Python-first compilation approach, leveraging Python's flexibility, readability, and extensive ecosystem to enhance accessibility, modularity, and adaptability in the compiler stack.
- **IRModule:** The IRModule serves as a core component in TVM, managing and transforming intermediate representations of programs. It encapsulates the computational graph, tensor programs, hardware primitives and their interactions providing a single central abstraction. It thus acts as a central hub where computations are collected, optimized, and compiled into highly efficient executable code.
- **Universal Deployment:** Beyond optimization, TVM is designed to support universal deployment across diverse hardware backends. By providing a unified compilation framework, it ensures broad compatibility and efficient execution on CPUs, GPUs, accelerators, and even edge devices. TVM is thus able to convert its abstractions to a native program that can run on different environments from basic Python to C++ to Javascript for the web.
- **Continuous Improvement:** TVM is an evolving ecosystem that continuously introduces new features, optimizations, and libraries. With active community contributions and ongoing research, the framework is regularly enhanced to meet the ever-changing needs of machine learning workloads.

5 TVM Unity

5.1 First-class Symbolic Shape Support

Traditionally in convolutional neural networks, the shapes of intermediate tensors, operator inputs, and outputs, are known and fixed. For example, it is a common practice to perform image transformations such as cropping and centering on the input image before feeding it into a CNN for image classification. In this scenario, the compiler faces the *static shape* problem, and the staticness of shapes is an important assumption of ML compilers of the prior era. However, language models present a different challenge, as the shape is growing as generation proceeds. The K and V matrices increase by one in the sequence length dimension at a time as a new token is generated and added to the KV cache. The same applies to the batch size in dynamic batching scenario.

Currently, machine learning frameworks such as PyTorch support dynamic shapes through the `?` notation for tensors. For example, a computation graph may contain `%conv : Tensor(?, 64, ?, ?)`, which indicates that the conv operator processes dynamic shapes with only the second dimension known as 64.

However, a significant disadvantage of this notation is that there is no relation information between the `?` shapes, while in reality two dynamic shapes can be equal to or multiple of each other, which brings opportunities in compiler optimizations. TVM unity uses *Symbolic Shape*. *Symbolic shape* denotes dynamic shapes with variables and trackable relationships. For example, two operators with symbolic relationships in their shapes can be expressed with,

```
%op1 : Tensor(n, 64)
```

```
%op2 : Tensor(n * 2, 64)
```

This allows shapes to be a part of the computation and more optimizations to be inferred, enabling certain static optimizations that are otherwise impossible with pure dynamic shapes. This includes *static memory planning for dynamic shape*, *dynamic shape-aware operator fusion* and *layout rewriting and padding*. In the worst case, this notation falls back to the equivalence of `?` shapes.

5.2 Composable Tensor Program Optimizations

Composable Tensor Computation is a technique that enables modular, adaptable, and hardware-efficient tensor processing through structured abstractions like **TensorIR**. This approach optimizes computation, scheduling, and deployment across various hardware architectures. The key idea of TensorIR is to divide the computations into sub-tensor computation *blocks* and generalize the loop optimization for these computations, providing a divide and conquer strategy compared to existing bottom-up where the representation of computational blocks is difficult and top-down approaches where loop optimizations are harder to detect.

Through these features, TensorIR is thus able to provide a structured approach to defining and optimizing tensor computations, ensuring efficient execution across various hardware platforms. It introduces blocks that encapsulate computation units with explicitly defined read and write dependencies, enabling better organization and modularity. Additionally, TensorIR supports loop transformations, memory optimizations, and hardware-aware scheduling, allowing for enhanced performance and adaptability to different computing architectures. TensorIR functions can be defined in TVM Unity and called via destination passing.

Loop nest transformations improve parallelism by reordering, splitting, and binding loops efficiently. **Memory planning** optimizes data layouts and enables static planning for dynamic shapes, ensuring efficient memory usage. **Operator fusion** reduces overhead by merging multiple operations, leading to faster and more efficient execution.

Analysis-Based Program Optimization

This approach applies **static** analysis to understand program structure and dependencies before applying optimizations. It allows for:

- Loop analysis and transformations, enabling efficient scheduling and execution. This ensures that loops are optimized for parallel execution and memory access patterns, minimizing overhead and maximizing throughput.
- Automated optimizations such as fusion, memory planning, and layout transformation. These optimizations reduce redundant computations, improve cache efficiency, and tailor data storage to better align with hardware capabilities.
- Improved performance and portability by adapting computations to different hardware architectures. By analyzing the program's dependencies, the optimizations are adapted to leverage the strengths of various platforms like CPUs, GPUs, or specialized accelerators.

