FusionStitching: Boosting Execution Efficiency of Memory Intensive Computations for DL Workloads

Guoping Long, Jun Yang, Wei Lin

guopinglong.lgp,muzhuo.yj,weilin.lw@alibaba-inc.com

Abstract

Performance optimization is the art of continuous seeking a harmonious mapping between the application domain and hardware. Recent years have witnessed a surge of deep learning (DL) applications in industry. Conventional wisdom for optimizing such workloads mainly focus on compute intensive ops (GEMM, Convolution, etc). Yet we show in this work, that the performance of memory intensive computations is vital to E2E performance in practical DL models.

We propose *FusionStitching*, a optimization framework capable of fusing memory intensive *elementwise*, *reduction* and fine grained *GEMM/Batched-GEMM* ops, with or without data dependences, into large computation units, then mapping and transforming them into efficient GPU kernels. We formulate the fusion plan optimization as an integer linear programming (ILP) problem, and propose a set of empirical heuristics to reduce the combinatorial search space. In order to map optimized fusion plans to hardware, we propose a technique to effectively compose various groups of computations into a single GPU kernel, by fully leveraging on chip resources like scratchpads or registers. Experimental results on six benchmarks and four industry scale practical models are encouraging. Overall, *FusionStitching* can reach up to 5.7x speedup compared to Tensorflow baseline, and achieves 1.25x to 1.85x performance speedups compared to current state of the art, with 1.4x on average (geometric mean).

Keywords   Compiler, GPU, Artificial Intelligence

1 Introduction

Recent years have witnessed a surge of industry scale applications of DL models, ranging from text/NLP, audio/speech, images/videos, to billion scale search and recommendation systems[31]. Such workloads are typically expressed with high level Python APIs, modeled as computation DAGs, and mapped to hardware accelerators (such as GPUs) through domain specific execution frameworks (such as Tensorflow[4], PyTorch[3], Mxnet[9], etc). The challenge is how to transform high level computation graphs into efficient kernels in order to maximize the execution efficiency on hardware.

In DL workloads, dense tensor computations (GEMMs, Convolutions, etc) are ubiquitous. Thus many prior research works either focus on optimizing performance of such compute intensive primitives[10, 16, 29], or target a kernel selection and computation scheduling problem[6, 25]. This approach works well for workloads that are dominated by FLOPs efficiency of GEMMs or Convs, for instance, convolutional neural networks[15, 24, 27]. However, recent advancement of the DL domain has resulted in many novel model structures which are dominated by memory intensive patterns (element wise layers, layout transpose or reduction operators). In addition, there are models with a large number of fine grained (< 10us) operators, causing notable runtime launch overheads when executing on GPUs. For these workloads, optimizing compute intensive ops alone is inadequate to unlock the full potential of execution efficiency.

To this problem, libraries such as cuDNN/cuBlas have the capability to fuse element wise layers into large compute intensive kernels. It is also possible to fuse multiple GEMMs into a large GEMM or a batched GEMM op[25]. However, these techniques do not work well for workloads that are dominated by memory intensive structures.

Another known approach is kernel fusion, a technique to fuse multiple memory intensive ops with data dependencies into a single kernel to reduce off chip memory accesses. Prior works have explored this idea extensively in database[32], image processing[5, 16, 23], HPC applications[18, 30], and AI workloads[7, 19]. However, there are two notable limitations when targeting memory intensive DL models. First, current techniques mainly focus on reducing memory traffic instead of reducing the number of kernel launches, therefore give up fusing as many computations with no data dependences as possible. Second, existing works only compose elementwise or reduction layers, lacking the ability to fuse and optimize compute and memory intensive ops together.

In this work, we propose *FusionStitching*, an optimization framework to systematically perform fusion space exploration and aggressive code generation. One key observation underpinning our methodology is the inherently repetitive nature of DL workloads. That is, by optimizing thoroughly once, it is possible to reuse the optimized implementation in future runs.

FusionStitching addresses limitations faced by current state of the art by composing as many fine grained ops as possible, with or without data dependences, into large GPU kernels, in order to reduce off chip memory accesses and launch overheads simultaneously. In particular, it comes with the capability to optimize compute and memory intensive ops

Guoping Long, Jun Yang, Wei Lin

in a uniform optimization scope, and perform global fusion and code generation in entirety. Besides, we make three unique contributions to the community:

- Naive composition of multiple computations may cause notable performance slowdown, because different portions of the kernel may have conflicting memory layout, parallelization or on chip resource requirements[34]. In this work, we formulate the fusion plan optimization as an integer programming problem. In addition, we introduce effective domain specific heuristics to make the solution space exploration practically tractable.
- To the best of our knowledge, this is the first work to integrate compute intensive ops into the fusion and kernel generation of memory intensive computations.
- We propose a lightweight, domain specific representation of potentially complex kernel implementations. The compact representation enables separation between the kernel optimization intent and implementations, and facilitates parametric performance tuning effectively.

Experimental results on six benchmarks and four practical industry models show promising results in both kernel compression ratio (kernel number reduction) and performance speedup. Compared to the Tensorflow XLA[19] baseline, *FusionStitching* can achieve up to 10x kernel compression ratio, with 2.8x on average. As to performance, *FusionStitching* can reach up to 5.7x speedup compared to Tensorflow baseline, and achieves 1.25x to 1.85x performance speedups compared to XLA, with 1.4x on average. Please note that although all benchmarks are expressed with Tensorflow APIs, our approach applies to other DL frameworks (such as MxNet, PyTorch, etc) as well.

The rest of the paper is organized as follows. Section 2 presents characterization of DL workloads that motivate this work. Section 3 presents the high level sketch of our approach. Section 4 and 5 presents our fusion and kernel generation mechanisms, respectively. Section 6 discusses experimental results. Section 7 discusses related works and section 8 concludes this work.

2 Motivation

We present key observations of emerging DL workloads to highlight the significance of memory intensive ops. Table 1 summarizes our workloads, including 4 production models under deployment at Alibaba, and 6 micro-benchmarks from Tensorflow-Examples[8]. Application domains of these benchmarks are diversified, ranging from basic ML algorithms (*logistic*, *word2vec*, *rnn*, *perceptron*, *var-encoder*), NLP (*nmt*[33], *aiwriter*), audio (*rokid*), to internet scale E-commerce search and recommendation systems (*multi-interests* [12, 31]). Among them, *nmt* is an inference application, and all others are training models.

