AStitch: Enabling a New Multi-dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures

Zhen Zheng*
Alibaba Group
China

Xuanda Yang
Alibaba Group
China

Pengzhan Zhao
Alibaba Group
China

Guoping Long
Alibaba Group
China

Kai Zhu
Alibaba Group
China

Feiwen Zhu
Alibaba Group
China

Wenyi Zhao
Alibaba Group
China

Xiaoyong Liu
Alibaba Group
China

Jun Yang
Alibaba Group
China

Jidong Zhai
Tsinghua University
China

Shuaiwen Leon Song
University of Sydney
Australia

Wei Lin
Alibaba Group
China

ABSTRACT

This work reveals that memory-intensive computation is a rising performance-critical factor in recent machine learning models. Due to a unique set of new challenges, existing ML optimizing compilers cannot perform efficient fusion under complex two-level dependencies combined with just-in-time demand. They face the dilemma of either performing costly fusion due to heavy redundant computation, or skipping fusion which results in massive number of kernels. Furthermore, they often suffer from low parallelism due to the lack of support for real-world production workloads with irregular tensor shapes. To address these rising challenges, we propose *AStitch*, a machine learning optimizing compiler that opens a new multi-dimensional optimization space for memory-intensive ML computations. It systematically abstracts four operator-stitching schemes while considering multi-dimensional optimization objectives, tackles complex computation graph dependencies with novel hierarchical data reuse, and efficiently processes various tensor shapes via adaptive thread mapping. Finally, *AStitch* provides just-in-time support incorporating our proposed optimizations for both ML training and inference. Although *AStitch* serves as a stand-alone compiler engine that is portable to any version of TensorFlow, its basic ideas can be generally applied to other ML frameworks and optimization compilers. Experimental results show that *AStitch* can achieve an average of 1.84× speedup (up to 2.73×) over the state-of-the-art Google's XLA solution across five production workloads. We also deploy *AStitch* onto a production cluster for ML workloads with thousands of GPUs. The system has been in operation for more than 10 months and saves about 20,000 GPU hours for 70,000 tasks per week.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Parallel computing methodologies**.

KEYWORDS

Machine Learning, Memory-Intensive Computation, Compiler Optimization, Fusion

ACM Reference Format:
Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: Enabling a New Multi-dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507723

1 INTRODUCTION

Machine learning models usually consist of two types of operations: *compute-intensive* operations and *memory-intensive* operations. Compute-intensive operations are typically composed of heavy computation kernels (e.g., GEMM/GEMV and Convolution), while memory-intensive operations are often bounded by memory bandwidth (e.g., element-wise and reduction operations). Many recent works [14, 16, 18, 41, 43, 55] have made significant efforts on optimizing compute-intensive operations since they dominate the execution of some DNN workloads, especially in the domains of computer vision (CV, such as image classification [29], segmentation[27]

*Email: james.zz@alibaba-inc.com

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507723

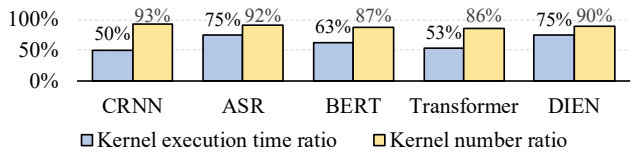


Figure 1: Ratio of memory-intensive computations. The ratio is the proportion of memory-intensive ops' metrics to those of all GPU kernels under study. Statistics on execution time and kernel count are collected from TensorFlow (v1.15) execution.

et al.). However, the recent advancement of machine learning has resulted in many novel model structures (e.g., Attention[44]) in which memory-intensive operations account for an even more significant portion of execution time than their compute-intensive counterparts.

Figure.1 shows the ratio of memory-intensive computation for five representative models used in real-life production on NVIDIA V100 GPU, including NLP (BERT[21], Transformer[44]), recommendation (DIEN[58]) and speech/character recognition (ASR[47], CRNN[40]). Note that GPU is one of the most widely-adopted accelerators for performing machine learning training and inference today. With an average ratio of 63.2% in execution time and 89.6% in total kernel numbers, memory-intensive computation has already become a dominating factor that significantly impacts the training/inference efficiency of many recent DNN workloads. Moreover, the ratio of computing power to memory bandwidth has also drastically increased on the recent generations of GPU architectures, e.g., a 5.6× increase from NVIDIA V100 to A100 (note that A100 uses TF32 as the default data type). As a result, for example, the average portion of the execution time contributed by the memory-intensive operations from the five models of Figure.1 increases to as high as 76.7% on A100. Therefore, effectively optimizing memory-intensive computations in today's DNN workloads becomes increasingly crucial and urgent.

Memory-intensive computations usually form a set of subgraphs, which are divided by compute-intensive operators in a machine learning computation graph. The overhead caused by memory-intensive computations mainly comes from intensive off-chip memory access, severe CPU-GPU context switch and high framework scheduling cost due to the large amount of kernels required to be launched and executed. Although the traditional kernel fusion techniques [37, 38, 45, 49] can help partially address these issues by putting memory-intensive subgraphs into one GPU kernel, they cannot fully meet the demand of current machine learning optimizations in which there are various ML model structures and innumerable customized variants, requiring just-in-time (JIT) optimizations for a given arbitrary model structure rather than ad-hoc solutions.

