Optimizing for Dynamic Shape in TVM via Auto-Tensorization with Micro-Kernels

Wuwei Lin*
wuweil@andrew.cmu.edu

Yingjing Lu*
yingjinl@andrew.cmu.edu

1 Introduction

1.1 Problem setup

Optimizing through fine tuned micro-kernel has become an essential technique for basic Linear Algebra Subprogram(BLAS) computation. It fits into the TVM framework pretty well in that most of the TVM computation revolves around dense matrix multiplications. Micro-kernel is also a essential media to leverage domain specific instruction sets such as SSE and AVX2 from x86 CPU, which TVM is lacking native support. Therefore replacing the nested loops with external procedure call to micro-kernels is commonly used as an optimization in TVM, which is called the tensorization.

To achieve good performance by AOT compilation using TVM, the input shape need to be known at compile time. For fixed input shape, TVM can perform a series of optimizations such as loop tiling and utilize the micro-kernels via tensorization, which replaced the nested loop bodies with a function call to external procedures. But the requirement of fixed input shape at compile time is impractical because we often want to change the batch size or the shape in runtime. Allowing variables in the input shape relaxes this constraint. However, optimizations such as loop tiling may often result in excessive amount of conditional statements to enumerate all possibilities of the variable. Such conditional statements in the nested loop bodies make utilizing the external micro-kernels impossible because we require the nested loop bodies to exactly match the computation of the micro-kernels. The attempt of using IR passes to partially evaluate the expressions and eliminate the conditional statements also fail in that the tensorization pass requires the information from the original computation definition, which is lost during the IR transformations. This constraints make it impossible to utilize micro-kernels to optimize for dynamic input shapes.

1.2 Approach

For the micro-kernel part, we aim to find and tune some generic templates to be applied to generate apply micro-kernel during the computation of dense matrix multiplication or some other computation forms such as convolution which can be viewed as strided matrix multiplication.

To support tensorization under dynamic input shape, we use loop partitioning to eliminate conditional statements. After that, we use a auto-tensorization pass to detect the candidate loops for tensorization and check if the candidate match the definition of the provided micro-kernels. If we find a match, we replace the original loops with a call to the micro-kernels.

1.3 Related Work

TVM [Chen et al., 2018a] is a deep learning compiler that uses the intermediate representation to describe and optimize the tensor operations, and then generate efficient code for different platforms. With machine learning based tensor computation cost model, it efficiently explore the optimization

* Denotes equal contribution

Preprint. Under review.

search space without manual tuning [Chen et al., 2018b] and achieved impressive performance for many different operations and platforms. TVM requires users to implement the operations with its tensor expression DSL API and the schedule primitives. Although it lowers the requirement of the engineering efforts, it still requires much expert knowledge. Therefore, automatic scheduling TVM has gain increasing attention from researchers such as [Zheng et al., 2020]. This project explores the automatic schduling by using auto tensorization to optimize the TVM schedule with micro-kernels by automatically applying appropriate predefined micro-kernels.

Micro-kernel optimization has been a widely studied area that overlaps with many of the traditional memory hierarchy optimization techniques. The GOTOBLAS framework Van Zee and Van De Geijn [2015] is one of the comprehensive framework that combines the loop and memory optimization techniques with the most recent development of vectorized SSE instruction set that greatly improves computation such as matrix multiplication, regression, etc.

1.4 Contributions

We explored and propose a generic template to be applied for generating micro-kernel that can be applied for computing matrix multiplication tasks. It leverages some of the basic optimization techniques such as loop unrolling, blocking for cache, and support for SSE and AVX2 instruction set on x86 architecture.

We also show the empirical optimization ablation through general matrix computation benchmarks that illustrates the importance of each optimization components. ans how they can be optimally combined with each other.

We also extended the well formulated GOTOBLAS [Van Zee and Van De Geijn, 2015] framework to support the most recent AVX/AVX2 instruction set and empirically demonstrates that it greatly improves the performance.

We introduced the auto tensorization pass to TVM that automatically choose the applicable micro-kernels to replace the original loop bodies and improves the performance for dynamic input shapes.