Table 1. Workload Description

Category	Name	Description
Application	nmt	Neural machine translation
Application	multi-interests	Recommender systems
Application	rokid	Speech recognition
Application	aiwriter	Dialog generation
Micro-Benchmark	logistic	Logistic regression
Micro-Benchmark	word2vec	Word to vector embedding
Micro-Benchmark	bi-rnn	Bi-directional RNN
Micro-Benchmark	dyn-rnn	Dynamic RNN
Micro-Benchmark	perceptron	Multilayer neural network
Micro-Benchmark	var-encoder	Variational encoder

Table 2. Workload Characteristics

Name	#Graph Size	#Kernels	Avg. Kernel Size	Mem. Ratio
nmt	1532	5940	9.2us	46%
multi-interests	992	495	202.8us	81%
rokid	3204	261029	4us	47%
aiwriter	27428	398265	5.4us	80%
logistic	62	1540	3.1us	60%
word2vec	206	64	2.5us	71%
bi-rnn	461	1085	3.7us	56%
dyn-rnn	957	1352	3.2us	63%
perceptron	108	49	5.4us	47%
var-encoder	239	112	4.5us	53%

Table 2 presents key characteristics of our workloads. *#Graph Size* denotes the number of Tensorflow ops of the running target graph. *#Kernels* denotes the number of GPU computation kernels. In this context, we do not consider data transfer kernels back and forth between CPU and GPU, because all workloads are dominated by computation. *Avg. Kernel Size* shows average kernel execution time, a metric to measure kernel granularity. *Mem. Ratio* denotes the percentage (exe. time) of memory intensive kernels in all computation kernels.

As can be seen, in all workloads except *multi-interests*, average kernel time is less than 10us. This is close or even smaller than a single driver launch latency[21, 25]. Even for *multi-interests*, close examination reveals interesting possibilities to optimize compute and memory intensive computations from global perspective (Section 3). Besides fine grained kernels, there are up to 60% on average of *Mem. Ratio* for all workloads. These observations motivate *FusionStitching*, our solution to boost the execution efficiency for memory intensive computations.

3 Overview

3.1 A Motivating Example

In this section, we present a motivating example from *multi-interests*. It is a pattern generated by the fusion process (Section 4), with irrelevant details omitted for illustration purposes. Figure 1 (a) and (b) show the data flow graph and computation specification, respectively. Here we adopt the XLA tensor operation semantics[2] in our presentation.

This example illustrates the necessity to optimize compute and memory intensive ops collectively. It includes a combination of nine elementwise, two reduction and two batched GEMM (*dot_1* and *dot_2*) ops. The terminal *tuple* groups all output tensors (*divide*, *log_1*, *multiply_2* and *subtract*) of this fused computation. In particular, *dot_1* takes two small tensors (12MB) as input and generate a large (552MB) output tensor. Finally, *dot_2* takes the large tensor originated from *dot_1* and produces a small output. Fusing all these ops into a kernel reduces off chip memory accesses substantially caused by intermediate results.

We highlight three design principles to high performance kernel generation for such complex fused computations. First, the work space of the entire computation must be parallelizable across shape dimensions. Second, on chip memory is preferable to transfer intermediate results, thus avoid redundant computation. Third, fast memory (such as registers) is preferred over relatively slow storage (such as shared/L2 memory) for computation reuse. Figure 1 (c) shows an implementation template. It defines one possible implementation of the fused kernel. This is achieved by specifying how to parallelize the work space (through *GRID,CTA,WARP*), how to transfer intermediate results (through *S* attribute), and kernel launch parameters (through *cta_num* and *cta_size*). We present details of kernel generation in Section 5.

3.2 Overview

Figure 2 shows the overview of our approach. In the fusion phase, the fusion plan generator takes domain specific heuristics, and generate a modest number of fusion patterns for evaluation. The fusion planner takes a cost model and produce a score for each pattern. Then the ILP solver outputs an optimized fusion plan for the entire computation, which is executed to generate a fused representation for kernel generation.

In the kernel generation phase, a template generator produces many templates, each with different trade-offs among parallelization, intermediate results sharing and launch settings. Since a template uniquely identifies a kernel, the kernel tuner can iterate over the implementation space by traversing these compact templates. Given a template, the resource planner orchestrates on chip resource usage. Instead of generating LLVM IR representation directly, the CUDA emitter produces a CUDA C kernel for diagnosis. With increasingly
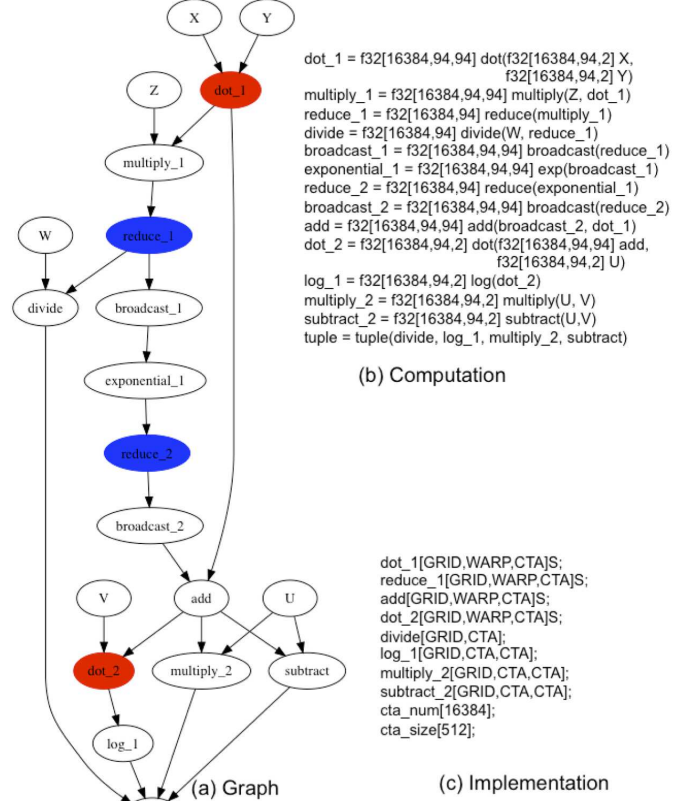


```
dot_1 = f32[16384,94,94] dot(f32[16384,94,2] X,
                              f32[16384,94,2] Y)
multiply_1 = f32[16384,94,94] multiply(Z, dot_1)
reduce_1 = f32[16384,94] reduce(multiply_1)
divide = f32[16384,94] divide(W, reduce_1)
broadcast_1 = f32[16384,94,94] broadcast(reduce_1)
exponential_1 = f32[16384,94,94] exp(broadcast_1)
reduce_2 = f32[16384,94] reduce(exponential_1)
broadcast_2 = f32[16384,94,94] broadcast(reduce_2)
add = f32[16384,94,94] add(broadcast_2, dot_1)
dot_2 = f32[16384,94,2] dot(f32[16384,94,94] add,
                             f32[16384,94,2] U)
log_1 = f32[16384,94,2] log(dot_2)
multiply_2 = f32[16384,94,2] multiply(U, V)
subtract_2 = f32[16384,94,2] subtract(U,V)
tuple = tuple(divide, log_1, multiply_2, subtract)
```

(b) Computation

```
dot_1[GRID,WARP,CTA]S;
reduce_1[GRID,WARP,CTA]S;
add[GRID,WARP,CTA]S;
dot_2[GRID,WARP,CTA]S;
divide[GRID,CTA];
log_1[GRID,CTA];
multiply_2[GRID,CTA,CTA];
subtract_2[GRID,CTA,CTA];
cta_num[16384];
cta_size[512];
```

(c) Implementation

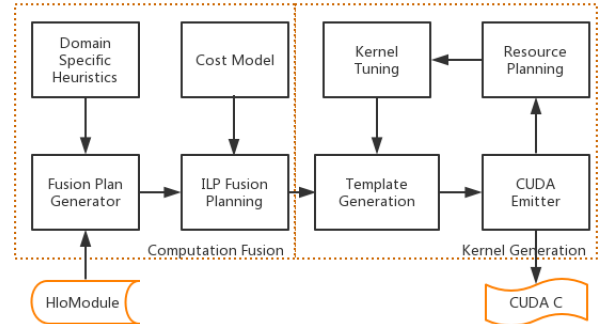(a) Graph

Figure 1. A Motivating Example

Figure 2. Overview

complex fusion patterns, this greatly ease debuggability and performance tuning as well.