In recent years, there have been several ML compilers [11, 18] supporting kernel fusion for general memory-intensive ops to reduce off-chip memory access, CPU-GPU context switching and framework-level operator scheduling overhead induced by frequent kernel launching. However, a unique set of new challenges emerge from executing these memory-intensive ML models in production.

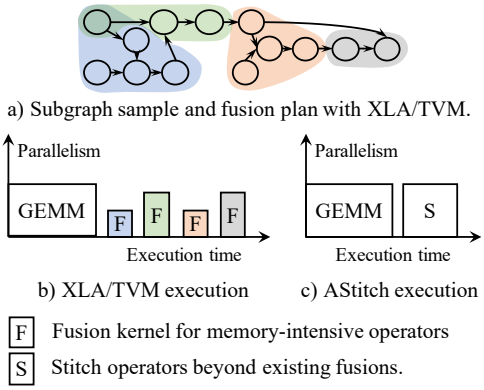


Figure 2: Conceptual illustration of how *AStitch* outperforms XLA and TVM for processing memory-intensive subgraphs. *AStitch* stitches large scope of memory-intensive operators together to reduce non-computation overhead and increase parallelism.

First, complex two-level dependencies combined with just-in-time demand exacerbates training/inference inefficiency (Sec.2.3.1). Constrained by this challenge, state-of-the-art ML compilers face the dilemma of executing costly fusion under heavy redundant computation, or skipping fusion which in turn generates massive number of kernels. *Second*, irregular tensor shapes in real-world production workloads often lead to poor parallelism control and severe performance issues in the current ML compilers (Sec.2.3.2).

To address these limitations, we propose *AStitch*, a machine learning optimizing compiler that opens a new multi-dimensional optimization space for memory-intensive ML computations by supporting efficient just-in-time operator stitching for arbitrary memory-intensive subgraphs. It provides a JIT-based joint optimization of dependency characteristics, memory hierarchy (locality) and parallelism. Specifically, we propose *hierarchical data reuse* technique (Sec.3.2) to address the complex two-level dependencies and enlarge fusion scopes, avoiding the dilemma of choosing between fusion with high computation overhead and inadequate fusion. An *adaptive thread mapping* technique (Sec.3.3) is also proposed to adapt different input tensor shapes and generate proper thread mapping schedules for maximizing hardware utilization and parallelism. Finally, we make several key design observations (Sec.4), and by leveraging them we design a compiler to enable the proposed optimizations automatically. We use "stitching" in this paper to differ our advanced fusion techniques from the existing ML compiler fusion approaches: our expansion of the current fusion scope is to "stitch" many small and basic fusions enabled by the current works into much larger and broader fusions. Figure.2 illustrates how *AStitch* outperforms XLA and TVM conceptually. We evaluate *AStitch* on a set of common machine learning models in production. *AStitch* achieves up to 2.73× speedup for inference over the state-of-the-art solutions. In summary, this work makes the following contributions:

- It reveals that memory-intensive computation is a rising performance-critical factor in recent non computer vision machine learning models.

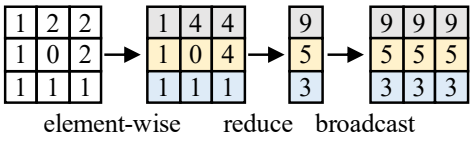


Figure 3: Typical memory-intensive operations.

- It tackles two major performance issues of memory-intensive ML computations, i.e., inefficient fusion and inputs with irregular tensor shapes.
- It is the first work to thoroughly investigate how to optimize memory-intensive ML computations from a joint aspect of dependency characteristics, memory hierarchy and parallelism.
- It designs a compiler to enable the proposed optimizations just-in-time for any given arbitrary machine learning model for both training and inference.
- It implements a production-ready ML compiler that is portable to any version of TensorFlow. Users do not need to modify source code or tune code generation schedule to use our compiler.

2 BACKGROUND AND CURRENT CHALLENGES

In this work, we focus our discussion on significantly improving training and inference efficiency of memory-intensive ML applications on the most widely-adopted general-purpose AI accelerators[6]: GPUs. NVIDIA chips (and its terminologies) are used in this paper as our validation platforms; however, the proposed techniques are general and applicable to other GPU architectures[1].

2.1 Essential Memory-Intensive Ops in Current Models

A machine learning workload is usually represented as a computation graph in modern frameworks [12, 36]. Most of the compute-intensive operators in the graph are disconnected. They divide the graph into a set of subgraphs, each with a set of tens or even hundreds of memory intensive operators. There are two types of widely-adopted operators that cover the majority of the memory-intensive computations in modern machine learning models: *element-wise ops* and *reduce ops*.

