Triton-TVM: A Transpiler from Triton Language to TVM TensorIR

Yongqi Zhuo
Adviser: Mingyu Gao

1 Background

This project aims to develop a transpiler that can convert Triton code to TensorIR code.

1.1 Triton

Triton [8], developed by OpenAI, is a domain-specific language for expressing computation kernels in deep learning, featuring block-level abstractions. The DSL enables developers to concisely write efficient schedules for a kernel. In contrast with the standard CUDA programming model where a program is run by all the threads, in Triton a program manipulates a block of threads. (Scalar Program, Blocked Threads v.s. Blocked Program, Scalar Threads [8].) The Triton compiler can then perform multiple optimizations and automatic tuning on the Triton code, and lower it to CUDA or HIP code for execution.

The workflow of the Triton compiler comprises the following lowering steps: Python DSL →Triton IR →TritonGPU IR →LLVM IR →target code, where Triton IR and TritonGPU IR are two MLIR [4] dialects. The DSL directly corresponds to Triton dialect, comprising high-level functional operations, while TritonGPU dialect includes low-level data movement operations across the memory hierarchy.

1.2 TVM TensorIR

TensorIR [3] is the central IR in TVM [2] Unity, which also serves as a frontend language in the TVM Python DSL TVMScript. Optimization and lowering passes, as well as automatic tuning, are performed on it. TVM, similar to Halide [6], puts emphasis on separating the algorithm from the schedule. Although TensorIR does not enforce such separation, it allows scheduling to be applied to the IR by transforming the IR in certain patterns, so this allows automatic schedulers to explore the search space (e.g., different tiling and loop reordering schemes). This contrasts with Triton where scheduling is directly expressed in the DSL, only template-based tuning on some parameters is performed, manually designed optimization passes are heavily relied on, and no further schedulings are applied.

1.3 Motivation

1.3.1 Increase TVM Adoption. A transpiler from Triton to TensorIR can help to increase the adoption of TVM by allowing Triton users to know and learn TVM, which is a more general-purpose tensor compiler.

1.3.2 Enable More Platforms. TVM supports more platforms than Triton, such as CPU and OpenCL. By transpiling Triton to TensorIR, Triton users can leverage the TVM ecosystem and target more platforms.

1.3.3 Enable Automatic Benchmarking. With such a transpiler, Triton users can easily compare the performance of Triton and TVM on the same hardware, which can help them to decide which one to use, and help them migrate to TVM if TVM is faster.

2 Related Work

2.1 `linalg` Dialect and IREE

`linalg` [4] is an MLIR dialect capable of expressing many linear algebra operations. It is used as a central IR for many optimizations in some end-to-end machine learning frameworks such as IREE [7]. These machine learning frameworks take neural networks as input and iteratively lower the IR to the target hardware. In contrast, Triton takes block-level computation specification as input and instead of making use of existing IRs such as `linalg` or TVM TensorIR, Triton focuses on optimizing for NVIDIA GPUs so the Triton IRs are catered to this specific use case.

In general, Triton aims to enable developers to write efficient kernels, while machine learning frameworks aim to automatically optimize the computation, and Triton is not designed to become a machine learning framework. This project aims to transpile Triton into TensorIR, enabling the engineering efforts, such as the highly convoluted fused attention kernel, devoted to Triton to be leveraged by the TVM community.

2.2 Triton-Shared and Triton-Linalg

Treating Triton as a front-end language and converting it to another IR is not a new idea. Triton-Shared [5] and Triton-Linalg [1] aim to convert Triton to the `linalg` dialect. They extract the semantics of Triton programs and convert them to `linalg` ops, thus enabling existing MLIR infrastructure to be leveraged, and more hardware backends such as DSA to be supported, while benefiting from the efficient hand-written kernels that are impossible to be automatically discovered by machine learning frameworks. However, in these projects, certain aspects are not considered.