Imperative schedule transformations

Imperative schedule transformation provides a step-by-step, **interactive** way to modify tensor computations. It allows:

- **Explicit control over transformations** like **loop reordering, tiling, and fusion**. This granular control enables developers to fine-tune computations for specific hardware, enhancing both speed and resource utilization.
- **Modular and extensible scheduling**, enabling developers to introduce new optimization primitives. This flexibility allows for custom scheduling strategies that can be adjusted as new hardware features or performance metrics become available.
- **Efficient tensor execution** by isolating tensorized computations and mapping them to specialized hardware efficiently. By isolating tensor operations and mapping them to hardware that excels at these tasks, computations can be executed with maximum efficiency and minimal latency.

Why Composable Tensor Computation Matters?

- Bridges the gap between ML modeling and ML engineering, ensuring models run efficiently across platforms.
- Enables continuous optimizations as new hardware and software advancements emerge.
- Helps in scaling AI applications beyond cloud environments to edge devices and consumer hardware.

5.3 Unifying Libraries and Compilation

Previously, there existed a significant boundary between library-driven and compilation-driven AI systems. The current trend focuses on utilizing the best of both worlds to create a combination of library-based and compilation-based approaches.

- **Abstraction to Unify Libraries and Compilation:** With *call_dps_packed*() function inside modules, users may call into runtime library function registered via TVM FFI (Foreign Function Interface).

- **Works to Unify Libraries and Compilation:** Users can map the sub-regions of computations, i.e. existing computation implementations (compilation) onto external libraries. For example, Relax-BYOC can offload Conv2D operator calls onto TensorRT which has better optimization on NVIDIA GPUs. External libraries can be other compiler and runtime frameworks. Users can also use library-based offloading and native compilation together. TVM makes it simpler to implement this through computational sub-region detection.

By combining the best of both worlds together, users are able to get high-performance libraries work collaboratively with compilation. The sweet spot between compilation and library is flexible and needs to be updated to fit specific computational sub-regions.

6 MLCEngine

MLCEngine is a domain specific compiler built specifically for large language models. It's self-described as a "Universal LLM Deployment engine" that can support LLM deployment on a wide spectrum of devices, ranging from cloud servers to embedded and edge devices. In most existing approaches, GPU kernel libraries are manually crafted by experts for each supported backend. However, such an approach is often impractical to scale and upkeep per backend due to the quick pace of advancements in ML models and hardware accelerators. MLCEngine, on the other hand, is able to achieve universal deployment by leveraging retargetable compilations that enables it to generate device specific (optimized) code. It also has an OpenAI-Compatible server that allows a user to be able to interact with other OpenAI-based applications.

There are a few notable examples and use cases that allow users to see the potential of MLCEngine right away. Firstly, there is an iOS Swift SDK that allows users to directly run LLMs on iPhones, including OpenAI-style Swift APIs. Moreover, there is an iOS app, called `MLC Chat`[1], which allows users to chat with open language models locally on their iPads and iPhones. Similarly, there is also an Android SDK and a demo Android app[2] with similar capabilities, which currently supports larger models than the iOS SDK. The MLCEngine also supports the Steam Deck, a Linux based gaming handheld device.

An interesting aspect to highlight is MLCEngine's ability to target different environments and hardware architectures. For instance, on iOS, it leverages the GPU on the device by utilizing Apple's Metal langauge to program the GPU. On Android, it uses the OpenCL framework, while for Steamdeck, it utilizes the Vulkan backend to support LLM deployment right out of the box.

7 Efficient Structured Generation

In order for LLMs to be more than chatbots and have more capabilities that involve calling functions or interacting with other programs, their generated output must satisfy structural constraints. In other words, a model should be able to generate structured data as well as plain text.

- **Purposes and Usage:** Structured outputs can be used to call search engines, weather apps, or other tools. In order to pass the output of an LLM, it must comply with the input format of the secondary tool. For example, a json formatted output containing the name of a country and its capital could be used to call a weather app.

[1]https://apps.apple.com/us/app/mlc-chat/id6448482937
[2]https://github.com/mlc-ai/binary-mlc-llm-libs/releases/download/Android-09262024/mlc-chat.apk

- **Advantages:** Passing the output of an LLM becomes much easier than with plain text. It is also more reliable than prompting strategies, since structured generation is guaranteed to satisfy the structural constraints.
- **XGrammar:** XGrammar is a library that supports context-free grammar to create generative outputs that conform to structural constraints without harming a model's inference. Its other benefits include that it does not slow generation, has near-zero overhead, and has already been integrated into other existing libraries.