As shown in Figure.3, the elements in an element-wise op are processed independently in an element-wise manner. Element-wise ops can be further classified into *light* element-wise ops and *heavy* element-wise ops. The former executes lighter computations like *add* and *sub*, while the latter executes significantly more expensive computations (e.g., *tanh*, *power*, and *log*). *Broadcast* is often regarded as element-wise.

A reduce op takes a tensor as input and reduces its one or more dimensions. A reduce is referred as *row-reduce* if it reduces on a dimension where elements are continuous in memory; otherwise, *column-reduce*. *Row-reduce* for one row is usually organized within a GPU thread block, in which adjacent threads read continuous memory addresses for efficiency. *Column-reduce* is applied to reduce

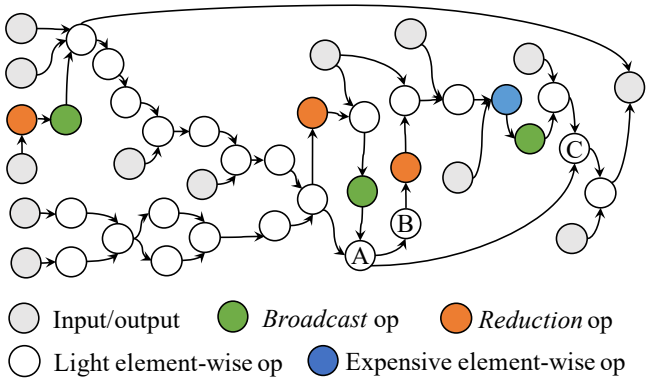


Figure 4: A typical subgraph in a Transformer model.

discontinuous elements, often requiring several thread blocks with atomic operations to ensure its correctness.

Due to the high frequency of *reduce* and *broadcast* applied in modern machine learning computation graphs, tensor shapes between operators become increasingly diverse. For instance, the *Transformer* model in Figure.1 contains 1,666 *reduce* operators which counts for approximately 10% of the total computation operators. These reduce operators can form arbitrary graph typologies that result in complex dependencies.

2.2 Memory-Intensive Op Fusion

State-of-the-art works (e.g., XLA[11] and TVM[18]) support kernel fusion of general memory-intensive ops to reduce off-chip memory access, CPU-GPU context switching and framework-level operator scheduling overhead induced by frequent kernel launching.

One of the most fundamental factors of fusion is code generation ability. ML compilers often make fusion decisions (e.g., pattern matching process in some studies) according to whether they can generate efficient code. For example, TVM/XLA's code generators deal with all data dependencies with per-element input inline to merge producer with consumer together. However, we have identified a set of limitations for such code generation approach in Sec.2.3. Simply modifying the fusion decision logic (e.g., enlarging fusion scope) may lead to poor performance for TVM/XLA.

In this work, we address the fundamental code generation problem rather than the surface-level pattern matching problem.

2.3 Major Limitations of the State-Of-The-Arts

There is a unique set of new challenges emerged from executing these memory-intensive ML models in production environment. We further elaborate them as follows.

2.3.1 Challenge I: Complex Two-Level Dependencies Combined With Just-In-Time Demand Exacerbates Training/Inference Inefficiency. A memory-intensive sub-graph usually consists of tens or even hundreds of operators. There are *two levels of dependencies*. **Operator-level** dependency describes operator connection represented in a subgraph, e.g., operator *B* and *C* depends on operator *A* in Figure.4. **Element-level** dependency indicates the dependency between elements within tensors, such as the element 9 depends on elements

1, 4, 4 for *reduce* in Figure.3. At *operator level*, operators tend to form complex connections in these memory-intensive subgraphs. At *element level*, during tensor processing, the frequent occurrence of *reduce* and *broadcast* may also incur many one-to-many dependencies. Furthermore, there are various machine learning model structures and innumerable customized variants, demanding just-in-time (JIT) optimizations for a given arbitrary model structure rather than ad-hoc solutions. In other words, unlike kernel fusion schemes used in traditional domains like HPC [45], database[49], and image processing[37, 38] in which static workloads are often the targets for optimization, ML practitioners are typically required to customize model structures via frequent tuning and execution, which demands automatic optimization at each running (i.e., just-in-time) rather than hand-tuned fusion before each trial. This also increases the overall complexity of the design optimization. Thus, the two-level dependencies combined with JIT demand makes fusion optimization extremely challenging for modern memory-intensive ML models.

Constrained by these unique challenges, we make **a key observation** that current machine learning optimization compilers (e.g., XLA[11], TVM[18]) cannot perform efficient fusion under such two-level dependencies. Specifically, for operators with complex dependencies, these optimization compilers either simply skip necessary fusions, or incurs redundant computation after fusion. We dissect the root causes behind these limitations as follows.

Key Inefficiency: large number of kernels generated by the ineffective fusion strategies for memory-intensive subgraphs.

