15-618 Final Report - Accelerating GPT-2 Inference with TVM

Rick Zhou, Charlie Ruan
{xzhou2, cfruan}@andrew.cmu.edu

1. Summary

In this project, we accelerate the inference of an autoregressive decoder-only transformer. Specifically, we leverage TVM [1], a domain-specific programming language for deep learning, to optimize the inference of GPT-2 (medium-size) [2], a 355 million parameter model. To optimize GPT-2 with TVM, we first implement our workload in the framework MLC LLM [3]. With the predefined optimization schedule, we compare a TVM-accelerated inference with the Huggingface version on different platforms – RTX 4090 and Apple M2 Ultra. Then, we analyze three of the operator-level optimizations performed by TVM – matrix-matrix multiplication (GEMM), matrix-vector multiplication (GEMV), and reduction – benchmarking their effect on the performance. Finally, we specifically look at GEMV, analyze how it leverages a parallel architecture, identify the tunable hyperparameters, and further tune its performance according to our specific workload.

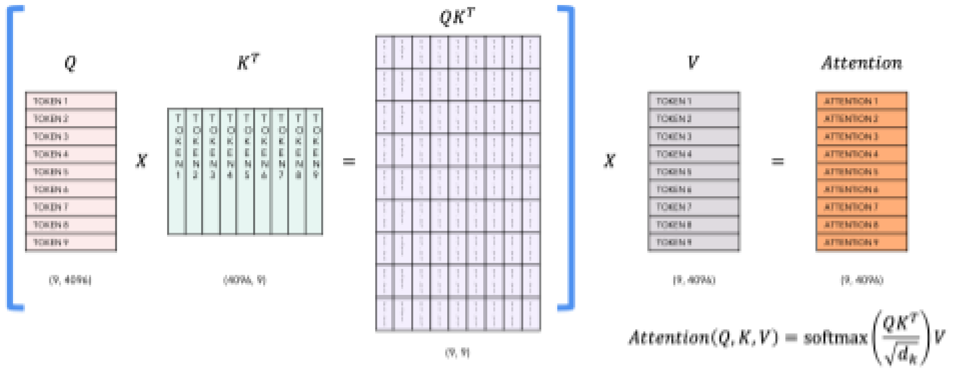
2. Background

In this section, we first describe the workload of autoregressive inference of a decoder-only transformer, then give an overview of how one can use MLC LLM and TVM to optimize such a workload.

2.1. Workload: Autoregressive Inference

Recently, we have seen various applications of large language models including question-answering, chatbot, and many more. The majority of such applications leverage the inference of decoder-only transformers, specifically generating output in an autoregressive fashion. To simplify the concepts, we mainly focus on the computation in the self-attention calculation, and simply think of them as various matrix-matrix and matrix-vector calculations, without much consideration to the general transformer's architecture.

We can divide the workload into two phases: prefill and decode. In the prefill phase, the model takes in the user's input (multiple tokens; think of each as an embedding of a single word), internalizes the information, and outputs the next token. Looking at Figure 1a from [4], where we have 9 tokens, making it two matrix-matrix multiplications. On the other hand, in the decode phase, the model generates the next single token based on its previous single token and repeats until a certain condition is met (hence "autoregressive"). Consequently, looking at Figure 1b, the decode phase only has one single token as an input. Thus, the computation becomes two matrix-vector multiplications. In summary, our workload is simply a sequence of matrix-matrix multiplication and matrix-vector multiplication, among other operators that we will ignore for our purpose. The takeaway is that such computation is highly parallelizable and can thus leverage the operator-level optimization in TVM to better utilize the parallel architecture of GPUs.



Q
K^T
QK^T
V
$Attention$
(9, 4096)
(4096, 9)
(9, 9)
(9, 4096)
(9, 4096)
$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