4 Computation Fusion

4.1 The ILP Formulation

Given a computation graph $G = (V, E)$, with V and E are sets of vertices and edges, respectively. We define a fusion pattern $P_i = (V_i, E_i)$ as a subgraph of G, with $V_i \subseteq V, E_i \subseteq E$. For each fusion pattern P_i, we define an integer variable X_i where $0 \le X_i \le 1$, and a real score function $f(X_i)$ where
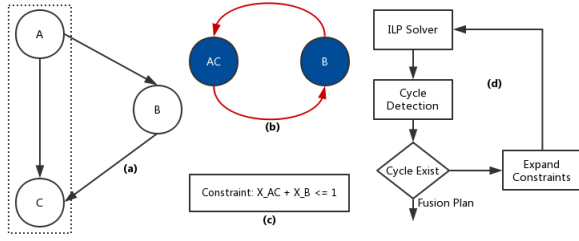
Figure 3. Fusion Plan: Cyclic Dependence

$f(X_i) \geq 0$. The larger the score $f(X_i)$, the higher the fusion gain w.r.t. performance. Given a set of fusion patterns $S = \{P_1, \ldots, P_k\}$, we have to

$$\text{maximize} \sum_{j=1}^{k} X_j f(P_j)$$

subject to: $X_u + X_v \leq 1, \forall u, v, 0 \leq u, v \leq k, P_u \cap P_v \neq \emptyset.$

We define a fusion plan of a computation graph as a subset of S. Informally, the objective is to resolve a fusion plan S^- in order to maximize the total fusion gain, such that any two fusion patterns in S^- are disjoint. That is, for any node in the graph, it can only exist in at most one fusion pattern.

We require that the score $f(X_i) \geq 0$. This means we give up all fusion patterns with negative gains. In theory, multiple patterns with negative scores may be fused together to produce a composite one with positive gain. There are two things to note. First, for practice, our design rationale is to cover known performance critical patterns in DL workloads, rather than exhausting all combinatorial possibilities, a prohibitively expensive operation. Second, even if we do not perform exhaustive search, there is still high probability that our domain specific pattern generator can discover the composite pattern. We discuss domain specific fusion space explorations in the next subsection.

Note that the computation graph is a DAG. We must ensure that, after executing the fusion plan, the resultant graph is also a DAG. However, the ILP solver alone does not enforce this property, as shown in Figure 3. In this simple graph (Figure 3(a)), nodes A and C are fused together, thus causing a cyclic dependence (Figure 3(b)).

We propose an iterative process to address this issue (Figure 3(d)). When the ILP solver generates a fusion plan, we check if there exists a cycle. If so, we identity the cycle, and enrich the solver with additional constraints (Figure 3(c)). We repeat the process until there is no cycle. The final fusion plan is used to transform the computation graph before kernel generation.

4.2 Domain Specific Pattern Generation

In this work, we consider fusion patterns consisting of *gemm*, *batched-gemm*, *reduction* and *elementwise* ops, with or without data dependences among them. Given a computation graph, a naive approach to generate fusion patterns is to iterate over the entire permutation space by evaluating each subgraph. This is too slow to operate in practice. We present two pattern search algorithms from DL domain specific perspective. The goal is to capture important fusion opportunities by traversing a modest subset of the search space.

4.2.1 Substitution Fusion

The substitution fusion targets graphs with tiny kernels, as shown in most models in Table 1. Algorithm 1 shows the procedure: it takes the *graph* and a *partition_ops* as input, and outputs fusion patterns. The *partition_ops* identifies ops that will never be fused with other ops. The basic idea is to minimize the number of kernels. To achieve this, we first topologically sort the graph and *partition_ops*. Then we iterate over the sorted list (*sorted_parts*), and collapse all ops between adjacent partition ops into a single fusion pattern.

For DL workloads, we use a multi-step heuristic to identify *partition_ops* and produce the full set of *fusion_patterns* for the ILP solver. First, we add all large *gemm* ops into *partition_ops*, and run Algorithm 1 to generate fusion patterns. Next, we extend *partition_ops* to further include *batched-gemm*, and again run the algorithm to obtain more fusion patterns. We then repeat this step to further include *column reductions*, *scalar reductions*, etc. Each time we extend the *partition_ops*, we run the algorithm to generate more fusion patterns, until *partition_ops* can not be extended anymore.

The motivation of this multi-step procedure stems from our observations in optimizing DL models. Given code generation capability, fusing multiple ops may or may not produce performance gains. Except *elementwise* or *row reduction* patterns, all others have dedicated requirements on parallelization strategies, on chip resource requirements, and launch constraints. These requirements may conflict with each other when aggressively composing them together.

The substitution fusion typically produces a minimum number of kernels. It also enjoys the property of being cycle free, because we never fuse across the partition op boundary. However, this may miss some interesting optimizations, such as the case shown in Figure 1. The exploratory fusion fixes this problem.

4.2.2 Exploratory Fusion

Unlike the substitution fusion, the exploratory fusion targets *elementwise*, *batched-gemm* or *reduction* patterns with large granularity. The objective is to minimize off chip memory accesses by composing as many data dependent ops as possible. Algorithm 2 shows the procedure. It is a recursive process taking the computation *graph* and *seed_pattern*, an

---

Algorithm 1 The Substitution Fusion Algorithm

Input: *graph*, *partition_ops*
Output: *fusion_patterns*
topo_order = TopologicalSort(graph)
sorted_parts = TopoSortPartOps(topo_order, partition_ops)
for *p* **in** *sorted_parts* **do**
    pattern = AllOpsUpToNextPartOp(p, topo_order)
    fusion_patterns.append(pattern)
end for

Algorithm 2 The Exploratory Fusion Algorithm

Input: *graph*, *seed_pattern*
Output: *fusion_patterns*
procedure EXPLORE(*graph*,*seed_pattern*)
    Initialize empty set: *candidates*
    ProducerExpansion(graph, seed_pattern, candidates)
    ConsumerExpansion(graph, seed_pattern, candidates)
    for *p* **in** *candidates* **do**
        fusion = Fuse(seed_pattern, p)
        fusion_patterns.append(fusion)
        EXPLORE(*graph*,*fusion*)
    end for
end procedure

---

initial set of ops for fusion, as input, and producing fusion patterns for the ILP solver. The producer (consumer) expansion examines all producers (consumers) of *seed_pattern* and put all fusible ops into *candidates*. We consider an *op* that can be fused into *seed_pattern* only if two conditions are both satisfied: (1) *op* must be a *elementwise*, *batched-gemm*, *reduction* op; (2) fusion of *op* into *seed_pattern* does not introduce any cyclic data dependence. After candidates expansion, we examine each op *p* in *candidates* by generating a new fusion pattern *fusion* by fusing *p* into *seed_pattern*. Then we explore starting from the expanded *fusion* recursively.

