ML Computation Graph Operator Fission and Fusion

Ashwin Venkatram[2] and Gabriele Oliaro[1]

[1]Department of Computer Science, Carnegie Mellon University
[2]Department of Electrical and Computer Engineering, Carnegie Mellon University

Abstract

The execution of deep neural networks (DNNs) on modern hardware accelerators can be approached as a mapping problem from tensor operators to hardware-specific kernels. Previous works apply operator fusion for this mapping. Usually governed by human heuristics, operator fusion with greedy approach has always been suboptimal.

Traditionally, graph level operator fusion respects the boundary of each operator and is constrained to silicon-vendor provided library support for accelerated kernels.

We propose that operator fission to generate a primitive operator graph exposes inter-operator optimization opportunities for a global search algorithm. We hypothesize that TVM's global genetic algorithm can find better fusion opportunities on a fissioned graph.

For the end-to-end case of Segformer, we find that operator fission followed by graph optimizations result in 20% speed-up over the vanilla computation graph.

1 Introduction

The paper introduces a new method to optimize kernel mapping for tensor programs to enable efficient execution of deep neural networks (DNNs). The current bottleneck in DNN execution is the mapping of a high-level DNN representation to hardware-specific computations that can be executed on accelerators. Existing frameworks use heuristic based operator fusion to map DNN operators to hardware-specific kernels, but this approach has limitations. The paper proposes a new approach that applies operator fission to decompose tensor operators into basic primitives and then uses the global genetic search algorithm in TVM to find and codegen more performant kernels for GPU hardware execution.

1.1 Related works

The most relevant existing works focus can be divided into three groups, based on what they focus on:

- Kernel mapping: frameworks such ad DNNFusion [8], or PyTorch 2.0 use rule-based fusion strategies to map tensor operators to the best pick among a set of existing kernels. DNNFusion, for example, first classifies the operator in a DNN into five groups (One-to-One, One-to-Many, Many-to-Many, Reorganize, and Shuffle) depending on the relation between the input and output tensors. Then, it picks the One-on-One operators whose fusion has the potential to yield the best performance gains (fusion seed operators). Then, it tries to greedily fuse the successors and predecessors. Unlike these systems, our system uses a more systematic approach to optimize kernel mapping by first applying operator fission to decompose tensor operators into basic primitives and then using TVM [3] to perform the code generation that will discover an optimal kernel execution strategy for each graph of primitives.

- Graph optimization: Several existing frameworks in the ML compilation sphere perform optimizations by transforming a DNN's computational graph. Systems such as TensorFlow [2], TensorRT [11], and MetaFlow [6], for example, apply graph transformation rules designed by domain experts. Other frameworks, such as TASO [5] and PET [12], attempt to generat the graph transformations automatically, and use backtracking search to apply the generated transformations. These works are complementary to our technique, as kernel mapping optimizations can be combined with existing graph-level optimizations. For instance, our system can use the graph transformations discovered by TASO to optimize the primitive graphs.

- Hardware-specific kernel generation: Frameworks such as Halide [7, 9], as well as many popular deep learning compilers, such as FlexTensor or TVM [3, 14], separate algorithm and schedule in two distinct components and automatically generate hardware-specific kernels for DNN computation. In particular, Halide proposes several strategies to discover highly optimized schedules,

whereas TVM [4] uses a learning-based approach to predicting the cost of a schedule and discovering efficient schedules in a pre-defined schedule space. Finally, Ansor [13] automatically generates schedule templates, resulting in more performant schedules than TVM. These techniques are also orthogonal to the optimizations proposed in our system; in fact, we directly use TVM and Ansor for generating high-performance kernels for a given set of primitives.