Fig 1a: Workload in Prefill Phase

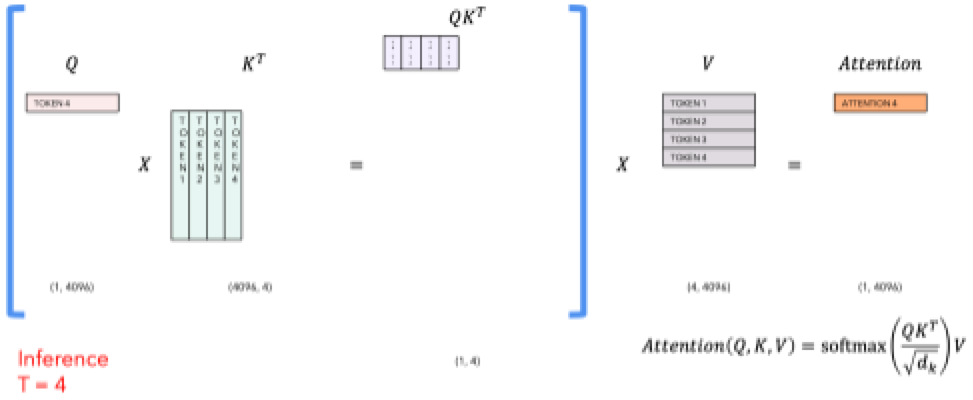


Fig 1b: Workload in Decode Phase

2.2. MLC LLM and TVM

With the workload of autoregressive inference in transformer models covered, we briefly go over the frameworks MLC LLM and TVM that help us accelerate such workload.

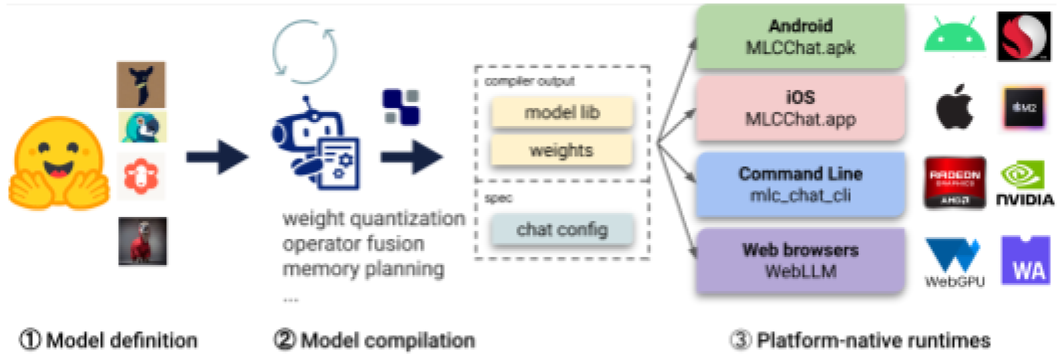


Fig 2: Workflow of MLC LLM

We can understand MLC LLM as an end-to-end framework that leverages TVM. To accelerate a model, GPT-2 in this case, we first need to define our model in a format recognizable by TVM (step 1 in Figure 2 from [3]). Then, taking this model definition as an input, TVM uses a default scheduler to perform various optimizations (step 2 in Figure 2), among which is the operator-level optimization we will focus on (specifically the parallelism aspects of it). Finally, in step 3, we can deploy the optimized model on various runtimes.

Therefore, to leverage this workflow to optimize the inference of GPT-2, we first need to implement the model definition, which is covered in Section 3. Then, we analyze three operator-level optimizations in TVM in Section 4 and pick a specific optimization (GEMV) to tune and analyze in detail in Section 5.