2 Details Regarding Designs and Approaches

2.1 Micro-Kernel

For the design of the template we assume that the source data can fit into the memory of the compute unit (RAM for CPU and DRAM for GPU) which is a fair assumption considering most current DL model requires the compute unit to be able to hold the parameters and the inputs at the same time. As a result, we would not need to consider to overhead of fetching from disk which is significantly more expensive than any of our optimization area. We also assume that the source parameter and input is significantly larger than the cache or the register capacity, otherwise trivial. So the general optimization involves to use loop to break down larger parts and minimize the data traffic in the smaller micro-kernels. Our design paradigm follows the GOTOBLAS framework [Van Zee and Van De Geijn, 2015] that involves the following:

- Access continuous memory areas as much as possible
- Allow different parts of the same loop to be able to computed simultaneously
- Retrieve blocks at different loop level according to the specs of the cache hierarchy: L3-L2-L1-Register
- Eliminate any unnecessary compute and storage overheads such as index computation and store.
- Pack input and output to force them in contiguous addresses.
- Align the size of the block to the size of the vectorized instructions (SSE, AVX, AVX512) to maximize the use of vector registers.

From top level to button, the deeper we got, the more restrictive on the shape our template can be applied. The basic loop unrolling and blocking is suitable for arbitrary shape. Templates involving SSE and AVX register instructions requires the shape to multiples of the template's aligned size.

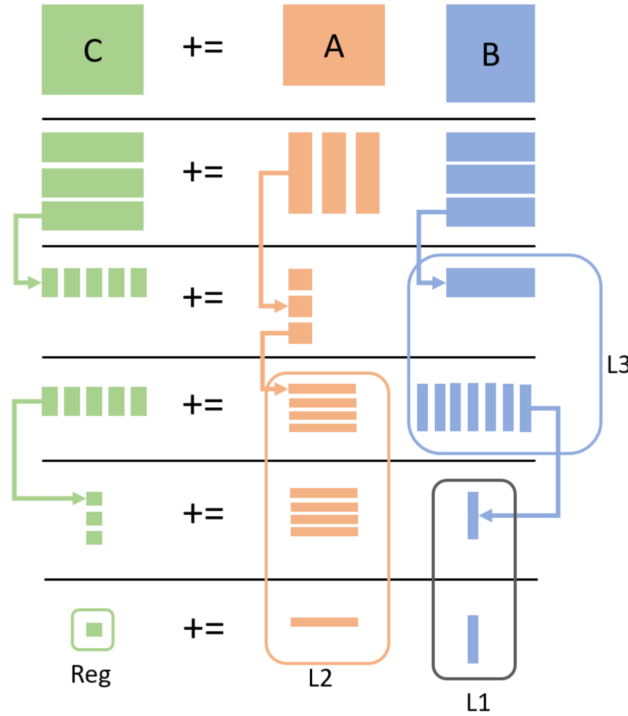

Figure 1: Matrix multiplication loop breakdown illustration for each level of loop and their utilization of the memory hierarchy

2.2 Support dynamic shape with loop partition and auto tensorization

In TVM, an implementation of an operation has two parts: compute and schedule. Similar to Halide[Ragan-Kelley et al., 2013], the compute uses a index formula to give the per-element definition, which only provides the semantics of the operation but does not involve the actual computation-related logics such as memory allocation and loop orders. The schedule is to apply a series of schedule primitives such as loop tiling, vectorization to lower the IR to actual machine dependent computation-related logics.

If the shapes of the input are known at compile time, TVM can deal with IR transformations very well. Its integer-set-based analyzer is able to compute the shapes of the result tensors of IR transformations and perform partial evaluation during the compile time to simplifies the expressions. However, when the shapes of the input are unknown, the optimizations during the schedule stage such as loop tiling introduces conditional statements. Although these statements are necessary to guarantee the access not crossing the boundaries, they impose extra difficulties for further optimizations such as tensorization.

To optimize for dynamic input shapes, we apply the loop partition pass first and then apply the auto tensorization pass. The whole pipeline is illustrated in Figure 2.