Naive application of the exploratory fusion risks exploring a huge search space. We mitigate this issue with careful selection of the initial *seed_pattern*. Specifically, we consider two heuristics. First, ops with large (> 10, for instance) number of operands are excluded, as Algorithm 2 explores all subsets of operands. Second, we only consider *elementwise*, *batched-gemm*, *reduction* ops with large input/output tensors.

In practice, we use the substitution fusion as a base strategy, and the exploratory algorithm as supplementary. If after applying these heuristics, the exploration still takes long time, we simply give up further exploration. While this may miss some marginal fusion possibilities. They are not significant to performance in all our workloads.

4.3 Fusion Pattern Evaluation

Fusion pattern evaluation measures the performance gain of the fused computation. It produces a real valued score for each pattern. A negative score means either we can not generate a kernel for the pattern, or the fused kernel performance is less than satisfactory. Thus only patterns with non-negative scores are fed into the ILP solver.

There are two techniques to evaluate fusion patterns. One is execution based. That is, we run the kernel generator to produce a kernel for the fused pattern, and measure its performance directly. Given a fusion pattern $P = \{Op_1, \ldots, Op_N\}$, the score is defined as follows:

$$f(P) = \sum_{j=1}^{N} K(Op_j) + (N - 1) * \phi - K(P)$$

Here $K(Op_j)$ and $K(P)$ denote kernel execution time of Op_j and the fusion pattern P, respectively. ϕ is the average launch latency (between $6us$ and $10us$) for a kernel. $f(P)$ measures the execution time saved after fusion. The above formula assumes the fused kernel has been generated successfully. If not so, we simply return a negative $f(P)$ value, and the fusion pattern is ignored.

The other approach is model based. Since we allow general composition of *gemms*, *batched-gemms*, *reductions* and *elementwise* ops, on chip shared memory is essential to transfer intermediate results (Section 5). We first check if the shared memory usage is satisfiable. If not, we simply return a negative score and ignore the pattern. Otherwise, we measure V, the volume (in Bytes) of input/output off chip memory accesses saved. Given fusion pattern $P = \{Op_1, \ldots, Op_N\}$, the score is thus defined as follows:

$$f(P) = M(V) + (N - 1) * \phi$$

Here, $M(V)$ models the extrapolated latency accessing V bytes of memory. We make two simplifications to enable fast calculation of $M(V)$. First, we assume consecutive access of V bytes of memory. Second, $M(V)$ depends on hardware memory bandwidth capability and the size of V. In order to avoid measuring $M(V)$ for each V, we use an memory bandwidth utilization model collected offline on the same hardware. We thus extrapolate $M(V)$ using the model, as shown in Figure 4.

The model based evaluation is fast, yet with less accuracy for complex fusion patterns. On the contrary, the execution based approach is accurate by generating, executing, and profiling the target kernel directly, but is time consuming. We adopt a combination of both. Specifically, we use model based approaches for most memory intensive patterns, but enable the execution based approach for complex ones. By complex, we mean domain specific heuristics, for instance fusion patterns with combinations of column/scalar and row reductions, gemm/batched-gemms and reductions, etc.
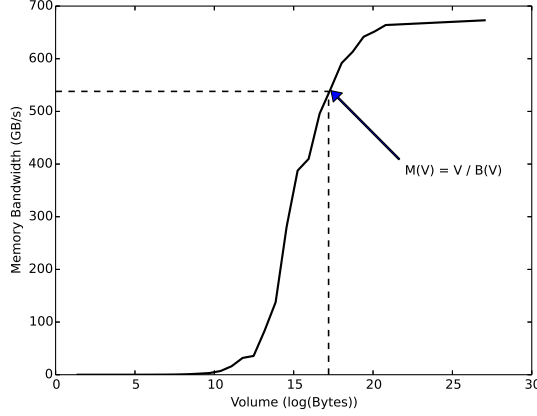
Figure 4. Memory Bandwidth Model

4.4 Implementation

We implement the fusion mechanism together with cost models as a code transformation pass in the XLA compilation framework in Tensorflow. As to the ILP solver, we use the publicly available Python *pulp* package.

5 Kernel Generation

5.1 Computation Composition

Given a fusion pattern, a key issue in kernel generation is how to compose computations of multiple ops into a fused kernel. Figure 5 summarizes four types of composition mechanisms supported in our work. For each composition, we show a computation example, the implementation template (Section 5.2), and the generated kernel sketch.

Kernel packing (Figure 5 (a)) packs computations of ops with no data dependences. Two observations of DL workloads motivate this. One is the existence of very fine grained ops as shown in most workloads in Table 2. Another is related to the backward phase of DL training. Groups of independent elementwise layers with similar or even identical shapes operate on large tensors for gradients accumulation. The kernel sketch includes two nested parallel loops, one for thread blocks and the other for threads within a block. To reduce control flow overheads, we perform aggressive loop fusion [14, 17] to merge as many elementwise ops as possible into a single loop structure. While the example only illustrates packing of elementwise ops, we also pack reduction and compute intensive ops as well. For DL workloads, kernel packing is instrumental in reducing loop control and kernel launch overheads.

Thread composition (Figure 5 (b)) fuses data dependent ops within a local thread context. Intermediate results are transferred via registers. For modern GPUs with large register files, this enables composition of many elementwise ops into large fused kernels. Warp composition (Figure 5 (c))

extends thread composition to fuse elementwise ops with a special form of reduction patterns. Such structures exist in common DL building blocks such as *softmax*, *batchnorm*, *layernorm* structures and their variants. Here we fuse the block loop, and employ warp reduction to enable register transfer of intermediate results of *reduce_1* to dependant elementwise ops (*mul_1*).

Block composition (Figure 5 (d)) is essential to compose *elementwise*, *reduction* and *gemm/batched-gemm* ops into a fused kernel. Unlike registers, we use on chip shared (*scratchpad* memory to transfer intermediate results. This is flexible, because we allow different ops (*reduce_1* and *dot_1*) to have independent parallel loops. Block composition unlocks the potential to enable composing non-homogenous computations into large fused kernels.

Previous works explored thread and block compositions in database[32], image processing[5, 16, 23], and HPC applications[18, 30]. The Tensorflow XLA[19] framework implements thread composition. To the best of our knowledge, this work is the first to investigate all four computation compositions thoroughly for DL workloads.