Triton DSL models block-level computation but the full semantics of the computation are not captured without considering the grid in which the blocks reside. In Triton, a separate piece of kernel launching code is required, which is equivalent to CUDA grids. However, the above two projects only convert the kernel code. Without the grid information, the semantics of the operator being implemented cannot be recovered, also further optimizations such as inter-block scheduling that involve the grid cannot be performed. This project aims to address this issue by taking the Triton kernel as well as the kernel launching code as input.

Triton kernels are already optimized specifically for blocked programs, but `linalg` is a rather high-level IR designed for automatic polyhedral compilation, so the conversion to `linalg` is stepping back through the optimization pipeline. For example, Triton kernels tend to use for-loops to factor out a reduction (e.g., `for i in range(N): sum += . . .`), but such patterns can be readily done by MLIR optimization passes on `linalg`, and the canonical form of `linalg` where no such control flow exists is more suitable for further optimizations. This project aims to convert Triton to TensorIR, which is a lower-level IR than `linalg` and is more suitable for reusing the manual optimizations already done in Triton.

The above two projects take Triton dialect instead of TritonGPU dialect as input, losing the chance to leverage memory-related optimizations in the Triton compiler. In the course of conversion from Triton dialect to TritonGPU dialect, the Triton compiler analyze data manipulation and assign them to different memory spaces (i.e., register file, shared memory and global memory), also applying multiple data layout transformations. This project aims to also take TritonGPU dialect as input, leveraging the memory-related optimizations in the Triton compiler.

3 Contributions

This work makes the following contributions:

TVM dialect: An MLIR dialect for TensorIR is developed, to serve as the bridge between the MLIR world and TVMScript (the Python DSL for TVM). The TVM dialect corresponds one-to-one with the TensorIR constructs, such as loop nests and computation blocks. In this project, this makes conversion of code easier. Moreover, the dialect can be further extended to other projects, and enables TVM passes to be written as MLIR passes. This opens up the possibility of using MLIR infrastructure to optimize TVM programs.

Targeting TVM TensorIR: A transpiler targeting TVM TensorIR is developed. While previous work target `linalg` dialect, this project targets TensorIR, which is a lower-level IR and is more suitable for reusing the manual optimizations already done in Triton. Also, kernel-launching code is captured by the transpiler, and this enables full semantics of the tensor program to be recovered, instead of being only able to launch the kernels with the user-provided kernel launcher with threadblock semantics.

TritonGPU dialect as input: The transpiler takes TritonGPU dialect as input. Instead of discarding all the scheduling information in Triton, Triton-TVM leverages the TritonGPU IR-specific information such as loop tiling to create a default schedule in TensorIR, which can benefit further optimizations, and can provide a baseline for the automatic schedulers in TVM.

4 System Overview

The workflow is as follows:

Python interface: The user sets Triton-TVM as a backend in the Triton compiler, and the compilation will be delegated to Triton-TVM, generating TensorIR code in TVMScript. Then the script will be loaded by the runtime and later computation will be dispatched to the TVM runtime.

Conversion to TVM dialect: Multiple MLIR passes are written to convert from Triton and TritonGPU dialect to TVM dialect. The passes include: rewriting tensor operations with scalar operations, adding loops for grid dimensions, eliminating all pointer arithmetic, and creating loop nests for TensorIR. The MLIR infrastructure enables the passes to be simplified, because generic canonicalization passes are already implemented and available for use.

TensorIR code generation: Program in TVM dialect is traversed and corresponding TVMScript constructs are emitted.

5 Design

5.1 Python Interface

Triton-TVM provides a user-friendly Python interface by integrating with the Triton compiler as a plugin. This is similar to the prior project Triton-Shared, as well as other internal components such as the CUDA backend and the AMD backend. Installing as a plugin enables the workflow of the user to remain unchanged, and only 2 lines of code are required to be added. See Listing 1.