State-of-the-art compilers (e.g., XLA and TVM) cannot efficiently fuse two common memory-intensive patterns due to the inability to deal with one-to-many element-level dependencies: (1) reduce ops with its consumers (e.g., orange circles in Figure.4), and (2) costly element-wise ops followed by *broadcast* ops (e.g., blue and green circles in Figure.4). We find that the current ML compiler frameworks face the following dilemma when conducting fusion on these two patterns:

(i) *Fuse? Heavy redundant computation.* If fusion is performed for the ops in these two patterns, we observe that neither XLA nor TVM communicates intermediate results between threads; they only leverage per-thread registers to fuse ops in the compiler. When there are one-to-many element-level dependencies, where one element generated by the producer is required by multiple elements of the consumer(s), each thread of the consumer will independently compute this common element, causing significant computation redundancy. Figure.5 illustrates such redundant computations when TVM fuses ops *power<2> - broadcast<2,128> - add<2,128>*[1] together. Here we only show one operand of *add* op to simplify the demonstration. In this case, every 128 elements of *add* require one element produced by *power*. However, *power* will recompute 128 times for the same value in 128 different threads because the compiler cannot cope with the one-to-many dependency effectively with its automatic strategy. *Power* is an expensive element-wise op and requires a large number of cycles to produce data.When the tensor shape is large, it requires several waves of threads, causing notable waste of GPU resources caused by redundant computation of *power*. Additionally, when fusing reduce ops with its consumer

[1] *op*<m,n> represents operator *op* processing a tensor with shape [m,n]

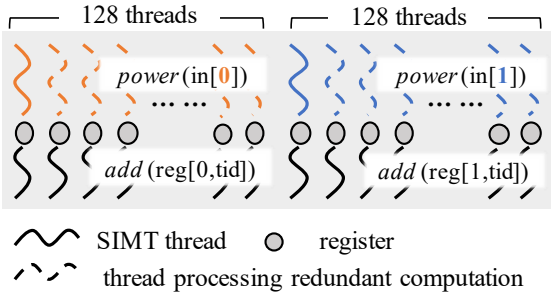


Figure 5: Redundant computation in TVM when attempting to fuse *power<2> - broadcast<2,128> - add<2,128>*[1] automatically with compiler. Different colors for *power* represent threads that process different elements in the input tensor.

(e.g., pattern(1)), the redundant computation will become more severe as each thread of the consumer needs to recompute the whole reduction independently. Note that *reduce* op itself is time consuming and is typically followed by broadcast ops which may cause one-to-many dependencies.

Note that compiler optimization is quite different from hard-coded optimization. As for Figure.5, GPU experts may manually buffer the results of *power* on shared memory within thread-block to enable reuse for *add*. However, giving compiler a subgraph (e.g., Figure.4) with frequent element-level one-to-many dependencies and diverged operator connections (operator dependency), it is quite tricky to decide how to organize intermediate data for reuse automatically given the complexity of the dependencies and underlying hardware. Meanwhile, compilers also care about parallelism of the operators, which further affects data locality. Thus, the state-of-the-art ML compiler frameworks, if deciding to fuse the operators from the two patterns, run into the heavy redundant computation issue discussed above due to the lack of consideration for a joint automatic optimization of complex dependency characteristics, parallelism and locality.

(ii) *Skipping fusion? More kernels are generated for execution.* To avoid these redundant computations, state-of-the-art designs tend to skip fusion when encountering one-to-many dependencies from the two patterns. For example, XLA skips fusion when encountering pattern (1) and (2) above, resulting in a large number of kernels for memory-intensive operators. For the *Transformer* model, XLA generates around 3 more times of kernels for memory-intensive computations than the corresponding compute-intensive computations. As discussed in Sec.2.1, since effective fusion should merge all the memory-intensive operators between two the compute-intensive operators to avoid unnecessary kernel launching overhead and off-chip memory access, ideally the number of kernels from the two types of computations should be roughly the same. On the other hand, TVM also skips fusion upon *reduce* ops (i.e., pattern(1)), but it continues to fuse for pattern (2). Although this may significantly reduce the number of generated kernels, it also introduces large redundant computation overhead discussed above.

Finally, at **operator level**, one-to-many dependencies, in which an op is the producer of multiple ops, may also lead to redundant

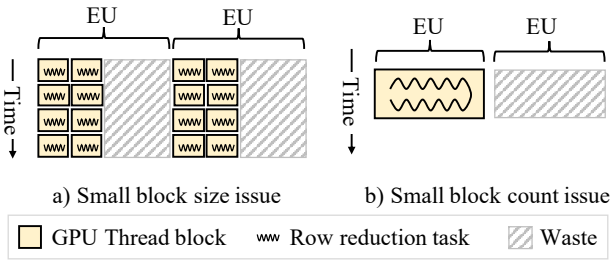


Figure 6: Typical poor parallelism issues in existing works.

computations. For example, in Figure.4, operators *B* and *C* will be separated into two different kernels in the state-of-the-art ML compilers with operator *A* inlined in the two kernels redundantly.