5.2 The Implementation Template

We propose the implementation template as a compact representation of concrete cuda kernels. Given a fusion pattern and such a template, the code generator synthesizes a kernel (Section 5.3). Following is the grammer for templates specification:

⟨*template*⟩	::=	⟨*schedule-list*⟩
⟨*schedule-list*⟩	::=	⟨*schedule*⟩ ⟨*schedule-list*⟩ | ⟨*schedule*⟩
⟨*schedule*⟩	::=	⟨*identifier*⟩ '[' ⟨*attr-list*⟩ ']' ';'
	|	⟨*identifier*⟩ '[' ⟨*attr-list*⟩ ']' 'S' ';'
⟨*attr-list*⟩	::=	⟨*attr*⟩ ',' ⟨*attr-list*⟩ | ⟨*attr*⟩
⟨*attr*⟩	::=	⟨*attrtype*⟩ | ⟨*subattr-list*⟩
⟨*subattr-list*⟩	::=	⟨*subattr*⟩ '-' ⟨*subattr-list*⟩ | ⟨*subattr*⟩
⟨*subattr*⟩	::=	⟨*attrtype*⟩ '_' ⟨*integer*⟩
⟨*attrtype*⟩	::=	'GRID' | 'WARP' | 'CTA' | 'THREAD'

The template consists of one or more schedules. A schedule denotes an *op* implementation that writes either shared (intermediate results used by other ops) or off-chip global memory (outputs of the entire fusion pattern). As an example, let's consider *reduce_1* in Figure 5 (d). In the schedule *[GRID,WARP,WARP,CTA]S*, there is a parallel loop tiling attribute for each input dimension. Here we perform block level parallelization on the first, warp level parallelization on the second and the third, and thread level parallelization on the last dimension. We also support multiple levels of tiling on the same dimension. For instance, in schedule *[GRID_128-WARP_2,WARP,WARP,CTA]S*, we perform both block and warp level tiling on the first dim. The *S* attribute instructs the kernel generator to cache results in shared memory.
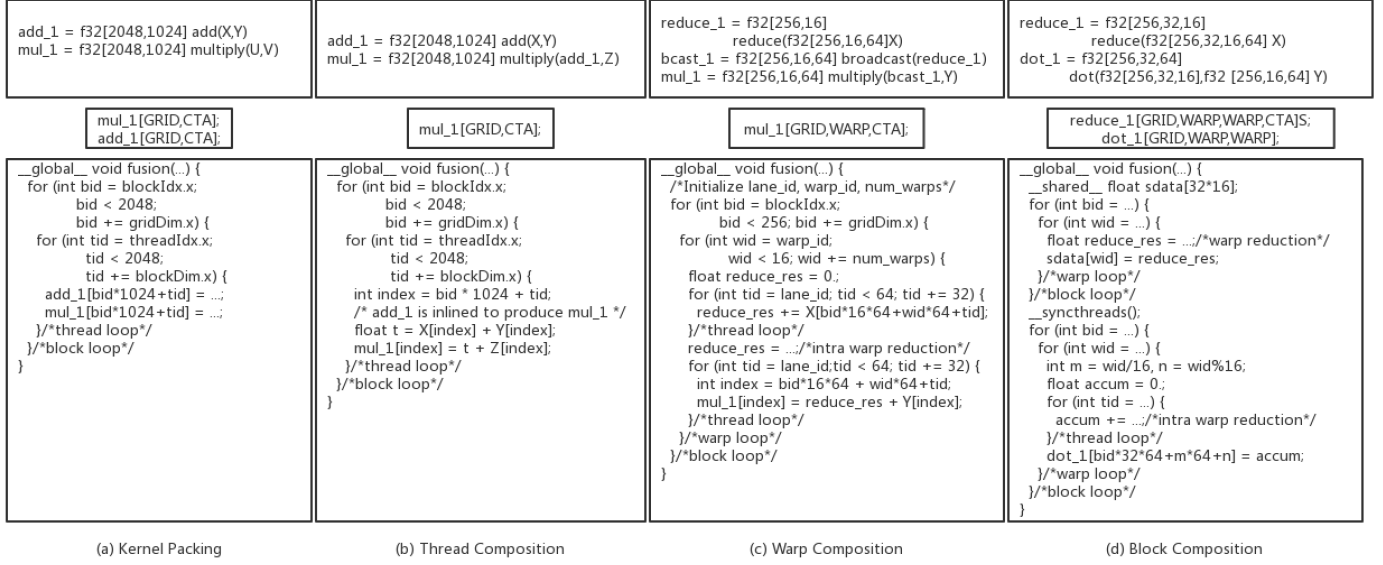
(a) Kernel Packing

```
add_1 = f32[2048,1024] add(X,Y)
mul_1 = f32[2048,1024] multiply(U,V)
```

```
mul_1[GRID,CTA];
add_1[GRID,CTA];
```

```
__global__ void fusion(…) {
  for (int bid = blockIdx.x;
       bid < 2048;
       bid += gridDim.x) {
    for (int tid = threadIdx.x;
         tid < 2048;
         tid += blockDim.x) {
      add_1[bid*1024+tid] = …;
      mul_1[bid*1024+tid] = …;
    }/*thread loop*/
  }/*block loop*/
}
```

(b) Thread Composition

```
add_1 = f32[2048,1024] add(X,Y)
mul_1 = f32[2048,1024] multiply(add_1,Z)
```

```
mul_1[GRID,CTA];
```

```
__global__ void fusion(…) {
  for (int bid = blockIdx.x;
       bid < 2048;
       bid += gridDim.x) {
    for (int tid = threadIdx.x;
         tid < 2048;
         tid += blockDim.x) {
      int index = bid * 1024 + tid;
      /* add_1 is inlined to produce mul_1 */
      float t = X[index] + Y[index];
      mul_1[index] = t + Z[index];
    }/*thread loop*/
  }/*block loop*/
}
```

(c) Warp Composition

```
reduce_1 = f32[256,16]
             reduce(f32[256,16,64]X)
bcast_1 = f32[256,16,64] broadcast(reduce_1)
mul_1 = f32[256,16,64] multiply(bcast_1,Y)
```

```
mul_1[GRID,WARP,CTA];
```

```
__global__ void fusion(…) {
  /*Initialize lane_id, warp_id, num_warps*/
  for (int bid = blockIdx.x;
       bid < 256; bid += gridDim.x) {
    for (int wid = warp_id;
         wid < 16; wid += num_warps) {
      float reduce_res = 0.;
      for (int tid = lane_id; tid < 64; tid += 32) {
        reduce_res += X[bid*16*64+wid*64+tid];
      }/*thread loop*/
      reduce_res = …;/*intra warp reduction*/
      for (int tid = lane_id;tid < 64; tid += 32) {
        int index = bid*16*64 + wid*64 +tid;
        mul_1[index] = reduce_res + Y[index];
      }/*thread loop*/
    }/*warp loop*/
  }/*block loop*/
}
```