2.3 Loop Partition

Loop partition is a built-in pass of TVM. It uses the integer-set-based analyzer to analyze the predicate of the conditional statements. It partitions the loop into ranges depending on whether predicate always holds true or the predicate can not be evaluated at the compile time. In this way, if the predicate always holds true, it can be safely eliminated. If the predicate holds true under certain conditions depending on the outer loop, it can also possibly hoist the conditional statements to the outer loops. This make is possible for us to perform the tensorization optimization after we eliminated all the conditional statements in the inner loops.

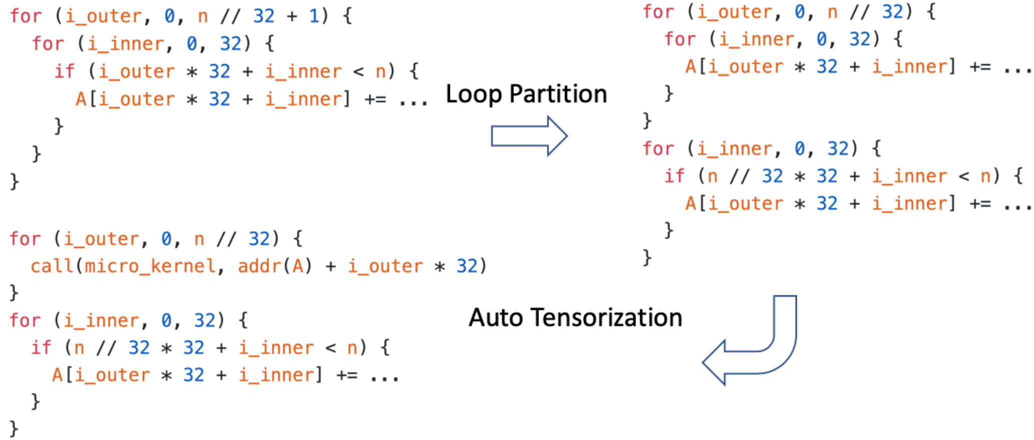

Figure 2: The optimization pipeline for dynamic shape input

2.4 Auto Tensorization

After the loop partitions, some of the conditional statements inside the loops are eliminated. Therefore, this provides possibility to utilize micro-kernels via tensorization. As mentioned before, the original tensorization pass in TVM is not applicable, therefore we implemented our own tensorization pass instead. Because now the tensorization happens after other IR passes such as loop partition, we cannot ask uses to specify the IR node to be tensorized. Therefore, it is required that the new tensorization pass should be able to automatically detect the possible tensorization candidates and then perform the tensorization. This is why we call our version auto tensorization.

Specifically, users need to provide a list of predefined micro-kernels using the registry API. The provided micro-kernel contains the computation semantics expressed in TVM tensor expression, as well as the actual code to be executed (in our implementation it is a LLVM byte code module). The auto tensorization pass recursively visit the IR tree and then identify the tensorization candidate by checking the semantics of the loops against the provided micro-kernels. If a candidate is found, it replaces the loop with a call statement to the micro-kernel. Internally, this step is achieved by binding the portion of buffer of the tensor visited within the loop to the input/output buffer of the micro-kernels via the buffer API of TVM. TVM will lower the buffer into actual pointers and offsets of the input and output tensor in later IR passes. The current only supports CPU because for GPU targets, the IR transformation involves more complex steps such as memory planing among different threads.

3 Experiments

3.1 Micro-kernel templates

For the Micro-kernel template optimization, we want to explore two folds: 1) we want to demonstrate that the micro-kernel optimization is indeed better than the baseline computation. And we want to use an ablation study to show how each component of optimization accumulates and how those contribute to the performance improvement. 2) We want to show that the reason for using our tersorization framework that choose among a set of micro-kernels as one single mocro-kernel is designed to target a set of shapes (such as squared or thin rectangular shaped computation), and that the more one assumption is violated, the more using single micro-kernel's performance degrades. Thus for generalization, it is better to combine different micro-kernels with different shape assumptions that can suit complicated shapes by using combination of multiple micro-kernels.