2 Design

2.1 Operator Fission

Our system decomposes each tensor operator into a small set of basic tensor *primitives*. Primitives are the smallest unit of computation; and they are required to only involve computations with the same parallelism degree and data access pattern. Thanks to this property, each primitive can be efficiently run in a single kernel. We categorizes primitives into four groups based on their input-output relationships. In particular, given a primitive p, the input tensor(s) I, of size n, and the output tensor [1] $O_p(I)$, we can define $O[\vec{x}]$ to be the output value at position $\vec{x} = (x_1, ..., x_m)$ and $I_k[\vec{x}]$ as the input value at position $\vec{x}$ of the k-th input tensor I_k. We will use this notation below to help define each type of primitive.

The four categories of primitives that we considered in our project were:

1. Elementwise primitives. For an elementwise primitive p, the output tensor has the same shape and data layout as all input tensors. The computation for each output element depends on the input elements at the same position, i.e.,
$$O[\vec{x}] = f(I_1[\vec{x}], I_2[\vec{x}], ..., I_n[\vec{x}])$$
Elementwise primitives do not require layout transformations and can be fused with other primitives as a preprocessing or postprocessing step.

2. Reduce and broadcast primitives. A reduce primitive takes a single input tensor and calculates the aggregated result of each row of the input tensor along a given dimension. It aggregates along the k-th dimension of I_1 and uses an aggregator $\oplus$ to compute
$$O[x_1, ..., x_{k-1}, x_{k+1}, ..., x_m]$$
On the other hand, a broadcast primitive replicates a single input tensor along a given dimension, i.e.,
$$O[\vec{x}] = I_1[x_1, ..., x_{k-1}, x_{k+1}, ..., x_m])$$

3. Layout transformation primitives. A layout transformation primitive transforms the data layout of a single input tensor without performing arithmetic operations. It maps position $\vec{x}$ of the output tensor to position $\vec{u}$ of the input tensor using a one-to-one mapping function $L(\vec{x})$, i.e.,
$$O[\vec{x}] = I_1[L(\vec{x})]$$
Concatenation and split are considered as layout transformation primitives since they require changing the underlying layouts of the input tensors.

4. Linear transformation primitives. Lastly, Sys captures compute-intensive operators in DNN models, such as matrix multiplication and convolution, using linear transformation primitives. A primitive p is considered a linear transformation primitive if its output is linear with respect to all input tensors. That is, $\forall Y, Z, \vec{x}, 1 \le k \le n$:
$$O(I_1, ..., I_{k-1}, Y + Z, ...)[\vec{x}] = \\ = O(I_1, ..., I_{k-1}, Y, ...)[\vec{x}] + O(I_1, ..., I_{k-1}, Z, ...)[\vec{x}]$$
and
$$\alpha \cdot O(I_1, ..., I_{k-1}, Y, ...)[\vec{x}] = O(I_1, ..., I_{k-1}, \alpha \cdot Y, ...)[\vec{x}]$$

More concretely, we list a few primitive types and the corresponding representative operators in Table 1 below.

Table 1: Example of some frequently used primitves.

Primitive Type	Representative Operators
Elementwise	Add, Sub, Mul, Div, Relu, Sqrt, Erf
Reduce and broadcast	ReduceSum, ReduceMean, MaxPool, Broadcast
Layout transformation	Transpose, Split, Concat, Slice, Pad, Reshape
Linear transformation	Conv, GEMM, Batched GEMM

To represent the primitive operator computation graph, we use ONNX. ONNX is an open source format for AI models, and its operators. The operator fission pass, implemented in Onnx GraphSurgeon, takes an ONNX graph as input, performs operator fission leveraging ONNX primitives, and outputs the modified computation graph.

As a case-study, we examine operator fission and associated speed-up for the Softmax operator. Other operators such as InstanceNorm, BatchNorm, etc can also leverage this innovation once proven.