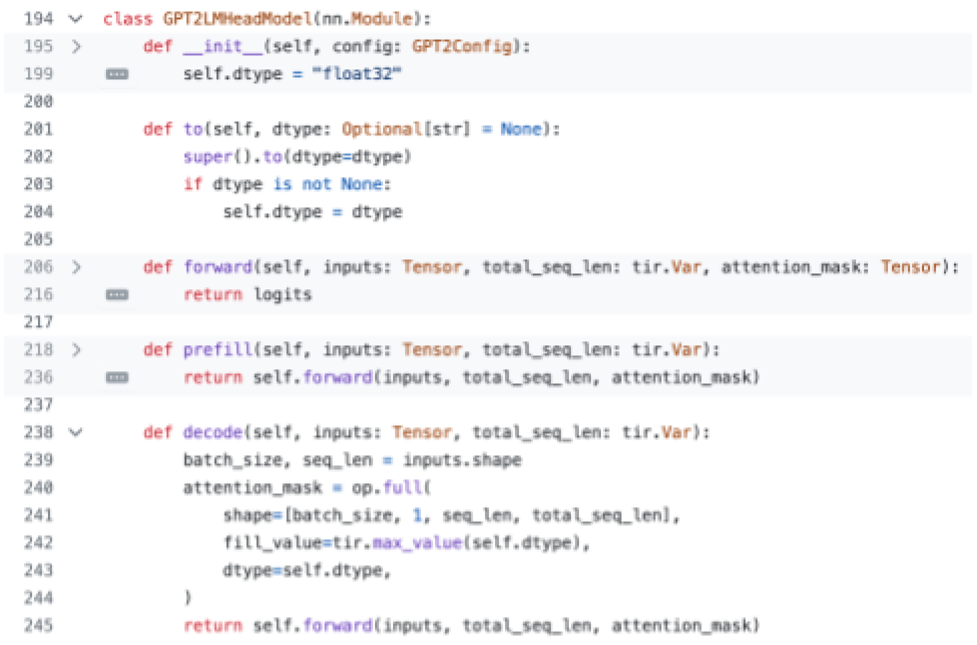
3. Approach

3.1. Implementing GPT-2 in MLC LLM

To optimize GPT-2 inference workload, we implement the model in MLC LLM in this GitHub pull request: https://github.com/mlc-ai/mlc-llm/pull/1314. The main implementation needed is under the folder gpt2. While most other files are needed to conform to the workflow of MLC LLM (e.g. loading the huggingface weights, quantizing the weights), the file that resembles the workload we care about for our purpose is the gpt2_model.py. The entire architecture and inference logic of GPT-2 is embodied in the class `GPT2LMHeadModel` as shown in Figure 3. This subclasses a TVM class called `nn.Module`, which we can understand as a `torch.nn.Module` but conformed to the domain-specific language recognized by TVM.

One caveat for implementing workloads in TVM is the usage of symbolic variables. In a typical deep learning framework like PyTorch [5], the inputs are all concrete values. For instance, the total sequence length argument (4 in the case of Figure 1b) in Pytorch would simply be an integer. However, to compile and optimize the workload, such a variable needs to be symbolic in TVM. As a result, operators like `op.full()` and `max_value()` all need to be implemented in TVM's operators that can take in symbolic inputs.

As an example, we can look at the `decode()` function we briefly covered in Section 2. We simply infer the batch size and sequence length from the input (both are 1 in the case of Figure 1b), create a causal mask, and call GPT-2's forward method that ultimately returns the logits.



```python
class GPT2LMHeadModel(nn.Module):
    def __init__(self, config: GPT2Config):
        self.dtype = "float32"

    def to(self, dtype: Optional[str] = None):
        super().to(dtype=dtype)
        if dtype is not None:
            self.dtype = dtype

    def forward(self, inputs: Tensor, total_seq_len: tir.Var, attention_mask: Tensor):
        return logits

    def prefill(self, inputs: Tensor, total_seq_len: tir.Var):
        return self.forward(inputs, total_seq_len, attention_mask)

    def decode(self, inputs: Tensor, total_seq_len: tir.Var):
        batch_size, seq_len = inputs.shape
        attention_mask = op.full(
            shape=[batch_size, 1, seq_len, total_seq_len],
            fill_value=tir.max_value(self.dtype),
            dtype=self.dtype,
        )
        return self.forward(inputs, total_seq_len, attention_mask)
```

Fig. 3: GPT-2 Implementation in MLC LLM

3.2. Optimizing GPT-2 Inference with TVM

After defining the workload in MLC LLM, we need to optimize it with TVM. The workflow is defined on a high level in Figure 4. Given the `tvm.nn.Module` we implemented in Section 3, which we can view as an intermediate representation, we first go through various high-level intermediate representation optimizations. This may include fusing transpose and matrix multiplication, workload-specific optimization like fusing a dequantization and transpose when the workload leverages a quantized model, and many more.

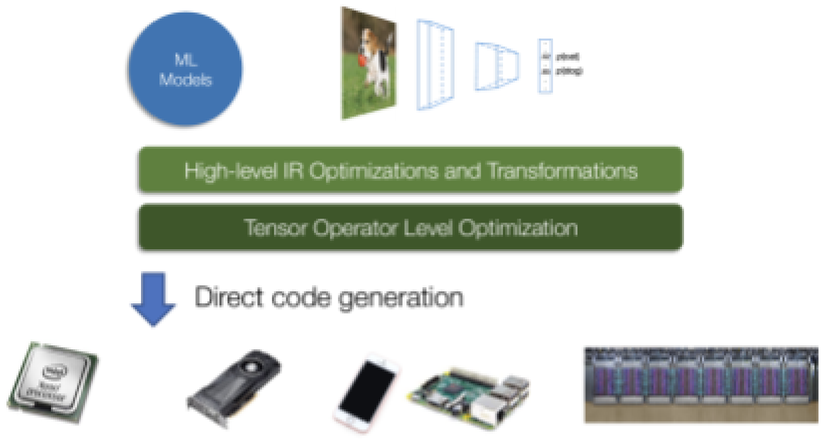

ML Models
High-level IR Optimizations and Transformations
Tensor Operator Level Optimization
Direct code generation

Fig. 4: Optimization Workflow

After the high-level optimizations, we then go through various passes at the operator level, which optimizes how the intermediate representation can leverage parallel computing platforms. Finally, we transform the optimized intermediate representation to generate the corresponding code, such that the model can be deployed on different platforms.