We primarily use 2D matrix multiplication as an example as typical N-dimensional matrix computation an be generalized from 2D and that in the use case of TVM, the loop tensorlzation primarily targeting breaking down loops that computes matrix computation into a set of combined matrix multiplications according to the conditionals that is set inside the loops.

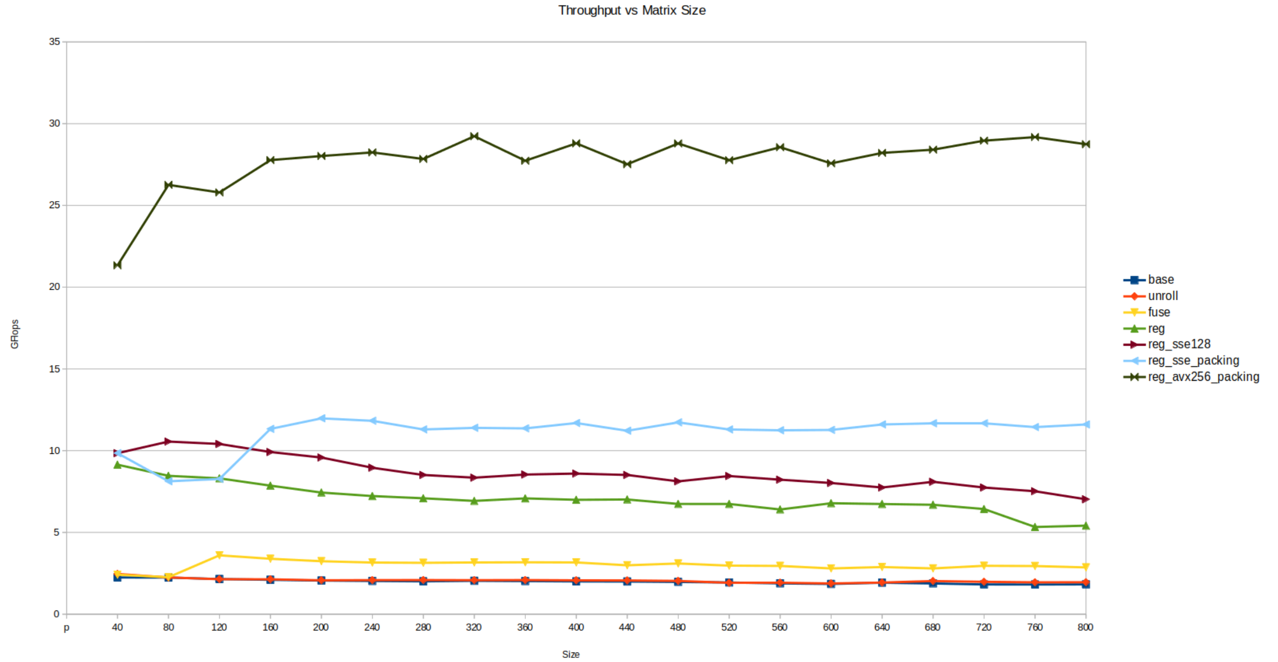

Figure 3: Throughput ablation study for square matrix multiplication with different side size and optimization techniques

To leverage the current architectural advancement of CPU, we use DOUBLE precision calculation and instruction sets. All following benchmarks are performed using Intel Core i7 6700k @4.2GHz with 32 GB DDR4 2400MHz RAM, Ubuntu 18.04 LTS, GCC 8.0. Run on a single core. Ablation study testing infrastructure was adopted from [Jianyu Huang, 2018].

The Figure 3 shows the ablation study on how different optimization techniques cons tribute to the overall performance. We use the generic squared matrix multiplication with different side sizes. The x axis denotes the side size of the square matrices, and y axis denotes the throughput under GFlops. We start with our very baseline of nested loop, one on one matrix multiplication, gradually apply loop unrolling by factor of 4, loop fusion that reuse the queries rows multiple times inside a inner loop, accumulating the result explicitly into registers, applying SSE128 instruction set to replace element-wise computation, apply packing that extract an rectangular part of the overall matrix that can fit into the L3 cache in contiguous addresses to be better loaded into L2, L1 cache. Finally, replacing SSE with AVX256 instruction set to leverage a more recently developed vectorized instruction set.