2.3.2 Challenge II: Irregular Tensor Shapes in Real-World Production Workloads. We also observe that there are tensor dimensions that appear highly uneven in production workloads, which we refer as *irregular tensor shapes*. This makes it challenging for compiler to generate kernels with good parallelism because it demands JIT optimization under the given tensor shapes that are not known in advance. We observe that state-of-the-art works lack of adaptive designs for this unique feature, resulting in poor performance.

Key Inefficiency: irregular tensor shapes lead to either too many small partitions or too few large partitions on GPU. We observe that tensor shapes in production workloads are usually different from the benchmarks in standard model-zoos. Figure.6 shows two cases of poor parallelism on GPU from production tensor shapes. For instance, processing row-reduce from shape <750000,32> to <750000>, a real-case in DIEN model[58], results in *small block size issue* on XLA (Figure.6-(a)). Here XLA auto-generates 750,000 GPU thread-blocks with the block-size of 32, leading to low parallelism. This is because there is an upper-bound number of thread-blocks that GPU can concurrently execute; when the thread-block size is too small, the concurrency at any given time is also low. Another case is for row-reduce from shape <64,30000> to <64>, a real-case in a Transformer model, resulting in *small block count issue* on XLA (Figure.6-(b)). XLA auto-generates 64 thread-blocks with the size of 1024, whereas a V100 GPU can concurrently schedule 160 thread-blocks for the same block size, causing serious hardware under-utilization. Therefore, these production workloads demand a better compiler design to automatically generate thread mappings suitable to various tensor shapes.

3 KEY DESIGN METHODOLOGY

High-level Objectives. To address challenge I and II discussed previously, we propose *AStitch*, a machine learning optimizing compiler that opens a new multi-dimensional optimization space for memory-intensive ML computations by supporting efficient just-in-time operator stitching for arbitrary memory-intensive sub-graphs.

In this section, we describe our basic idea named *hierarchical data reuse* (Sec.3.2) to address challenge I (Sec.2.3.1), and *adaptive thread mapping* (Sec.3.3) to address challenge II (Sec.2.3.2). We present our operator-stitching scheme abstraction based on joint consideration (Sec.3.1) for dependency abstractions discussion.

Note that this section highlights the code generation insights of stitching, which can be applied either manually or via compiler optimizations. We will describe how to develop the optimizing compiler in Sec.4 based on the insights from this section.

Table 1: Stitching scheme abstraction with joint consideration.

Scheme	Dependency	Memory Space	Locality v.s. Parallelism
Independent	None	None	-
Local	one-to-one	Register	-
Regional	one-to-many	Shared memory	CAT locality first
Global	Any	Global memory	Parallelism first

3.1 Operator-Stitching Scheme Abstraction

As is shown in Table.1, we abstract four types of stitching schemes, covering all the scenarios of dependencies from the joint consideration of dependency, memory hierarchy and parallelism.

Independent scheme represents operators that are independent of each other. **Local** scheme represents adjacent operators with element-level one-to-one dependency (i.e., element-wise manner) and the intermediate data is buffered in the per-thread register. Here thread level data locality is guaranteed. These two schemes are adopted by the state-of-the-art designs [11, 18, 55].

Regional scheme indicates one-to-many element-level dependencies and the intermediate data here are buffered on GPU's shared memory for its consumers. The thread mapping (parallelism) should guarantee thread-block level data locality. Take the example in Figure.5, where each result of *power* can be cached on shared memory and reused by multiple threads performing *add*. **Global** scheme is to cope with any complex dependency and the intermediate data is buffered on global memory. Since this is a parallelism-oriented scheme, there is no locality requirement (global memory is visible to all threads). Note that just like *Regional* scheme, the most common case for the current machine learning models is still one-to-many element-level dependency. However, sometimes optimizing for block locality hurts parallelism, leading to poor overall performance. For example, Block locality requires organizing corresponding threads into a single thread block, which prevents the potential higher parallelism by mapping these threads onto more thread blocks. In this case, *AStitch* decides the tradeoff between locality and parallelism (i.e., Regional vs Global) based on the characteristics of the operator (Sec.4.3). In Sec.4, we will discuss how to automatically determine the best stitching scheme according to operator's thread mapping.

3.2 Hierarchical Data Reuse Illustration

The multiple operator-stitching schemes enable a hierarchical data reuse, with which we can stitch any operators together efficiently without large computation redundancy. It helps to break the dilemma in Challenge I (Sec.2.3.1). We first illustrate how the intermediate data between operators are maintained and reused for each operator-stitching scheme.

3.2.1 Data Reuse. Given the stitching schemes in Table.1, *AStitch* enables two-levels of data reuse across memory hierarchies (i.e., registers, shared memory and global memory) to eliminate the fusion dilemma:

Element-level data reuse. In *AStitch* , for one-to-many element-level dependencies, the producer processes each data item only once without incurring redundant computations. The result can be maintained on GPU shared/global memory buffer for its consumer(s) to reuse in regional/global stitching scheme.