We note that the entire workflow is already implemented in MLC LLM as a pre-set schedule. We compare the performance of an RTX 4090 and an M2 Ultra. Specifically, we prefill 322 tokens (the entire Gettysburg Address) and let the model generate 512 tokens autoregressively.

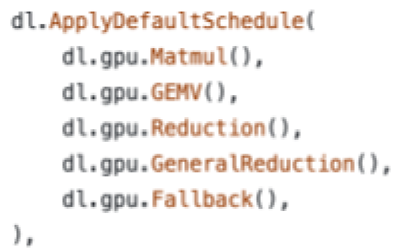
	Prefill (tokens/sec)	Decode (tokens/sec)
RTX 4090 - TVM	17809.243	338.614
M2 Ultra - TVM	3989.572	117.239

Table 1: Comparison of TVM-optimized GPT-2 inference between platforms.

4. Analysis of Operator-Level Optimization in TVM

In the previous section, we implement the workload in the domain-specific language and optimize it with a pre-defined schedule. In this section, we analyze each operator-level optimization in the pre-defined schedule in TVM.

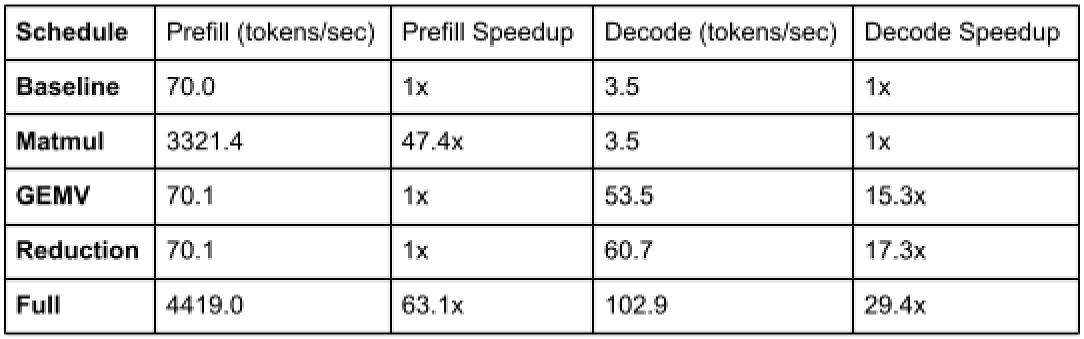
Recall in Section 3.2, one main part of the optimization workflow is the operator-level optimization. Concretely, we go through roughly four steps as shown in Figure 5. `Matmul()` performs tiling and loop fusion for matrix-matrix multiplications, `GEMV()` achieves similar optimizations but for matrix-vector multiplication, both `Reduction()` and `GeneralReduction()` optimize reduce-focused operators (e.g. `softmax()`, `LayerNorm()`, and `RMSNorm()`), and `Fallback()` applies thread block mapping to the spatial loops (as opposed to reduction loops).



```
dl.ApplyDefaultSchedule(
    dl.gpu.Matmul(),
    dl.gpu.GEMV(),
    dl.gpu.Reduction(),
    dl.gpu.GeneralReduction(),
    dl.gpu.Fallback(),
),
```

Fig. 5: Operator-Level Optimizations in TVM

After understanding what each optimization does, we analyze how each affects the performance of GPT-2 inference to identify a potential bottleneck that we should further tune. In Table 2, we compare how applying a single rule can affect the end-to-end performance of GPT-2 inference. “Baseline” denotes without using any optimization, and “full” denotes using all optimizations. Note that we consider `Reduction()` and `GeneralReduction()` as a single optimization, and we include `Fallback()` in all schedules as it is required for correctness. The experiment is run on a single RTX 4090, performing the same workflow as in Table 1 (here we use gpt2-xl instead of gpt2-medium and decode 64 tokens instead of 512; hence the different results from Table 1).