From the result we see that obviously the AVX256 instruction set is better than the SSE instruction by doubling the throughput under the same micro-kernel setting. Secondly accumulating the partial result in explicitly in allocated register improves from simply storing in variables as the variables might confuses the compiler to store it in cache and mess up the cache replacement. On a lower level, the loop fusion combined with the loop unrolling by 4 allows the same column for the same matrix to be used 4 time in the inner loop, allowing data reuse better than the baseline.

In particular, we would like to point out the importance of packing. The purpose of using an additional loop to extract a piece of larger portion of matrix before advancing to the inner loop to apply the micro-kernel improves the performance is because as matrix size increases, it is harder to fit into the L3 cache, thus looping either column-wise or row-wise may not take full advantage of the data reuse. Thus we pack out a smaller matrix allowing that "pack" to fit into the L3 cache to maximize the cache reuse. In the Figure 4 we see that using packing significantly improves the throughput when the matrix gets larger.

Finally, we use an ablation study to justify that it is essential to select from a wide range of micro-kernels as one micro-kernel's assumption cannot fit for all different variation of shapes.Figure 5 shows using a single micro-kernel optimized for computing squared matrix multiplication on different

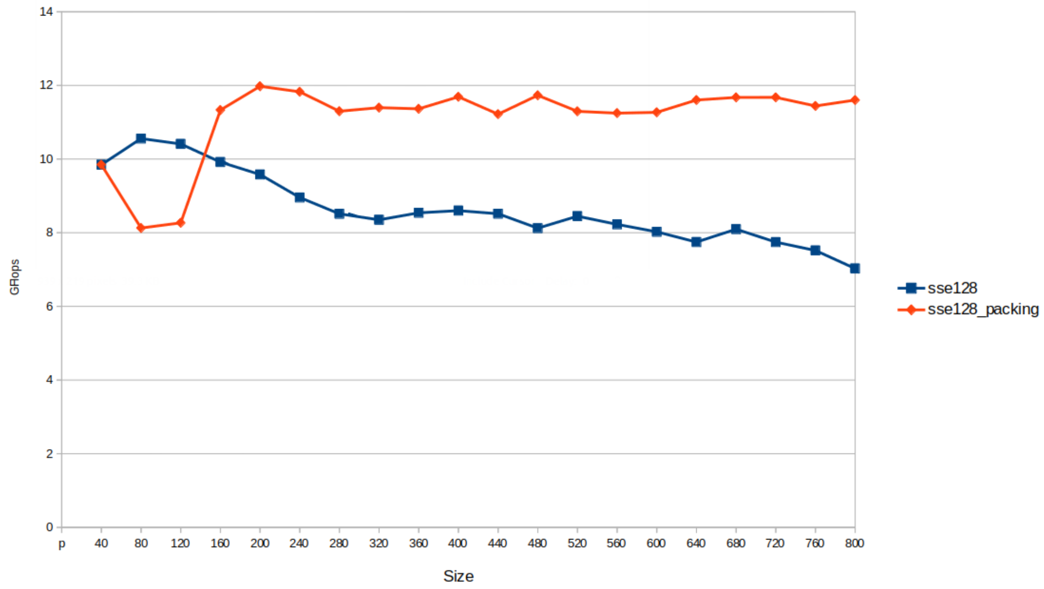

Figure 4: Throughput ablation for using and without using packing to the contiguous region inside the L3 cache