Listing 1. Code modification in order to use Triton-TVM

```
from triton.backends.triton_tvm.driver \
    import TVMDriver
triton.runtime.driver.set_active(TVMDriver())
```

After the above modification, the Triton compiler will automatically invoke the Triton-TVM transpiler to process the TritonGPU IR code, and TensorIR in TVMScript will be generated. When the JIT function is called, the TVMScript module will be loaded by the runtime, and the computation will be dispatched to the TVM runtime. This provides the user with a seamless experience.

5.1.1 Caveat: Static Dimensions.

One difficulty is that Triton enforces the SPMD abstraction, and there is no way to access the grid size in the IR. The grid size is passed to the kernel launcher per invocation, and is not accessible to the compiler. Specifically, the kernel launching code is fully user-customizable, dependent on concrete input sizes, and is not visible to the Triton compiler, making it impossible to obtain grid size programatically. See Listing 2 for an example.

Listing 2. Example kernel launching code

```
z = torch.empty_like(x) # output
n = z.numel()
grid = lambda meta: \
    (triton.cdiv(n, meta['BLOCK_SIZE']), )
add_kernel[grid](x, y, z, n, BLOCK_SIZE=1024)
```

To handle this, we patched the JIT function interface to intercept the grid size upon first invocation of the kernel, and then pass it to the transpiler. This is a workaround and is not ideal, because in this way we can no longer support dynamic shapes, but it is the best we can do given the current Triton design.

5.2 TVM Dialect

Dialect conversion is a very important concept in MLIR. The MLIR infrastructure provides a convenient way to define operations, and operations are grouped logically into dialects. Different dialects can model different layers of abstractions, and dialect conversions enable the level of abstraction to be changed. The MLIR infrastructure provides a dialect conversion framework that enables conversion passes to be easier to write than plain graph rewrites. In this project, we define a TVM dialect so that we can make better use of the conversion infrastructure.

TVMScript is a Python DSL employed by TVM Unity, where the various IRs are given textual forms. Although multiple IRs are available, we mainly target TensorIR. Compared with the functional graph-level IR, Relax, which is also available in TVMScript, TensorIR explicitly manages buffers and loop nests. In TensorIR, the input values are buffers and the output values are out parameters. Buffers can be explicitly allocated. Tilings are explicitly expressed as loop nests. All computation are wrapped in blocks, where memory read and write locations are required to be declared, and the computation itself is rather functional, using expressions. See Listing 3 for an example.

Listing 3. Example of TVMScript code

```
@T.prim_func
def main(
    a: T.Buffer((8,), "float32"),
    b: T.Buffer((8,), "float32")):
    for i in range(8):
        with T.block("B"):
            vi = T.axis.spatial(8, i)
            B[vi] = A[vi] + 1.0
```

To model these constructs, several MLIR operations are defined in the TVM dialect. See Table 1.

As can be seen in Table 1, the TVM dialect ops are designed to closely resemble the TensorIR constructs. Moreover, some existing dialects in MLIR are reused, including `arith` for arithmetic operations such as addition, `memref` for buffer description, and `scf` for-loop constructs. This saves the efforts of redefining the operations, and allows reusing many of the optimization and simplification passes already implemented for these dialects.

5.3 Conversion of Triton and TritonGPU Ops

Conversion in MLIR involves *legalizing operations*, that is, converting each operation that is not legal in the target. Now we have to find a conversion scheme for each Triton operation.

See Listing 4 for a Triton program example. The program computes the softmax function. The abstraction layer of Triton is somewhat hybrid, in that it uses tensors for computation, which actually reside in shared memory in GPUs, but uses low-level pointer arithmetic to read from and write to global memory. It also supports low-level control flow such as loops. The conversion scheme for each kind of operations is discussed below.