For example, consider the commonly-used `softmax` operator, which converts a vector of numbers to a vector of probabilities:
$$\texttt{softmax}(x_i) = e^{x_i} / \sum_j e^{x_j} \quad (1)$$

The `softmax` operator includes three element-wise computations: the exponential function, a vector-wise aggregation (summation term in the denominator), and vector-wise broadcast (i.e., the scaling factor $s = 1/\sum_j e^{x_j}$ is used to compute

[1] Multiple output tensors can be handled by analyzing them sequentially.

all output elements). These components in `softmax` involve different degrees of parallelism and memory access patterns, making it challenging to generate a single high-performance kernel for `softmax`.

Understandably, the bottleneck is in the calculation of the summation term in the denominator. This also blocks any fusion opportunity with downstream nodes.

An example of Softmax operator fission is shown in Fig. 1.

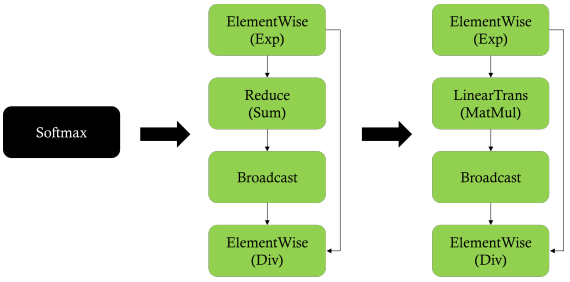


Figure 1: SoftMax Operator Fission

By performing operator fission, it is possible to generate a dedicated kernel for each of these components. However, this would be inefficient due to high kernel launch overhead and unnecessary instantiations of intermediate results. We obtain significant improvements by performing inter-operator fusion of the operator's subcomponents at a finer-grained level if fusion opportunities exist with preceding or succeeding operator.

2.1.1 Reduction as a Linear Transformation

A key innovation in this operator fission is that the ReduceSum operator in Softmax can be representated as as linear transformation, MatMul operation. A reduction operation is costly and is always the bottleneck to parallelization and efficient computation.

Representing a ReduceSum as a MatMul exposes potential optimization with upstream or downstream MatMul operations as discussed below.

2.1.2 MatMul Horizontal Fusion Heuristic

We implement a secondary pass within the operator fission implementation to discover MatMul horizontal fusion as shown in Fig 2.

This enable potential fusion with upstream or downstream MatMul operations, if any. In the sub-graph test case shown in Fig. 3, we explicitly include a MatMul node. The division operation can be delayed, while still preserving semantic equivalence, and the MatMuls can be horizontally fused as a single operation. The only overhead here is the insertion of Concat and Split node to access the desired tensors at a later stage.

2.1.3 SoftMax Semantic Equivalence

During operator fission, it was discovered that the ONNX operator internally implements a normalization operation on the input tensor before passing it to the Exp primitive. To ensure semantic equivalence, the primitive fissioned graph includes the ReduceMax operation as well.

With this approach, downstream code-gen can be efficient. HPC techniques can be leveraged for efficient data-forwarding from one CUDA kernel call to the next, and reduce system calls for DRAM allocations. We believe that TVM's efficient code-gen backend will handle this.