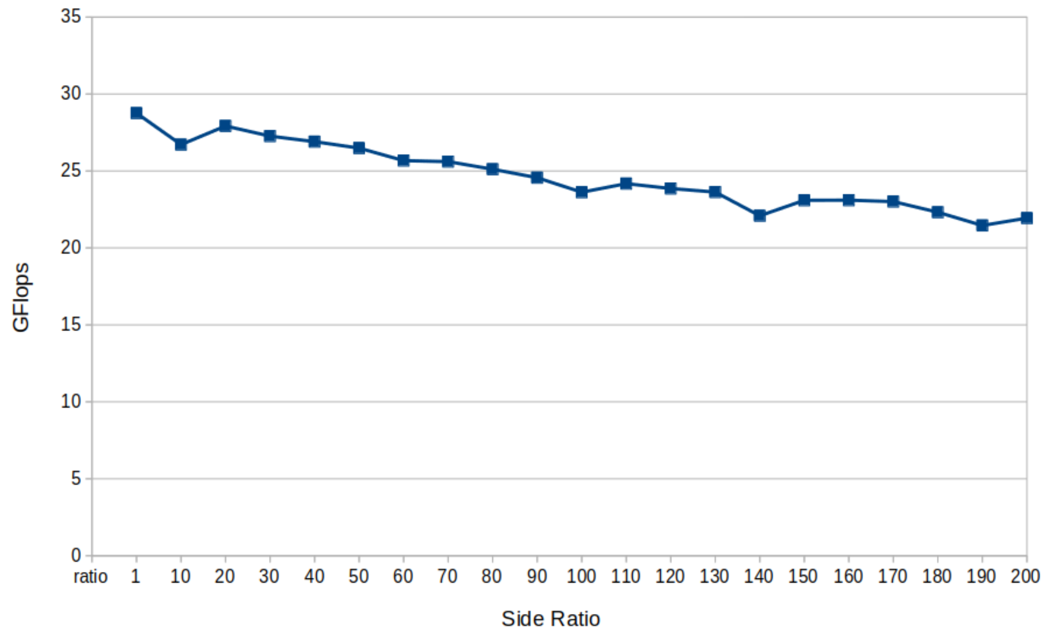

Figure 5: Performance degrade as the same micro-kernel applied to different shape of matrix multiplication with less reuse according to the assumed shape (square)

rectangularly shaped matrix multiplication. For M multiplies N with shape (m,k), (k,m) the x axis denotes the ratio between m and k. As the ration grows larger, the matrix is "stretched" thinner and less "square", making the micro-kernel fit less data in the matrix under squared container. This result in less data reuse as both packing and the registers are able to load less data at once into the L3 cache and vector registers, resulting in less data reuse and degraded performance.

3.2 Performance of dynamic shapes in TVM

To verify our optimization for dynamic shapes in TVM using the proposed auto tensorization, we use TVM to generate GEMM kernels with support for dynamic batch size for the input. We compare our performance with the GEMM kernels generated by TVM 1) with fixed input shapes 2) with support for dynamic input shapes but does not utilize tensorization. We also compare our performance with OpenBLAS library [Xianyi et al., 2012]. Note that for GEMM kernels with fixed input shapes, the input must have the required shape otherwise a runtime assertion fails. Therefore, to take another input shape, we need to generate a new GEMM kernel for the newly-specified shape again.

The experiments are performed on a AWS c5.xlarge instance with Intel Skylake architecture CPU and 8GB memory. The LLVM version is 6.0.

We select different input shapes for GEMM in the experiment. The results are reported in GFLOPS in Figure 6, using auto tensorization, the performance is nearly the same as that of the fixed-input-shape GEMM kernels. Besides, since TVM does not support tensorization for dynamic input shapes previously, the performance for dynamic input shapes are not well-optimized. Therefore, our optimization drastically improves the performance for dynamic input shapes. However, our performance is still below that of OpenBLAS. We think that the performance of the micro-kernels can still be improved.