Listing 4. Softmax in Triton

```
# Load
row_idx = tl.program_id(0)
col_offsets = tl.arange(0, BLOCK_SIZE)
input_ptrs = row_start_ptr + col_offsets
row = tl.load(
    input_ptrs,
    mask=col_offsets < n_cols,
    other=-float('inf'))
# Compute
row_minus_max = row - tl.max(row, axis=0)
numerator = tl.exp(row_minus_max)
denominator = tl.sum(numerator, axis=0)
softmax_output = numerator / denominator
# Store
output_row_start_ptr = \
    output_ptr + row_idx * output_row_stride
output_ptrs = output_row_start_ptr + col_offsets
tl.store(
    output_ptrs,
    softmax_output,
    mask=col_offsets < n_cols)
```

5.3.1 Tensor Computation. This kind of operations involves broadcasting ops (`broadcast`, `expand_dim`), element-wise ops (`exp`, `+`), and ops that involve reductions (`max`, `sum`,

TensorIR	TVM Dialect	Notes
`T.var("float32")`	`tvm.var : f32`	
`T.match_buffer`	`tvm.match_buffer`	Match shape with variables
`T.alloc_buffer`	`tvm.alloc_buffer : memref`	Allocate in specified memory space
`for k in range(n): ...`	`scf.for %k = %c0 to %n { ... } { tvm.for_kind = serial }`	Available scheduling directives are: `serial, parallel, vectorized, unroll`
`with T.block(): ...`	`tvm.block { ... }`	Specify computation
`T.where`	`tvm.where`	Predicate for constraining loop region
`T.axis.spatial(n, k)`	`tvm.axis spatial %n = %k`	Available axis bindings are: `spatial, reduce, scan, opaque`
`A[i, j]`	`tvm.ref %A[%i, %j]`	Index into a buffer (`memref`)
`T.if_then_else`	`tvm.if_then_else`	
`T.reads([A[i, j]])`	`tvm.read %ref`	Explicit declaration of read/write locations in memory
`T.writes([A[i, j]])`	`tvm.write %ref`	
`A[i, j] = ...`	`tvm.assign %ref = %expr`	Definition of computation
`with T.init(): ...`	`tvm.init { ... }`	
`T.min_value("float32")`	`tvm.min_value : f32`	FP infinity for arbitrary precision
`T.max_value("float32")`	`tvm.max_value : f32`	

Table 1. TVM dialect ops

`dot` (i.e., matmul)). Non-reduction ops can be directly converted into their counterparts in TVM dialect, with an arbitrary loop nest and a straightforward line of assignment. One key to note is that element-wise ops and broadcasting ops should be fused (i.e., not *materialized*) as much as possible to avoid excessive allocation of shared memory buffers and reduce memory footprint. Meanwhile, reduction-containing ops require materialization, allocating buffers when necessary. In brief, converting tensor computation is not difficult.

5.3.2 Loops.

for-loops in Triton are already in the form of `scf.for` in the MLIR code. The difficulty lies in the reductions performed by the for-loop, i.e., `for k: ptrs += stride` and `for k: acc += tensor`. In MLIR, which features static single assignment, this is expressed with the `iter_args` arguments that serve as the phi-nodes of these reduction variables. We can perform pattern matching on `iter_args` to find out the semantics of the loop, and replace them with their counterparts. For pointer arithmetic, we can rewrite to `ptrs = offset + k * stride`, and for tensor reductions, we bind `k` to a reduction axis `tvm.axis reduce %n = %k` and handle `tvm.block` correspondingly.

5.3.3 Load and Store.

The pointer arithmetic in Triton is extremely expressive, allowing many access patterns such as gather and scatter in addition to blocked load and stores. This enables sparse tensors to be loaded, which cannot be expressed with `memref`, `triton` and `linalg` dialects, and previous work such as Triton-Shared and Triton-Linalg just cannot handle them. However, in TVM, sparsity is supported, and data loading and storing can also be expressed with `block` constructs and for-loops. One problem is that pointers are used to index the linear address space, while TVM requires indices to index a tensor. The solution is not difficult. Since we have obtained the static shapes of all tensors as in Section 5.1.1, we can *delinearize* the pointer offset relative to the base address, and obtain the indices, for example, for tensor shape `[3, 4, 5]`, offset `42` gives indices `[42 / 20, (42 / 5) % 4, 42 % 5] = [2, 0, 2]`.