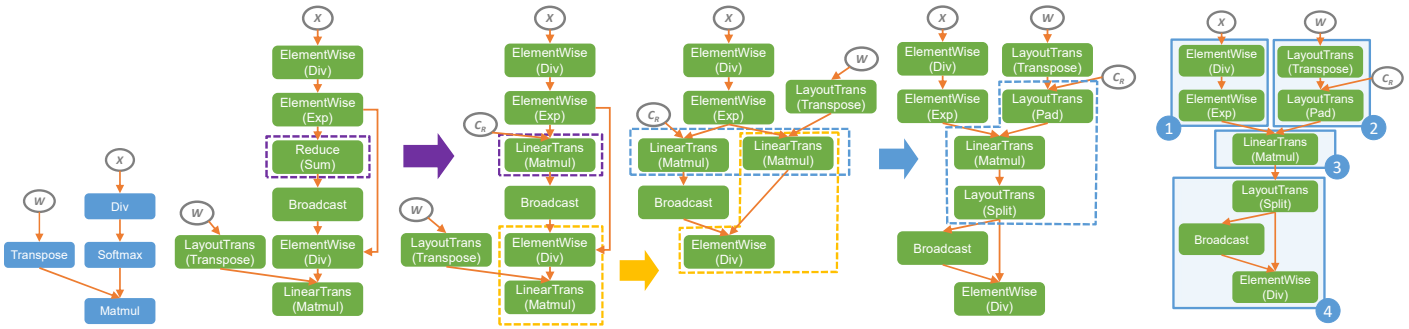
Data: ONNX graph
Result: Primitive ONNX graph
while *not last node* **do**
  get_current_node();
  find_upstream_node();
  next_node = find_downstream_node();
  MatMul_node = find_MatMul();
  if *node == SoftMax and MatMul_node !=NULL*
   then
    disconnect_node();
    perform_op_fission_w_MatMul();
  else
  end
  if *node == SoftMax and next_node ==NULL* **then**
    // this is last node, no MatMul pass
    disconnect_node(); perform_op_fission();
  else
end

Algorithm 1: Graph Parse Algorithm

We implement a graph parsing algorithm in ONNX Graph Surgeon illustrated in Algorithm 1 followed by unique passes that would perform the operator fission with or without the MatMul horizontal fusion innovation. The MatMul pass is allowed to trigger if and only if element-wise operations are found between the MatMul operations.

2.2 TVM Kernel Codegen

We leverage MetaSchedule [10], a probabilistic scheduler DSL developed in Tensor IR (TIR), to auto-generate high performing kernels for the computation graph and target GPU hardware. MetaSchedule accepts an ONNX graph input, lowers it to TIR schedule given pre-defined scheduling rules, and leverages Ansor backend to generate the best-performing schedule. The profiler then returns the latency measurement of each kernel task as well as the end-to-end cost of model execution.



(a) Comp. graph. (b) Primitive graph and transformations. (c) Kernel mapping.

Figure 2: Operator fission enables subsequent optimizing transformations on primitive graphs. In 2b, the combination of the three transformations fuse the reduce primitive in `softmax` and the subsequent `matmul` into a single `matmul`.

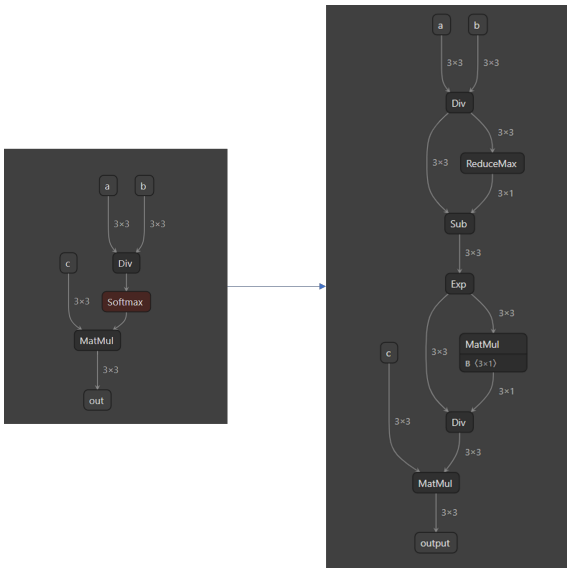


Figure 3: SoftMax Operator Fission

2.3 Experiment Setup

Platform Our evaluation is performed on Nvidia V100 GPU and A100 GPU. For Nvidia V100 GPU, we use an AWS p3 series instance [1] with Deep Learning AMI GPU PyTorch 1.13.0 (Ubuntu 20.04), which is equipped with a 8-core Intel Xeon CPU, a V100 16GB SXM GPU and CUDA 11.7.

We utilize ONNX-GraphSurgeon and TVM to implement this work and leverage TVM runtime to benchmark our results. We also compare with TensorRT where possible.