Schedule	Prefill (tokens/sec)	Prefill Speedup	Decode (tokens/sec)	Decode Speedup
Baseline	70.0	1x	3.5	1x
Matmul	3321.4	47.4x	3.5	1x
GEMV	70.1	1x	53.5	15.3x
Reduction	70.1	1x	60.7	17.3x
Full	4419.0	63.1x	102.9	29.4x

Table 2: Performance gain obtained from each operator-level optimization

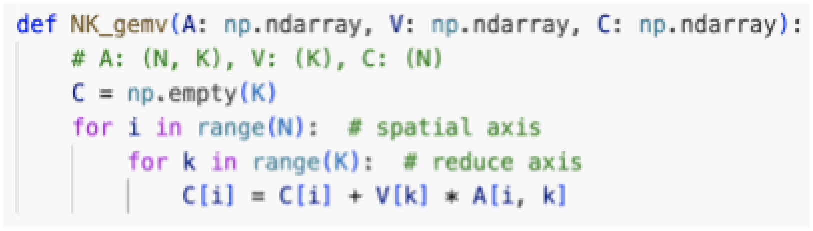
Across the three optimizations, we observe that only `Matmul()` can provide speedup to the prefill phase. This makes sense because, as brought up in Section 2 when describing the workflow, the prefill phase mainly consists of matrix multiplications due to how the Q matrix has multiple tokens. However, interestingly, with all optimizations applied, we gain more speedup for prefill despite other optimizations do not have an effect on prefill on their own. We note that `Reduction()` only offers speedup to the decode phase. This is also expected because, whenever we sample the next token in the decode phase, we need to run a `softmax()` to make the logits a distribution. Finally, we observe that `GEMV()` can bring speedup to decode, since, as we mentioned in Section 2 as well, the decode phase comprises many matrix-vector multiplications.

5. Further Optimization: Tuning GEMV

On an application level, we are particularly interested in cases where developers use GPT-2 as a generative tool. For instance, we recently encountered a project that leverages a GPT-2-based model to generate music autoregressively in a streaming fashion [6]. In this case, the decode phase is particularly important. Partly motivated by this, and partly seeing how the decode speedup is not as significant as that of the prefill speedup, we decide to further optimize `GEMV()`.

5.1. Workload of GEMV in TVM

GEMV in TVM focuses on the workload described in the code snippet in Figure 6. That is, we are essentially computing the product of an N-by-K matrix `A` and a K-long vector `V`, resulting in an N-long vector. We denote the axis across N to be the spatial axis `i`, and the axis across K to be the reduction axis `k`. This is illustrated in Figure 7.



```
def NK_gemv(A: np.ndarray, V: np.ndarray, C: np.ndarray):
    # A: (N, K), V: (K), C: (N)
    C = np.empty(K)
    for i in range(N):  # spatial axis
        for k in range(K):  # reduce axis
            C[i] = C[i] + V[k] * A[i, k]
```

Fig. 6: The workload of GEMV in TVM.

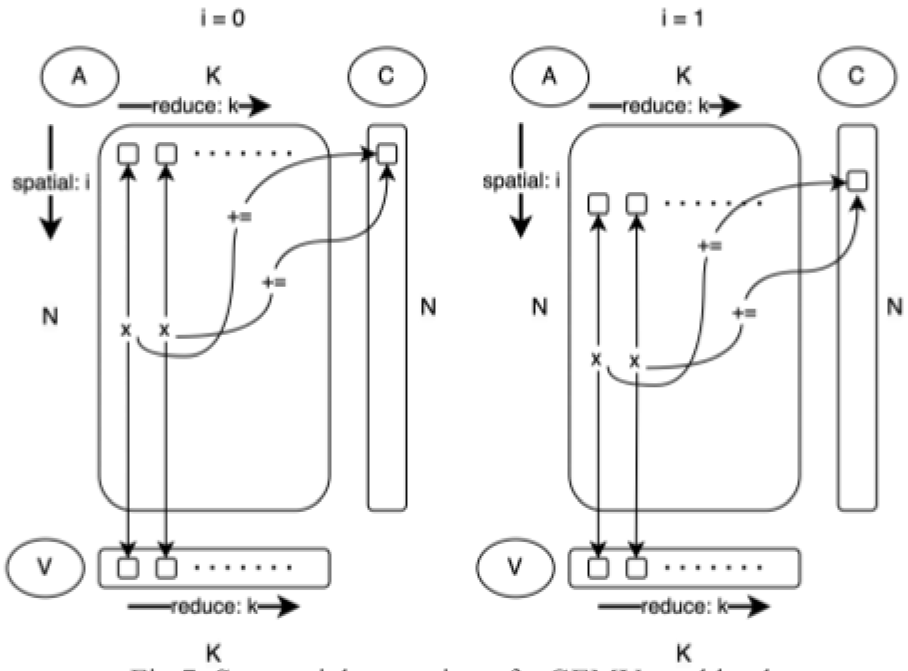


Fig.7: Sequential execution of a GEMV workload.