5.3.4 How to Build Loop Nests.

While tensors are pure in Triton dialect, TritonGPU dialect attaches an encoding attribute to every tensor type in the program, which they name *layout*. As the name suggests, the layout encodes how tensors are distributed across the GPU threads. In this work we mainly focus on the *blocked layout*, because other layouts are specifically designed for matrix multiplication, which involves too much NVIDIA GPU-specific details, and is better left for TVM to be optimized.

A blocked layout of a tensor of rank `r` comprises of 4 rank-`r` arrays, `sizePerThread`, `threadsPerWarp`, `warpsPerCTA`, and `order`, and the `i`-th dimension of a tensor corresponds to the `i`-th component of these arrays. `sizePerThread * threadsPerWarp * warpsPerCTA` is the shape of a *CTA tile* (defined as `shapePerCTATile`; CTA stands for cooperative thread array, which is just a block in CUDA terminology), which is a tile kept by a CTA in the tensor. The `i`-th dimension of the tile is divided into `warpsPerCTA[i]` pieces to be kept by different warps, and each piece is then divided into `threadsPerWarp[i]` pieces to be kept by different threads in the warp. Each thread then keeps a tile of shape `sizePerThread`. The `order` array specifies the order of dimensions to be accessed (the earlier a dimension is accessed, it is a more inner loop), where generally the most contiguous dimension is accessed first (dimension `order[0]`). Moreover, if a CTA tile is smaller than the tensor, then the tensor is divided into multiple CTA tiles. So following the layout, we can write the pseudo code for the loop nest as in Listing 5.

Listing 5. Pseudo code for-loop nest based on blocked layout

```
for tile_id[r-1] in range(shape[order[r-1]]
    / shapePerCTATile[order[r-1]]):
 ...
  for tile_id[0] in range(shape[order[0]]
      / shapePerCTATile[order[0]]):
   for warp_id[r-1] in range(
       warpsPerCTA[order[r-1]]):
    ...
     for warp_id[0] in range(
         warpsPerCTA[order[0]]):
      for thread_id[r-1] in range(
          threadsPerWarp[order[r-1]]):
       ...
        for thread_id[0] in range(
            threadsPerWarp[order[0]]):
         for i[r-1] in range(
             sizePerThread[order[r-1]]):
          ...
           for i[0] in range(
               sizePerThread[order[0]]):
            ... # computation
```

The layouts yielded by the Triton compiler are already optimized, and it is better to reuse them in the transpilation process. In contrast, Triton-Shared and Triton-Linalg do not take TritonGPU dialect as input, and they lose the chance to leverage the memory-related optimizations in the Triton compiler. In this project, we can use the layouts as the default schedule for the TensorIR program, and this can provide a baseline for the automatic schedulers in TVM.

5.4 Dialect Conversion Passes

A dialect conversion pass in MLIR defines a set of pattern matching (matching *illegal operations*) and rewriting rules (to *legal operations*), and the MLIR infrastructure will apply these rules in a systematic way. In this project the illegal operations are Triton ops (`tt`), TritonGPU ops (`ttgpu`) and `tensor` dialect ops (because we manipulate buffers instead of value-semantic tensors). Next we cover some important passes.

5.4.1 LowerToTensorIdiomsPass.

This pass converts ops that return tensors to `tensor.generate` op in a best-effort manner. `tensor.generate` in `tensor` dialect encompasses a basic block, which takes `r` indices as arguments and yields a scalar result, in the operation, and creates a rank-`r` tensor comprising of the scalars yielded by the basic block. For example, the line `softmax_output = numerator / denominator` in Listing 4 can be written as in Listing 6.

Listing 6. Example `tensor.generate` op

```
%softmax_output = tensor.generate {
  ^bb0(%i: index):
    %e = tensor.extract %numerator[%i]
    %div = arith.divf %e, %denominator
    tensor.yield %div
}
```

The transformation has 3 benefits.