8 WebLLM

WebLLM is an application that allows users to locally run LLM's through their browsers by combining the MLC LLM engine with WebGPU. WebGPU is a new standard that can access the GPUs of users' machines through the browser.

The way WebLLM works is that when the user first tries to access a model on the webpage, it will download the model's weights into the browser's cache from the web. While this may take some time, it allows future calls to the model to be executed incredibly quickly, as it can directly load the weights from cache.

One of the benefits of this approach is that because the model and its weights are all local, no user data is sent over the network. This is useful for people concerned with privacy or who operate on private data. Furthermore, WebLLM allows for easy integration with applications that are built on top of it by providing a TypeScript API and npm package, which works identically to OpenAI's API.

You can try this application for yourself at this link: https://webllm.mlc.ai/

9 Q&A

- **Will WebLLM work if there is no GPU in your local machine?** Unfortunately, it will not work because WebLLM's functionality is specifically designed to use WebGPU. However, there is WebAssembly that can attempt to leverage the CPU on local machines instead. However, you'd need to build this WebAssembly backend.
- **If we plan to use the local GPU, wouldn't an app be more user friendly?** Initially, they do not want to create an opinion on what firmware would be considered better, so they just started with offering an API. Additionally, an advantage of using the web is it is easy to get started on and share with other people.
- **What's the motivation for this project? If we can run the models locally, wouldn't relying on using the website as an interface remove the advantage of not needing internet connection to run the models?** In this case, it is important to note that you are only downloading data from the internet, nothing is being sent out. If the website has been cached, you can actually visit the website without having access to the internet.
- **How did you come up with XGBoost idea at that time?** XGBoost originated from some research needs that was needed at the time. They were working on a machine learning project that required gradient boosting, but the solution was not that fast. As such, they built XGBoost for that project. Later, they discovered it would be useful for other developers, so they packaged it as open source code.

- **What are the current limitations of TVM?** One limitation is that hardware keeps evolving, for example, tensor core-based NVIDIA GPUs. There are also new compiler solutions, like the Pytorch compiler and OpenEye Triton, which all have their own advantages that TVM can learn from. They can also bring in the Python first approach to be able to do more fun customization development.
- **Since the function of ML compilers is to generate efficient code and LLMs are gaining stronger ability of inference and coding, will we one day be able to utilize LLMs to optimize themselves?** This topic is definitely something to look out for, and having ML compilation frameworks might make this more possible.
- **Can we utilize MLC for these specialized small models across various devices?** MLC is a reasonable option if you can get a 1-7 billion model to work. Another thing you may want to do is think about the one-solution-fits-all case, where the same model can work across all environments. If we start with supporting the Python code, server, and the API, it will make it much easier to put it onto such environments.
- **With the LLM running on a web browser are there any safeguards so that malicious attackers cannot exploit the system to trigger expensive computations and cause Denial of Service attacks?** In this case, it is running on the web browser, but it is not being hosted on the server. Once the user gets the weights, they are just running the computations locally. As such, it would not cause a DoS issue.
- **How does this compare in terms of performance compared to Groq or Cerebras?** Both of those still run on the server, so you have to be okay with sending your data away.
- **Where do you see the future of ML compilers heading?** There is still a bit of uncertainty in the area. You are starting to see frameworks adopting compiler technology. It is becoming a base layer that every framework needs to consider. It is possible that the term will start to disappear, as frameworks start to adopt it as a default. Some interesting questions are how do we make it more user-friendly, and how do we use similar methods to optimize elastical distributed workloads.
- **How are browser cache able to store weights of Deepseek for us to use offline?** WebLLM will cost about 4 gigabytes, and the browser cache can usually store up to your disk size.
- **Does WebLLM provide substantial benefits over the traditional cloud-based inferences that people use today?** If you are looking at local first and running it on no internet connection or if you care about restricting your data, then WebLLM will be of interest. The switch from a local API and a server API is also not that bad. Essentially, WebLLM gives you another choice.
- **Do you plan to apply MLC LLM idea in serverless LLM?** There are some optimizations in serverless that might help look at how to do pipelining of weight loading together with computations. In these cases, you can start to craft your own solution for an influence pipeline and do some of the computations in compiler transformation.
- **With the emergence of deep-seek or smaller models, how do you predict NVIDIA's future in terms of demand?** They can't comment too much, but computing will still be essential in a lot of models.
- **From a business perspective how would using something like WebLLM improve factors like UX and finances/resources?** There are two things to consider. One is data protection, where you may not want to send your information over the server. The other is efficiency and cost because the personal compute will add up.
- **What would be the top three skills for building a career in ML Systems?** One principle that is useful is to remember is that viewing all the modeling techniques, computing, scaling, and data, together is important to solve a problem.