(d) Block Composition

```
reduce_1 = f32[256,32,16]
             reduce(f32[256,32,16,64] X)
dot_1 = f32[256,32,64]
             dot(f32[256,32,16],f32 [256,16,64] Y)
```

```
reduce_1[GRID,WARP,WARP,CTA]S;
dot_1[GRID,WARP,WARP];
```

```
__global__ void fusion(…) {
  __shared__ float sdata[32*16];
  for (int bid = …) {
    for (int wid = …) {
      float reduce_res = …;/*warp reduction*/
      sdata[wid] = reduce_res;
    }/*warp loop*/
  }/*block loop*/
  __syncthreads();
  for (int bid = …) {
    for (int wid = …) {
      int m = wid/16, n = wid%16;
      float accum = 0.;
      for (int tid = …) {
        accum += …;/*intra warp reduction*/
      }/*thread loop*/
      dot_1[bid*32*64+m*64+n] = accum;
    }/*warp loop*/
  }/*block loop*/
}
```

Figure 5. Computation Composition

The *THREAD* attribute has no parallelization implications. For example, in schedule *[GRID,CTA,CTA,THREAD]*, each thread performs sequential reduction independently. This is useful when the reduction dim size is trivially small (not uncommon in practical workloads), applying this schedule essentially treats the reduction as elementwise, resulting in more efficient kernels in some cases.

We use templates to separate representation and implementation. Templates provide a compact tool to iterate over the kernel space and to expose performance critical tunable parameters. While current template design can not express arbitrary optimizations, especially for compute intensive ops, it is enough for expressing and tuning memory intensive patterns under study in this work.

5.3 Code Generation and Kernel Tuning

Algorithm 3 shows the procedure for kernel generation, evaluation and performance tuning. It takes as input a computation subgraph *fusion_pattern*, and outputs the optimized *best_kernel* after exploring and tuning a series of templates. In practical workloads, while the number of fusion kernels can be huge, there are only dozens of unique fusion patterns. It is only necessary to generate a kernel for each pattern once, and reuse it repetitively.

TemplatesGeneration produces a set of templates, each with different trade-offs among parallelization, on chip resource requirements and kernel launch settings. In JIT compilation, it is too time consuming to tune many templates. We address this issue either by employing a offline tuning procedure for complex patterns, or by reducing the templates to a small number with human expert knowledge. This is reasonable for repetitive DL workloads, since we can optimize once and run many times.

Algorithm 3 Kernel Generation

Input: $fusion\_pattern$
Output: $best\_kernel$
$templates = TemplatesGeneration(fusion\_pattern)$
for $template$ **in** $templates$ **do**
　Initialize: $kernel$
　$RegisterPlanning(template, fusion\_pattern)$
　if $!SharedPlanning(template, fusion\_pattern)$ **then**
　　$continue$
　end if
　for $schedule$ **in** $template.schedulelist$ **do**
　　$clusure = SchedClusure(schedule, fusion\_pattern)$
　　if $ReductionSchedule(schedule)$ **then**
　　　$ReductionGen(closure, kernel)$
　　else if $DotSchedule(schedule)$ **then**
　　　$DotGen(closure, kernel)$
　　else
　　　$ElemwiseGen(closure, kernel)$
　　end if
　end for
　$KernelEvalUpdate(kernel, best\_kernel)$
end for

RegisterPlanning targets *elementwise* ops whose results are shared by multiple data dependent ops. We maintain a list of all such ops in thread local context, avoiding generating code repetitively for them.

SharedPlanning optimizes shared memory usage. Either *elementwise*, *reduction*, or *batched-gemm/gemm* ops have reasonable use cases. However, there are two constraints. The first is the volume constraint: total shared space allocated should not exceed an upper threshold T. We discuss shared

space optimization in Section 5.4. The second is layout constraint: we must ensure that for each *op* with shared allocation, the shared space is only accessed within a single thread block context. Only when both the volume and layout constraints satisfied can kernel be generated successfully.

To generate a kernel for a *template*, we traverse its *schedulelist*, and generate a code piece for each schedule. Depending on the schedule type we call separate code emitters. For *elementwise* schedule, a parallel loop is emitted with thread composition (Figure 5 (a)), with special care upon register level results sharing. For *reduction* schedule, either thread, warp, or block composition is selected depending on schedule parameters. For *gemm/batched-gemm* schedule, we also support either thread or block compositions, depending on the reduction dimension size. However, we avoid complex tiling loop transformations since our primary focus here is optimizing memory intensive patterns, not FLOP efficiency.

Each time we generate a kernel. We call *KernelEvalUpdate* to evaluate its performance and update *best_kernel* if necessary.

5.4 Shared Memory Optimization

Shared memory is the key to compose large granularity of computations effectively in a kernel. Algorithm 4 outlines a lightweight shared space optimization scheme. Since fusion patterns in our context are typically large, our goal is not to save space in general, but to constrain worst case shared memory usage. An example is shown in Figure 1. Both *dot_1* and *add* need $94 * 94 * 4$ bytes of shared space. In this graph, the *add* can reuse the space allocated for the *dot_1*. Without optimization, the hardware occupancy is too low thus compromising performance of the kernel.

A key building block is the dominance tree algorithm[11]. Instead of using it for control flow analysis, we divert its use here in data flow graphs. The algorithm takes a computation graph and shared memory requests (*req_map*) as input, and outputs an allocation map (*alloc_map*). To optimize shared space sharing, we traverse ops of the computation graph in topological order. For each op, if no shared space is needed, we simply propagate allocation information from all its operands. This propagation along data flow edges, together with the dominance relation of the graph, is vital for shared space sharing. Otherwise, if it need shared space for on chip intermediate results transfer, we merge allocations of all its operands (*CollectAllocInfo*), test the dominance relation to check if we can share any pre-allocated space for current op, and reuse the space if possible.

5.5 Implementation

We implement templates generation, enumeration, and kernel construction by replacing the XLA *IREmitter* framework. In particular, we propose and implement *CUDAEmitter*, a backend to generate CUDA-C code directly instead of LLVM

Algorithm 4 Shared Memory Optimization

Input: $fusion\_pattern$, req_map
Output: $alloc\_map$
$dom = BuildDominanceTree(fusion\_pattern)$
$topo\_order = TopologicalSort(fusion\_pattern)$
for $inst$ **in** $topo\_order$ **do**
    if $req\_map.count(inst)$ **then**
        Initialize: $prev\_allocs$
        for $operand$ **in** $inst.operands()$ **do**
            $CollectAllocInfo(operand, prev\_allocs)$
        end for
        Initialize: $shared = False$
        for $prev\_inst$ **in** $prev\_allocs$ **do**
            if $dom.Dominates(inst, prev\_inst)$ **then**
                if $!shared$ **then**
                    $Share(inst, prev\_inst, prev\_allocs)$
                    $shared = True$
                    $continue$
                end if
                $Reclaim(prev\_inst)$
            end if
        end for
        if $!shared$ **then**
            $Alloc(inst, alloc\_map)$
        end if
    else
        for $operand$ **in** $inst.operands()$ **do**
            $PropagateAllocInfo(inst, operand)$
        end for
    end if
end for

IR. We compile CUDA-C with NVCC and integrate the resultant CUBIN binary with the rest of XLA runtime execution engine to power DL workloads.