Compact for-loops: A `tensor.generate` op can be seen as a compact loop nest, where the loop bounds are inferred from the tensor shape. This is more concise than the explicit for-loops. The basic block encompassed in the `tensor.generate` op performs only scalar operations, so is far easier to pattern-match and optimize, and makes later generation of loop nests easier.

Leveraging existing passes: MLIR has existing canonicalization and optimization passes for `tensor.generate` ops. For example, when a `tensor.extract` op extracts from a tensor produced by a `tensor.generate` op, the basic block in the `tensor.generate` op can be inlined to the call site. This effectively performs operator fusion (producer-consumer fusion) and eliminates the intermediate tensor, as mentioned in Section 5.3.1.

Canonicalization for `tt.addptr` ops: In the Triton compilation stack, pointer arithmetic is performed by `tt.addptr %ptr, %offset`, where the operands are almost always tensors, and the IR is full of `tt.splat` ops (which is just a broadcast of a scalar value). Although Triton can canonicalize `addptr(addptr(p, x), y)` into `addptr(p, add(x, y))`, it is observed that it failed to canonicalize `addptr(splat(addptr(p, x)), y)` into `addptr(splat(p), add(splat(x), y))`. This is a result of lacking proper canonicalization rules due to not using the official `tensor.splat` op (I guess the reason for this is that `tensor.splat` does not support non-integer-or-float values as operands, such as Triton pointers). Manually adding the canonicalization rules is just cumbersome. In this project, we rewrite all arithmetic into scalar arithmetic, and by rewriting `tt.splat` into `tensor.generate`, we can fully canonicalize the program. In this way, it becomes trivial to match the pattern of `addptr(base, offset)` and delinearize the offset into indices, as mentioned in Section 5.3.3.

5.4.2 RewriteSPMDToLoopsPass.

As mentioned in Section 5.1.1, the Triton program is a SPMD program and only describes computation in a CTA. In this pass, the static grid size is passed in. The program is then wrapped in the loop nest `for blockIdx[...] in range(gridDim[...])` and `tt.get_program_id` is converted into `blockIdx` in the loop nest. This transforms the SPMD program into a full tensor program, recovering the full semantics of the computation.

5.4.3 MaterializeTensorsToTVMBuffersPass. This pass replaces each `tensor.generate` op with loop nests. It generates the loop nests according to the layout of tensors, as described in Section 5.3.4, and copies the content in the basic blocks of `tensor.generate` ops to the center of the loop nests, and also replaces `tensor.extract` ops with `tvm.ref` ops. Notably, we employ *slice analysis* to correctly analyze the producer-consumer relationship of tensors and emit the `tvm.read` and `tvm.write` ops. After this pass, we get a fully legal TVM dialect program.

5.5 TensorIR Code Generation

The code generation pass traverses the IR and picks out the TensorIR constructs, and emits the TensorIR Python DSL. An expression formatter that automatically analyzes the precedence of Python operators was devised to eliminate extra paranthesis for pretty printing, which enhances the readabilty so that later inspection and manual optimization is easier. Also, to translate the operations in `arith` and `math` dialect, some template metaprogramming techniques were employed so that more operations can be supported with less code, improving maintainability and extensibility.

6 Evaluation

We evaluate Triton-TVM on a set of benchmarks, including vector addition and fused softmax. The benchmark was run on an NVIDIA RTX 3060 Laptop GPU (30 SMs), with CUDA 12.6. Precision is set to `float32`. For fused softmax, the number of rows is set to 1024. The triton programs are from the end-to-end test scripts in the Triton-Shared repository. The baselines are PyTorch native operators and the default Triton compiler with CUDA backend. To better illustrate the performance, we compare the number of DRAM accesses per second, which can be directly compared with the memory bandwidth of the GPU. In this case, the memory bandwidth is 336 GB/s.

Figure 1 shows the performance of vector addition. It can be seen in this simple setup, no major difference among PyTorch, Triton and Triton-TVM is observed, and the computation is mostly memory-bound, as the peak performance is around 316 GB/s. This indicates that Triton-TVM is capable of loyally translating the schedules related to GPU blocks and threads into TensorIR, preserving the performance of the original Triton program.