We note that Figure 7 describes what a sequential execution of the GEMV workload would look like. Specifically, we visualize when the spatial loop is 0 (on the left) and when the spatial loop is 1 (on the right). We immediately realize that there is massive parallelism that can be exploited. Specifically, the computation of each iteration across the spatial axis is independent of each other, since they write to a different location in `C`.

5.2. Parallelizing GEMV in TVM

To parallelize GEMV, TVM does the following steps:

1. Break the N rows into groups of `TS` contiguous rows; each threadBlock takes a group. (Note each threadBlock's work is independent from each other since we split across the spatial axis)
2. Each block uses a thread layout of `TS` by `TR`
   (i.e. each block has `TS*TR` threads; as a result, each thread has `TS*K/(TS*TR) = K/TR` amount of work)
3. Each thread block works on a tile of shape `TILE_S` by `TILE_R` at a time
   (Again, `TILE_S` is split across the spatial axis, and `TILE_R` across the reduction)

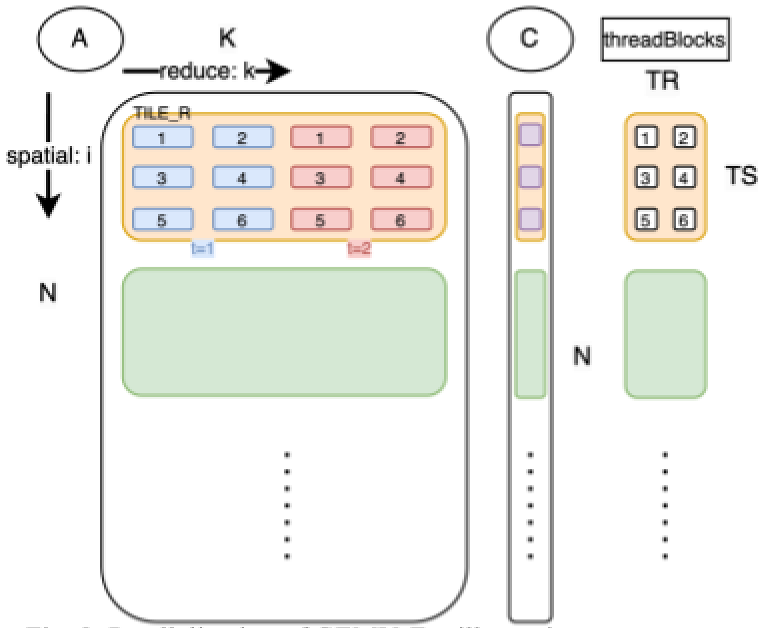


Fig. 8: Parallelization of GEMV. For illustration purpose, we make K=8, TILE_R=2, TILE_S=1 TR=2, TS=3. As a result, there are two iterations in total, as demonstrated by the color red and blue.

We visualize the workload scheduling in Figure 8. This gives us the search space we will use to tune the GEMV optimization in TVM, namely the values of `TS`, `TR`, `TILE_S`, and `TILE_R`.

5.3. Tuning GEMV in TVM and Results

To reduce the search space, we arbitrarily make `TILE_R` to be multiples of 8 and consider two possible schedules:

- **Schedule 1: Load vectors over N and compute vectors over K**
  In this case, we set `TILE_R` to be 8, and search for different values of `TILE_S`.
- **Schedule 2: Load vectors over K and compute vectors over N**
  In this case, we set `TILE_S` to be 1 and search for different values of `TILE_R`.

We base our tuning on the provided repo in [7] but they differ in several aspects. We do not inference a quantized model, so our GEMV workload is more simplified. We also use input shapes specific to the GPT-2 medium. Besides, we tune for RTX 4090 (CUDA backend) and M2 Ultra (metal backend) separately, since each hardware's different architecture will give us different results.