6 Evaluation
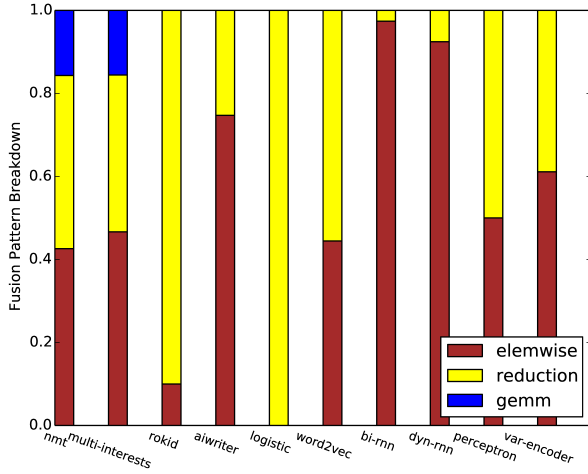
6.1 Experimental Setup

We evaluate the fusion optimization and kernel generation mechanisms on four practical models are six micro-benchmarks. Table 1 and Table 2 summarize workload characteristics. We use the default Tensorflow implementation without compilation, as well as the XLA compiled version as baselines for our performance comparison study. All evaluation results are collected on Nvidia V100 GPUs.

6.2 Kernel Launch Savings

A direct result of computation fusion is reduced number of kernels. We measure this with the kernel number compression ratio. Table 3 shows the results. The column *xla/tf-kernel* shows the kernel number compression ratio of *xla*,

Table 3. Summary of Evaluation Results

Name	xla/tf-kernel	fs/tf-kernel	xla/tf-perf	fs/tf-perf
nmt	1.65	3.25	1.36	1.84
multi-interests	1.63	2.27	1.25	1.57
rokid	1.3	13.5	1.09	1.35x
aiwriter	2.6	5.7	2.53	4.6
logistic	1.5	4.6	1.46	1.83
word2vec	2.9	4.9	2.6	3.3
bi-rnn	2.8	6.8	2.5	4.1
dyn-rnn	3.2	7.8	3.8	5.7
perceptron	1.1	1.3	0.94	1.18
var-encoder	1.5	1.9	0.8	1.09

Figure 6. Fusion Pattern Composition

Figure 7. Normalized Kernel Performance

Figure 8. Kernel Performance Breakdown

with respect to the kernel number of the Tensorflow baseline. Similarly, the *fs/tf-kernel* column shows the kernel number compression ratio of our work. Compared to *xla*, the kernel number compression ratio of our approach is 1.18x to 10.38x higher, with 2.9x on average. This is not surprising because we have relaxed fusion conditions by allowing packing of computations with no data dependences and fusion of *gemm/batched-gemm* with *elementwise* and *reduction* patterns as well.

6.3 Performance Speedups

Larger fusion granularity comes with increased kernel generation complexity. It is important that aggressive fusion does not compromise GPU efficiency. The last two columns of Table 3 show performance results. Compared to Tensorflow, our approach achieves 1.09x up to 5.7x speedups, with 2.6x on average. Compared to *xla*, our approach achieves 1.24x up to 1.84x speedups, with 1.4x on average.

6.4 Fusion Patterns Analysis

We classify a fusion pattern to be one of three categories. A fusion pattern is *elemwise*, if neither *reduce* nor *gemm*

ops exist in its computation. Otherwise if it contains *gemm* ops, it is marked as *gemm* pattern. The rest are all marked as *reduction* patterns. Note if a pattern contains both *gemm* and *reduce*, it is marked as *gemm* instead of *reduction*. Figure 6 shows fusion pattern composition for all benchmarks. While *elementwise* and *reduction* are common in all workloads, each workload have different pattern distributions. Both *nmt* and *multi-interests* have 16% *gemm* patterns.

6.5 Kernel Performance Analysis

In order to study the kernel performance of our approach. We collect execution time of all kernels with the *Nvprof* tool for four industry workloads. These models reflect common requirements of user customized model structures, which offer notable potential for large granularity fusion and kernel optimizations.

Figure 7 shows the accumulated kernel execution time of all kernels, normalized to *xla* baseline, for all applications. On average our approach achieves 39% kernel execution time

Table 4. Shared Memory Statistics

Name	pt-ratio	exe-ratio	shd-avg	max-shd	alloc/req
nmt	42%	87%	3.6KB	18.6KB	0.81
multiinterests	24%	56%	7.5KB	36KB	0.88
rokid	90%	95%	5.5KB	16.4KB	0.98
aiwriter	34%	62%	1.4KB	4.1KB	0.87

reduction, illustrating the effectiveness of fusion and kernel generation towards large granularity. To further study which fusion patterns contribute most to kernel performance, Figure 8 shows the execution time breakdown on three categories of patterns. Collectively *elemwise* patterns contribute approximately half of the execution time. The other half is contributed by *reduction* and *gemm* patterns together. Please note that *aiwriter* and *rokid* have *gemm* layers but no *batch-gemm* ops. But the fusion engine does not fuse them because the granularity is large enough, that calling *cuBLAS* routines is more efficient than generating kernels directly. Unlike *aiwriter* and *rokid*, *nmt* and *multi-interests* have both *gemm* and *batch-gemm* ops with workload specific shapes which imply interesting fusion opportunities. Our fusion engine handles these cases gracefully.

6.6 Shared Memory Analysis

As discussed extensively in Section 4 and Section 5, on chip shared memory is critical when composing numerous computations in a unified kernel. Table 4 shows statistics for four applications. The *pt-ratio* column shows the percentage of fusion patterns that need shared memory. The *exe-ratio* column shows the percentage of kernel execution time to which such fusion patterns contribute. As can be seen, in all applications, shared memory usage is critical for fusion and kernel generation.

The shared memory is a precious resource in modern GPUs. How much shared space fusion patterns actually use? Columns *avg-shd* and *max-shd* show average shared size allocated across all patterns, and the maximum size required, respectively. Except *multi-interests*, all other applications have modest shared memory requirements.

Shared space can be shared among multiple ops in a fused computation. The last column *alloc/req* shows the ratio between allocated size and the total space requested. The lower the ratio, the higher the shared memory reuse within the kernel. As can be seen, the reuse degree is relatively low (high *alloc/req* ratio). This seems reasonable, because we not only fuse *reduction* and *gemm* ops with *elementwise* patterns, but also do packing of computations with no data dependences. While the averge reuse degree is low, shared space sharing is critical to reduce worst case shared memory usage. For instance, without sharing, most demanding fusion patterns of *multi-interests* request up to 72KB of shared space.

This results in very low execution occupancy, compromising all merits of fusion.