Workloads We selected to run our experience on the following test cases:

1. Linear Attention subgraph
2. Softmax-subgraph
3. Segformer model (resolution: 512x512) - a vision transformer for semantic segmentation

Note: All the DNNs are run with FP32 precision

2.4 Results

In this section, we present our results for the three workloads mentioned above.

Linear Attention Subgraph TensorRT has particular optimizations targeting linear attention. Nevertheless, for the QKV linear attention sub-block with Softmax, we observe a **2X speed-up over TensorRT**.

Softmax Subgraph Examining the Softmax subgraph in isolation, we observe a **1.11X speed-up over TensorRT**.

In both subgraph cases, we attribute the speedup observed to the two key innovations detailed in the earlier section. Primitive operator fission does expose better fusion search opportunities. Expressing ReduceSum as a Linear Transformation (MatMul) enables discovery of further fusion with upstream and downstream nodes for efficient kernel codegen by TVM Metaschedule.

Segformer ViT Running vanilla Segformer through TVM generates 130 kernels with a mean E2E running time of **5.8484ms**.

Op fissioned Segformer with TASO graph through TVM generates 119 kernels. With full semantic equivalence, the optimized model achieves an E2E execution time of **4.8977ms**.

This is **20%** faster than the vanilla implementation.

2.5 Analysis

The results above show that our fission technique is able to unlock some exciting performance improvements. The gains are particularly significant for Segformer ViT case. While a 20% speedup might not sound very large, we have to consider that Segformer is a model that has already been heavily optimized by existing DNN frameworks, including TVM.

The performance improvements are made possible by operator fission by decoupling coarser-grained operators into small primitives whose computation graph can be more easily transformed.

3 Surprises and Lessons learned

We applied the learnings from this class to develop a generalized heuristic for operator fission and applied it to the Softmax testcase.

In implementing operator fission we discovered that we needed to handle the cases where the SoftMax node is embedded between other nodes, versus at the end of the computation graph.

We also had to handle semantic equivalence of the fissioned SoftMax node as per the ONNX standard manually by including the ReduceMax operator into the fissioned computation graph. While the illustrated cases in this paper show good fusion opportunity, the addition of the ReduceMax node presents a new boundary to operator fusion in the fissioned graph.

We believe this work has potential and can be extended to other operators as well.

A challenging aspect in our project was the time required to iterate on the graph optimization. For example, optimizing the Segformer model took us about 12h of computation each time. This issue was not a full surprise to us, as we encountered similar bottlenecks in our research project before. However, it forced us to think very carefully how to design each optimization job before running it.

4 Conclusion and Future Work

In conclusion, we show that operator fission to a primitive subgraph exposes more aggressive fusion opportunities with preceding and succeeding DL operators.

While small subgraph test cases showed promise over TensorRT, running an end-to-end model, Segformer through operator fission pass, graph optimizations with TASO, followed by efficient codegen with TVM improves performance by 20%.

Although implemented only for the SoftMax node, the more aggressive fusion opportunities provided to TVM graph compiler shows potential for further work with other reduction operators.

This entire workflow can be automated and integrated into industry-grade deep learning compilers and systems stack.

Potential directions to extend this work:

1. Integrate operator fission pass from ONNX-GraphSurgeon (python) into TVM Relay and MLIR graph backend passes (C++)

2. Build on top of this proof-of-concept pass to create a generalized operator fission pass that can be applicable to other Reduction nodes such as InstanceNorm, BatchNorm, to name a few

5 Distribution of Work

50%-50% split between both team members.

References

[1] Amazon ec2 p3 instances. https://aws.amazon.com/ec2/instance-types/p3/, 2022.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[4] 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 31*, NeurIPS'18. 2018.

[5] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[6] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[7] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), 2016.

[8] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[9] 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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[10] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs, 2022.

[11] NVIDIA TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017.

[12] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[13] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[14] 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.