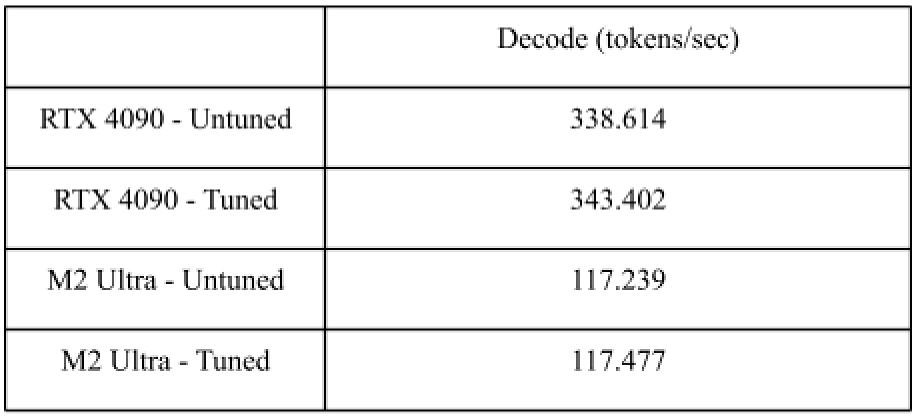
The resulting best configuration for RTX 4090 is `TILE_S=1`, `TILE_R=8`, `TR=16`, and `TS=8`.
The resulting best configuration for M2 Ultra is `TILE_S=1`, `TILE_R=16`, `TR=32`, and `TS=2`.

Beyond finding the best config, for each platform and schedule, we visualize:

1. Given a pair of `TS` and `TR` (the best ones), how do changing `TILE_S` and `TILE_R` affect timing in Figure 9.
2. Given a pair of `TILE_S` and `TILE_R` (the best ones), how do changing `TS` and `TR` affect timing in Figure 10.

The darker the color, the faster the configuration is. However, despite being easy to tell which configuration is the fastest for a platform, it is hard to find any pattern. Therefore, our takeaways are: such hyperparameter tuning needs to be done for each platform separately as they exhibit different patterns; tuning such parameters may need more than just domain knowledge. Here we are only leveraging a brute-force grid search, but more advanced methods like leveraging a cost model are also popular research areas.

Finally in Table 3, we benchmark the performance of our tuned model against our previously untuned model compiled with the default GEMV setting. Even though we indeed observe performance gain, it is relatively non-significant. For future work, we may look into leveraging more advanced techniques than grid search (e.g. learning-based approach with a cost model), increase our search space, and tune for the reduction optimization as well.



	Decode (tokens/sec)
RTX 4090 - Untuned	338.614
RTX 4090 - Tuned	343.402
M2 Ultra - Untuned	117.239
M2 Ultra - Tuned	117.477

Table 3: Comparison of GPT-2 inference with (un)tuned GEMV.

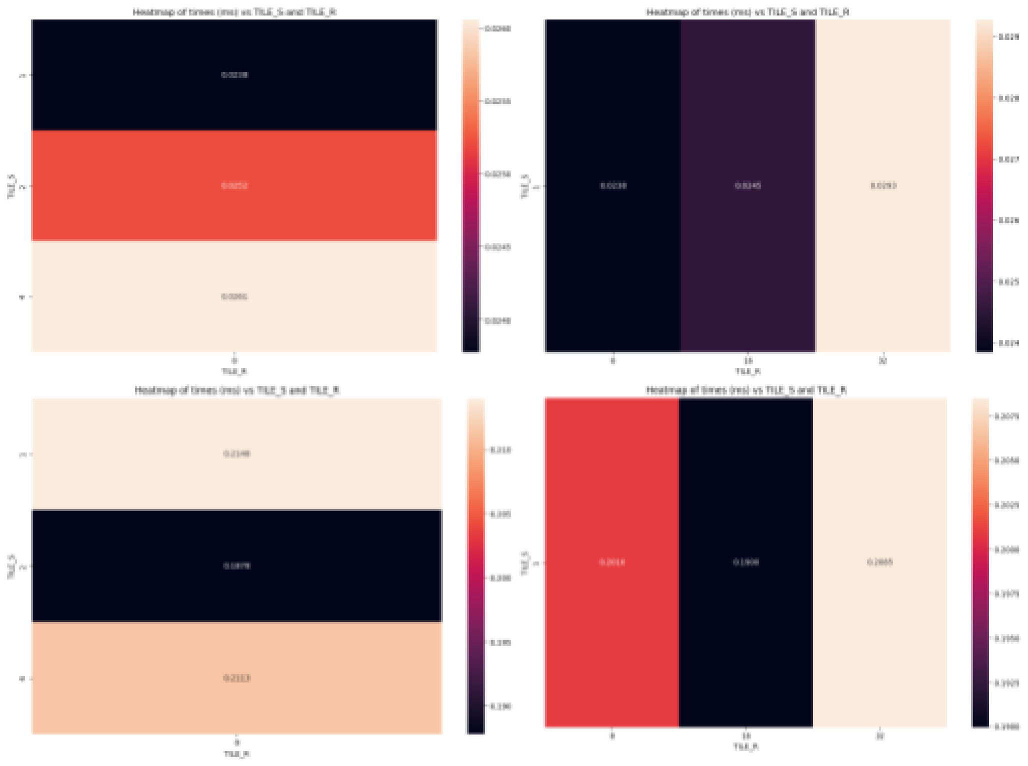


Fig. 9: Comparison of how TILE_S and TILE_R affect GEMV given a set pair of TS and TR. Top figures are from RTX 4090; bottom figures are from M2 Ultra. Left figures are from schedule 1; right figures are from schedule 2.