Figure 2 shows the performance of fused softmax. The difference among the three curves is mostly within 10%, and while PyTorch beats the other two Triton-based implementations in most cases, no major difference between Triton and Triton-TVM is observed. This proves that even if there are multiple buffer allocations, Triton-TVM can still replicate the performance of the original Triton compiler, preserving all the GPU threads scheduling information.

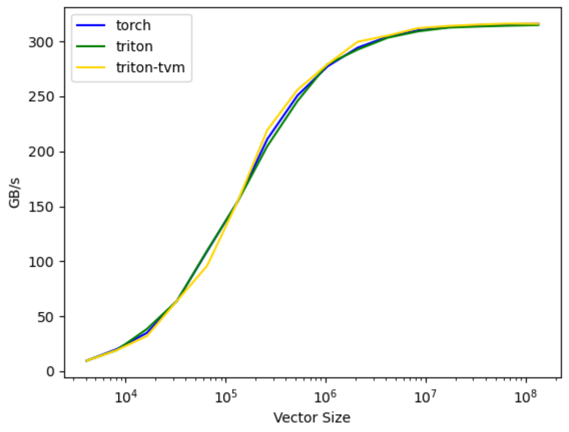

Figure 1. Performance of vector addition

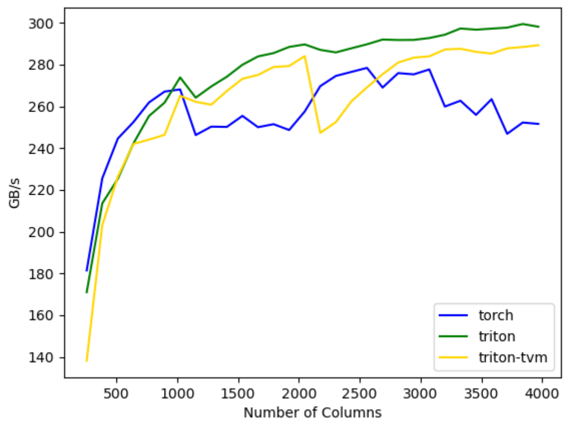

Figure 2. Performance of fused softmax

7 Discussion

The evaluation results show that Triton-TVM can faithfully transpile Triton programs to TensorIR, and the performance is on par with the original Triton compiler. This enables the Triton community to leverage the TVM ecosystem, to learn more about the cross-platform compilation stack. This also enables the TVM community to benefit from the highly optimized Triton kernels.

However, there are still some limitations. Optimizations that involve Tensor Cores require calling specific intrinsics of GPUs, and this requires significant engineering efforts, so has not been implemented in Triton-TVM. This is the reason why matrix multiplication is not included in the benchmarks.

8 Future Work

8.1 Support for Dynamic Shapes

TVM provides the ability to accept dynamically-shaped tensors as inputs, but for simplicity, we only support static shapes in this work, and all shape parameters must be marked `tl.constexpr` in the Triton program for Triton-TVM to function. In the future, dynamic shapes can be supported by using the `tvm.var` and `tvm.match_buffer` op in the TVM dialect.

8.2 Match Reduction Patterns

The design mentioned in section 5.3.2 has not been implemented yet. However this should not be difficult, as the MLIR infrastructure already provides such pattern matching functions.

8.3 More Layouts and Tensor Cores

The current implementation can only support FMA for matrix multiplications. To support Tensor Cores, swizzled shared memory layouts must be considered in addition to blocked layouts. This involves legalizing `ttgpu.convert_layout` ops, which can be challenging.

References

[1] Cambricon. triton-linalg. https://github.com/Cambricon/triton-linalg, May 2024.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[3] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[4] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[5] Microsoft. triton-shared. https://github.com/microsoft/triton-shared, September 2023.

[6] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[7] The IREE Authors. IREE. https://github.com/iree-org/iree, September 2019.

[8] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.