Operator-level data reuse. For one-to-many operator-level dependencies, *AStitch* processes the producer only once and buffers its result for its multiple consumers to reuse. For local stitching scheme, the to-be-reused data is maintained on register, while the data is maintained on shared/global memory for regional/global memory. Note that in current ML compilers the multiple consumers (e.g., operators *B* and *C* in Figure.4) may be separated into different kernels, causing redundant computations of the producer (e.g., operators *A* in Figure.4).

3.2.2 Kernel Form Illustration. In Figure.7-(a), we illustrate what a generated kernel looks like with hierarchical data reuse for a sample subgraph. Here we assume the stitching schemes for the operators are: *regional* scheme for *reduce.1*, *global* scheme for *power.1* and *reduce.2*, *independent* scheme for *multiply.1*, and *local* scheme for other operators.

Figure.7 (b) and (c) show how XLA and *AStitch* form the kernel differently for the subgraph. *AStitch* forms one GPU kernel, while XLA forms 4 kernels ending with *reduce.1*, *power.1*, *reduce.2* and *multiply.1*. *AStitch* eliminates 3 kernel launches with fine-grained data management and multi-level thread barriers, which reduces CPU-GPU context switch and framework scheduling overhead (with extra lightweight thread barriers in a kernel). The output of *reduce.1* does not need to be flushed to off-chip memory; it is buffered on-chip for its consumer to read (i.e., element-level data reuse). The values of *parameter.2* and *broadcast.2* only need to be loaded once from the off-chip memory and are buffered on registers (i.e., operator-level data reuse), while XLA loads them twice from the global memory in different kernels.

Additionally, TVM will form 3 kernels for this case, where *power.1* and *reduce.2* are merged into one kernel. This results in redundant computations of *power.1*. *AStitch* avoids such redundant computations. The detailed design optimizations for automatic thread mapping and stitching strategies will be discussed in Sec.4.3.

3.2.3 Global Barrier. *Global* stitching scheme requires a global barrier for all the GPU threads under the following constraints: the total number of GPU thread-blocks should not exceed the allowable number that can simultaneously execute on GPU per wave [50]. Otherwise, it will cause dead-locks between active and inactive thread-blocks. This constraint is met under Sec.3.3 which helps limit the thread-block number while still retaining high parallelism. Compared to separated kernels, *global* stitching inlines the implicit global thread barriers between kernel calls into a single kernel.

3.3 Adaptive Thread Mapping

As discussed in Sec.2.3.2, state-of-the-art ML compilers lack of designs to support irregular tensor shapes presented in production

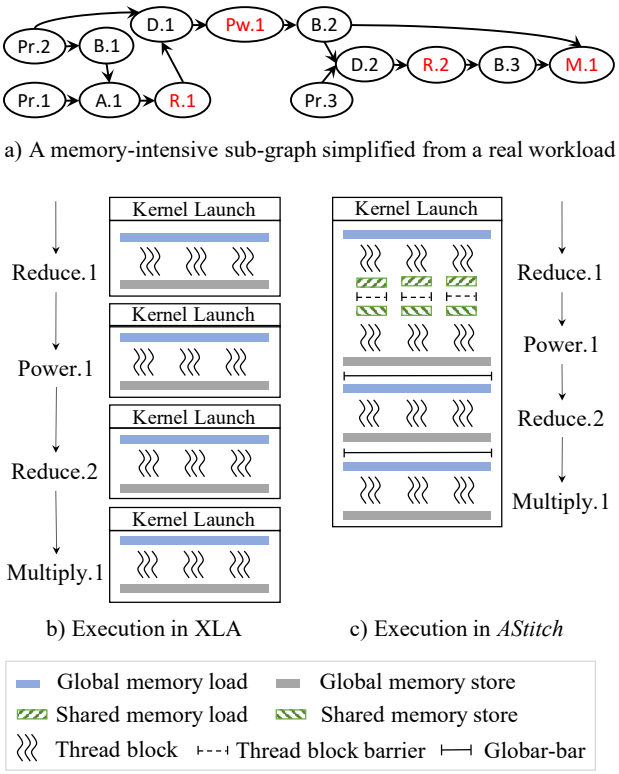


Figure 7: Execution scheme of a memory-intensive subgraph. *AStitch* reduces kernel launches and off-chip memory access with hierarchical data reuse. Pr: parameter. A: add. B: broadcast. R: reduce. D: divide. Pw: power. M: multiply.

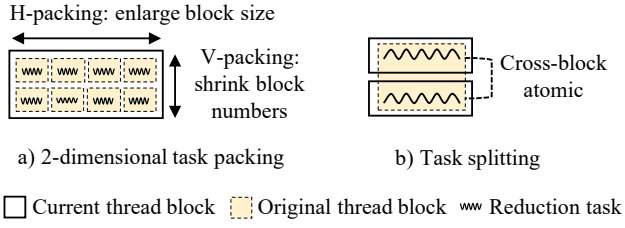


Figure 8: Task packing and splitting optimization for row-reduction on GPU. Task mappings in existing work are in Fig.6.

workloads (Challenge II), causing significant performance issues. To address this issue, we propose a *task packing and splitting* approach based on the SIMT nature of GPU execution to adaptively process various tensor shapes. Note that what suffer from irregular tensor shapes are mainly *reduce* ops, which are the most time-consuming operators among memory-intensive computations.