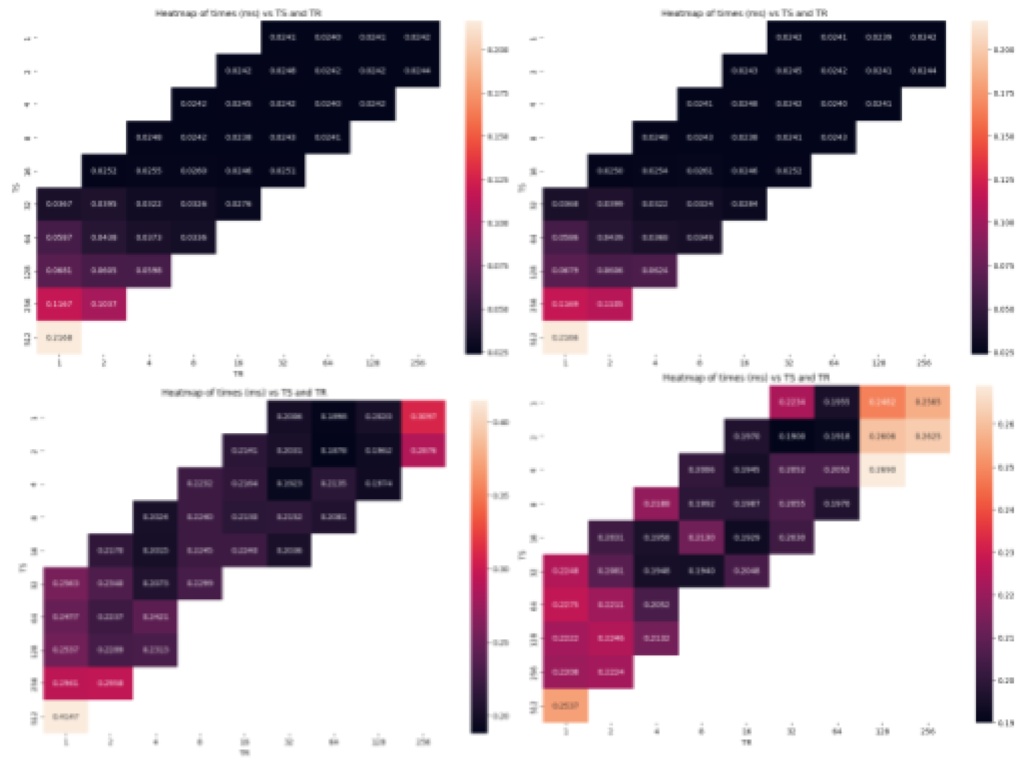


Fig. 10: Comparison of how TS and TR affect GEMV given a set pair of TILE_S and TILE_R. Top figures are from RTX 4090; bottom figures are from M2 Ultra. Left figures are from schedule 1; right figures are from schedule 2.

6. References

[1] 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), arXiv:1802.04799 [cs.LG].
[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al., Language models are unsupervised multitask learners, OpenAI blog 1, 9 (2019)
[3] MLC Team, MLC-LLM (2023) https://github.com/mlc-ai/mlc-llm
[4] U Jamil, LLaMA Explained (2023) https://github.com/hkproj/pytorch-llama-notes
[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. K¨opf, 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), arXiv:1912.01703 [cs.LG].
[6] J. Thickstun, D. Hall, C. Donahue, and P. Liang, Anticipatory music transformer (2023), arXiv:2306.08620 [cs.SD].
[7] MLC Team, Dlight-Bench (2023) https://github.com/mlc-ai/dlight-bench

7. List of Work By Each Student, And Distribution of Total Credit

Rick Zhou: 50%

- Implemented GPT-2 in MLC-LLM
- Verified correctness GPT-2 by comparing against Huggingface implementation
- Learning DLight (the pre-set schedule) and GEMV in TVM
- Tuning GEMV on RTX 4090 and M2 Ultra
- Report writing

Charlie Ruan: 50%

- Implemented GPT-2 in MLC-LLM
- Profiling performance under various settings
- Learning DLight (the pre-set schedule) and GEMV in TVM
- Poster making
- Report writing