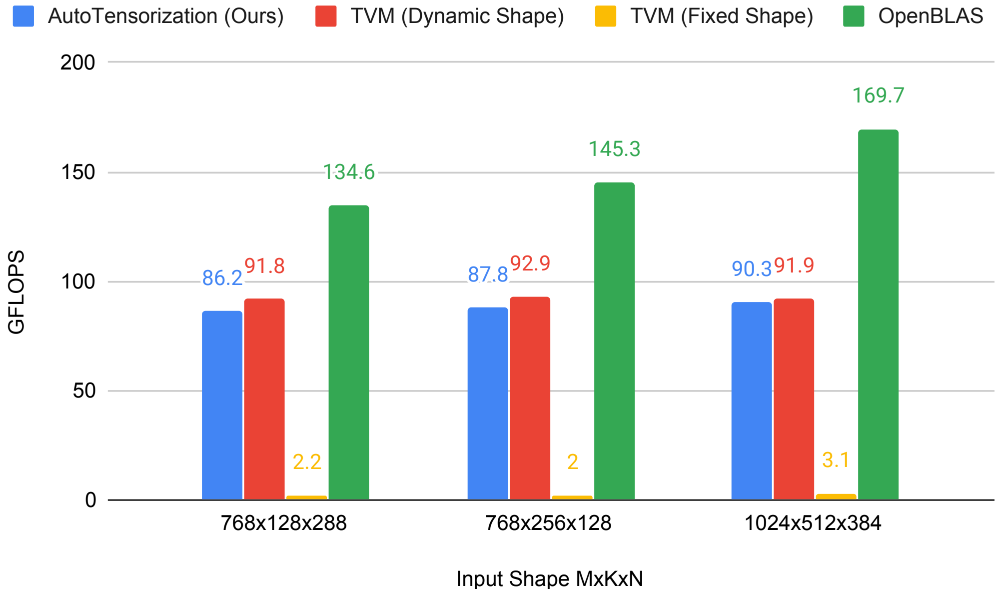

Figure 6: Benchmark of GEMM with Input $[M, K]$ and $[N, K]$

4 Surprises and Lessons Learned

We originally plan to construct a cost model that can guide the loop tensorization process through the pre-computation of cache reuse. However, we found that the TVM IR is not obvious to extract each partition of the computation with variable input sizes. Thus We turned to explore the pipeline to construct micro-kernel construction and use the developed empirical result to construct a set of micro-kernels that could fit for general deep learning tensor computation.

As for the TVM part, we originally plan to adjust the order of optimization passes of TVM in order to apply the loop partition pass first to eliminate the conditional statements and then perform the tensorization. However, we found that the original tensorization pass in TVM requires information from the original computation body. If we perform the loop partition first, such information in the original computation body is lost and therefore the tensorization pass is no longer applicable.

5 Conclusions and Future Works

For the micro-kernel part, we adopted the comprehensive framework of GOTOBLAS and extended that to adopt the AVX2 instruction set which greatly improved the throughput compared to the SSE instruction set used by original authors. For future work, we encourage researchers to look into auto generating micro-kernels during compile time. Mostly current compilers are mainly using hand tuned templated micro-kernel that is optimized for a specific set of shapes. It is not a trivial task to construct optimized micro-kernel ad-hoc during the compile time according to mostly shared set of shapes in the program. The conditionals may break matrix computations into irregular shapes and little assumption can be made. Using greedy algorithm to search through the search space to find optimal or sub-optimal set of shapes to construct micro-kernels ad-hoc can be expensive as the search space can be really large. Future development can explore optimizations on micro-kernels that can compress and support matrices with "holes" might alleviate the issue.

We applied the loop partition pass to eliminate conditional statements and then used the proposed auto tensorization pass to match the nested loops against the provided micro-kernels and then perform the tensorization optimization. The proposed optimization has improved the performance for dynamic input shapes. Currently, the micro-kernel-marching stage uses some ad-hoc algorithms that only work for certain kinds of micro-kernels such as GEMM micro-kernels. Because of the discrepancy between the original computation definition and the IR after a series of transformations, the marching algorithm is difficult to generalize for all the patterns of micro-kernels. We leave it as a future work to implement general marching algorithm for detecting tensorization candidates.

6 Project Website

`https://yingjinglu.github.io/pages/compiler-proj.html`

7 Distribution of Total Credit

Total credit is distributed equally.

References

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 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, Carlsbad, CA, October 2018a. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018b.

Robert van de Geijn Jianyu Huang. *How To Optimize Gemm*, 2018. URL https://github.com/flame/how-to-optimize-gemm.

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. *SIGPLAN Not.*, 48(6):519–530, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL https://doi.org/10.1145/2499370.2462176.

Field G Van Zee and Robert A Van De Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):1–33, 2015.

Z. Xianyi, W. Qian, and Z. Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691, 2012.

Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.