Task packing includes two dimensions: horizontal and vertical. *Horizontal packing* is to pack multiple small blocks, each of which processes the reduction of one row, into one large thread block. This

fixes the small block-size issue shown in Figure.6-(a). For example, for the case of row-reduction from shape <750000,32> to <750000> (Figure.6-(a)), we pack 32 small blocks with size of 32 into one thread-block with size of 1024 to increase parallelism, and every 32 threads in the thread-block process a row. *Vertical packing* is to pack tasks of multiple thread-blocks into one to reduce the block count. This helps pack multiple waves of thread-blocks into one wave to meet the requirement of a global barrier. The block size is unchanged in the vertical packing and each thread processes elements from multiple tasks in order. Figure 8-(a) describes how the two-dimensional task packing works to increase GPU utilization while effectively limiting the block counts per wave.

On the other hand, *task splitting* is to split the task within one thread-block into several thread-blocks to increase block count, in case there is under utilization problem caused by small block count (Figure.6-(b)). It requires a cross-block reduction between split blocks for row-reduction execution. Figure 8-(b) describes how the row-reduction task within one block is split into two blocks via a cross-block atomic to increase parallelism.

4 COMPILER DESIGN AND OPTIMIZATIONS

It is challenging to apply the optimizations described in Sec.3 automatically for a given complex memory-intensive subgraph. It is required to determine the stitching scheme along with thread mapping for all the operators just-in-time. However, enumerating the schemes for each operator results in combinatorial explosion. Fortunately, we identify *two basic characteristics* of memory-intensive operators according to the abstraction of the stitching schemes. We make *a key observation* that *AStitch* only needs to determine the stitching scheme and the corresponding thread mapping for several key operators, and these will propagate to all the other operators in the subgraph. We describe how *AStitch* enables the proposed optimizations automatically as follows.

4.1 Stitching Scope Identification

First, we describe how *AStitch* identifies what operators to stitch together given a machine learning model. To minimize CPU-GPU context switch, framework scheduling overhead and reduce off-chip memory access, *AStitch* stitches as large scope of memory-intensive operators together as possible (under resource constraints) for a given ML computation graph. Note that *AStitch* is capable to manage hardware resources effectively to prevent resource explosion for larger kernels (Sec.4.4, Sec.4.5).

Given a ML computation graph, *AStitch* first applies a BFS algorithm to identify memory-intensive sub-graphs automatically. It then replaces each of the sub-graphs with a new operator, named *stitch op*. To further enlarge the stitching scope, *AStitch* groups disconnected *stitch ops* together and forms a larger *stitch op*, which we call *remote stitching*. The process of *remote stitching* is to traverse all the existing *stitch ops* and merge two together if they have no data dependency. A constraint to form a *stitch op* is that no cyclic dependence is allowed. *AStitch* does not merge computations together if the merging results in a circle. Each of the *stitch ops* will be compiled into a CUDA kernel.

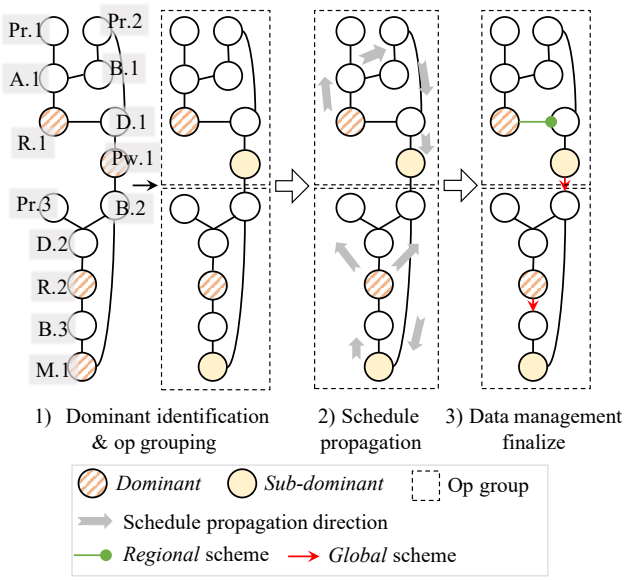


Figure 9: Schedule propagation and data management planning for the graph topology shown in Figure.7-(a).

4.2 Key Design Observations

We make the following observations in regard to the abstraction of stitching schemes (Sec.3.1):

Observation-A: If an operator is of *local* scheme, its thread mapping can be determined by propagating from its consumer's thread mapping. This is because *local* scheme indicates element-level one-to-one dependency. Given the schedule of the consumer op, the schedule of its producer can be derived directly from the consumer by element-wise index propagation. For the same reason, if both an operator and its consumer are all of *local* scheme, the thread mapping schedule of this op can be propagated to its consumer.