7 Related Work

GPU kernel fusion, inspired from classical loop optimizations[13, 14, 17], is known to boost performance in other application domains. In database domain, *KernelWeaver*[32] proposed transformations to fuse execution of multiple operators into a single kernel. This work provided support for both thread and block (CTA) composition of operators, yet with little support for tuning of implementation schedules. In the HPC domain, [30] formulated GPU kernel fusion as an combinatorial search problem, and searched the solution space for an optimized fused kernel. In image processing domain, [22, 23] formulated the image pipeline fusion as a graph cut problem. For machine learning workloads, [7] proposed a kernel fusion technique to generate efficient kernels for a specific computation pattern. The *XLA* compilation framework[19] can handle more general computation pattern, but offers only basic capability for fusion and kernel generation. However, XLA relies on empirical rules to encode fusion opportunities, and does not support fusion and code generation of composition of *elementwise*, *reduction* and *gemm* patterns.

The separation of optimization specification and implementation is important for performance modeling and tuning. There are extensive works for compute intensive operators, such as Halide[16], TVM[1, 10], and TensorComprehensions[29]. The implementation template in our work targets large, complex memory intensive computation patterns.

There are recent advances on code generation of compute intensive DNN layers. [6] proposed a solution for selecting fast kernel implementations in the global context by formulating it as a PBQP problem. Boda[20] is a code generator that generates code for CNN layers on mobile platforms. Latte[28] is a DSL system for DNN allowing users to specify, synthesize and optimize code for NN layers. SLINGEN[26] is another DSL system which takes mathematical specifications and generates optimized C functions for linear algebra operators with small input sizes. These research are relevant but complementary to our work.

8 Conclusion

Fine grained memory intensive computations are abundant in DL workloads. This work tackles this problem from two aspects. First, we propose a novel computation fusion framework to explore and optimize fusion plans. In particular, we propose an ILP formulation for fusion plans optimizations. Our fusion framework not only supports composition of *elementwise* and *reduction* ops with or without data dependences, but also support composition of *gemm/batched-gemm* ops, thus enabling collective optimizations of both compute and memory intensive computations.

Together with fusion plan optimizations, we propose a code generation algorithm to produce optimized kernels for GPUs. With extensive intermediate results sharing via either registers or shared memory, our work is capable of supporting very large fusion granularity. Experimental results on six benchmarks and four industry scale practical models are encouraging. Overall, *FusionStitching* can reach up to 5.7x speedup compared to Tensorflow baseline, and achieves 1.25x to 1.85x performance speedups compared to current state of the art, with 1.4x on average (geometric mean).

References

[1] [n. d.]. TVM: Open Deep Learning Compiler Stack. https://github.com/dmlc/tvm

[2] [n. d.]. XLA Operation Semantics. https://www.tensorflow.org/xla/operation_semantics

[3] 2015. Torch NN. https://github.com/torch/nn

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[5] A. M. Adnan, S. Radhakrishnan, and S. Karabuk. 2015. Efficient Kernel Fusion Techniques for Massive Video Data Analysis on GPGPUs. arXiv:cs.PL/1509.04394 https://arxiv.org/abs/1509.04394

[6] A. Anderson and D. Gregg. 2018. Optimal DNN Primitive Selection with Partitioned Boolean Quadratic Programming. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. Vienna, Austria.

[7] A. Ashari, S. Tatikonda, K. Campbell, and P. Sadayappan. 2015. On Optimizing Machine Learning Workloads via Kernel Fusion. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. San Francisco, CA, USA.

[8] aymericdamien. [n. d.]. Tensorflow-Examples. https://github.com/aymericdamien/Tensorflow-Examples

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015).

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of Operating Systems Design and Implemention (OSDI)*.

[11] K. D. Cooper, T. J. Harvey, and K. Kennedy. [n. d.]. A Simple, Fast Dominance Algorithm.

[12] P. Covington, J. Adams, and E. Sargin. [n. d.].

[13] C. Ding and K. Kennedy. 2004. âĂĲImproving Effective Bandwidth Through Compiler Enhancement of Global Cache Reuse. *J. Parallel and Distrib. Comput.* 64 (2004), 108–134.

[14] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. 1993. Collective Loop Fusion for Array Contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*. London, UK.

[15] K. M. He, X. Y. Zhang, S. Q. Ren, and J. Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).

[16] J. R. Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* Volume 61 Issue 1 (2018).

[17] K. Kennedy and J. R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[18] M. Korch and T. Werner. 2018. Accelerating Explicit ODE Methods on GPUs by Kernel Fusion. *Concurrency and Computation: Practice and Experience* (2018).

[19] C. Leary and T. Wang. [n. d.]. XLA: Tensorflow, Compiled.

[20] M. W. Moskewicz, A. Jannesari, and K. Keutzer. 2017. Boda: A Holistic Approach for Implementing Neural Network Computations. In *Proceedings of the Computing Frontiers Conference (CF'17)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/3075564.3077382

[21] Nvidia. [n. d.]. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[22] B. Qiao, O. Reiche, F. Hannig, and J. Teich. 2018. Automatic Kernel Fusion for Image Processing DSLs. In *Proceedings of International Workshop on Software and Compilers for Embedded Systems*. St. Goar, Germany.

[23] B. Qiao, O. Reiche, F. Hannig, and J. Teich. 2019. From Loop Fusion to Kernel Fusion: A Domain Specific Approach to Locality Optimization. In *Proceedings of International Symbopium on Code Generation and Optimization (CGO)*.

[24] K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[25] M. Slvathanu, T. Chugh, S. S. Singapuram, and L. D. Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 909–923.

[26] D. G. Spampinato, D. F. Traver, P. Bientinesi, and M. PÃijschel. 2018. Program Generation for Small-Scale Linear Algebra Applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. Vienna, Austria.

[27] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR* abs/1512.00567 (2015). arXiv:1512.00567 http://arxiv.org/abs/1512.00567

[28] L. Truong, R. Barik, E. Totoni, H. Liu, C. Markley, A. Fox, and T. Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 209–223. https://doi.org/10.1145/2908080.2908105

[29] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:cs.PL/1802.04730 https://arxiv.org/abs/1802.04730

[30] M. Wahib and N. Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *Proceedings of SC'14*. New Orleans, LA, USA.

[31] J. Z. Wang, P. P. Huang, H. Zhao, Z. B. Zhang, B. Q. Zhao, and D. L. Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. London, United Kingdom.

[32] H. C. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *Proceedings of 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Vancouver, BC, Canada.

[33] D. Y. Xiong, J. H. Li, A. Branco, S. H. Kuang, and W. H. Luo. 2018. Attention Focusing for Neural Machine Translation by Bridging Source and Target Embeddings. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers.* 1767–1776. https://aclanthology.info/papers/P18-1164/p18-1164

[34] Z. Zheng, C. Y. Oh, J. D. Zhai, X. P. Shen, and W. G. Chen. 2017. VersaPipe: A Versatile Programming Framework for Pipelined Computing on GPU. In *Proceedings of 45th Annual IEEE/ACM International Symposium on Microarchitecture.* Boston, MA, USA.