Observation-B: The patterns of *reduce* and expensive element-wise ops followed by *broadcast* ops need to be supported by either *regional* or *global* scheme because the two patterns induce complex element-level *one-to-many* dependencies (Table.1).

4.3 Automatic Compiler Optimization Design

According to these two key design observations, we propose a just-in-time compiler design that automatically determines the stitching scheme and thread mapping for each operator. Using the graph topology in Figure.7-(a) as an example, Figure 9 illustrates how the compiler works in three steps.

Step 1: dominant identification and op grouping. According to the observations in Sec.4.2, *AStitch* only needs to determine the thread mapping of several key operators, and then propagate them to all the other operators. We name these key operators as *dominant ops*. *AStitch* first identifies several candidates for becoming *dominant ops*, and eventually identify the final ones with *dominant merging*.

First, according to observation-B, the light element-wise ops and the expensive element-wise ops that are not followed by *broadcast* are of *local* scheme by default. The ops that are not of *local* scheme

will be considered as the candidates for *dominant ops*. That means that *reduce* and expensive element-wise ops followed by broadcast are candidates for *dominant ops*. Here, a special case is the output of *stitch ops* (e.g., *M.1* in Figure.9), which is also considered as a candidate *dominant op*. Under this rule, *reduce.1*, *power.1* and *reduce.2* are candidates for *dominant ops* in Figure.7-(a).

AStitch then identifies the final *dominant ops* from the candidate *dominants*, which is referred as **dominant merging**. If two candidate *dominants* connect with each other through ops of only *local* scheme, *AStitch* chooses one as final *dominant op*, and regards the other as **sub-dominant**. Thus we only need to determine the thread mapping schedule for the *dominant op*, and can get the the schedule of *sub-dominant* through propagation. *AStitch* prefers to choose reduction as the final *dominant op* when merging. This is because using a time-consuming op like reduction as *dominant op* to generate the overall schedule usually leads to better performance.

Finally, *AStitch* will form a group for each *dominant* op, including all the ops that connect to this *dominant* under only *local* scheme. In this way, different groups are connected with each other through only *dominant* and *sub-dominant* ops. Later, *AStitch* will propagate thread mapping within each group, and identify how different groups communicate with each other by determining the stitching schemes for *dominant* and *sub-dominant* ops.

In Figure.9-1, *reduce.1* and *reduce.2* are the final *dominants*. *Power.1* (*multiply.1*) can connect to *reduce.1* (*reduce.2*) through ops of only *local* scheme. For example, the path between *reduce.1* and *power.1* is *add.1 - broadcast.1 - parameter.2 - divide.1*. *AStitch* regards *Power.1* and *multiply.1* as *sub-dominants*.

Step 2: adaptive thread mapping and schedule propagation. *AStitch* generates the parallel code for each *dominant* op according to Sec.3.3, and propagates the thread mapping schedule within the corresponding group. In this way, we get thread mapping schedule for all the operators.

Tensor Shape Adaptation. *AStitch* automatically applies task packing and splitting for *dominant* ops according to tensor shapes and hardware resources. Take row reduction as an example, if the number of *rows* to be reduced is smaller than the number of blocks allowed per-wave, and each *row* contains a large number of data items (i.e., larger than 1024), *AStitch* splits the row to increase parallelism. Otherwise, *AStitch* will apply task packing to form large enough blocks, and limit the block count to be smaller than the per-wave threshold to meet the requirement of global barrier.

We want to emphasize that the *dominant merging* in Step-1 can enable more operator-level data reuse (Sec.3.2.1). For example, in Figure.7-(a), without merging *reduce.2* and *multiply.1* into one group, *broadcast.2* will appear in two groups. This may result in different thread mapping schedules for *broadcast.2* in two groups; thus the per-thread loaded value of this op cannot be reused between the two groups due to the schedule incompatibility. Under our dominant merging of the two groups, *broadcast.2* now can be cached on registers for reuse as the whole group is dominated by the same thread mapping schedule.

Step 3: Finalization. *AStitch* determines the stitching schemes for the *dominant* and *sub-dominant* ops in the last step. Note that other ops within each group are of *local* scheme, discussed in Step-1. Different groups connect with each other through *dominant* and *sub-dominant* ops, for which the stitching scheme is either *regional* or *global*.

Regional scheme requires block-level data locality. For an op whose output is in *regional* scheme, whenever the op produces a range of data in a block, its consumers should read the same range of data within the same block. Otherwise, the scheme should fall back to *global*. We apply locality check to identify the stitching scheme between *regional* and *global*.

Passive block-locality checking. Take the example in Figure.7-(a), we calculate how many continuous elements each of *reduce.1*, *power.1* and *reduce.2* produces per-block, and how many continuous elements their consumer requires per-block, respectively. In this case, only *reduce.1* matches block locality and is given *regional* scheme for data buffering on the GPU shared memory; the other two are given *global* scheme for data buffering on the global memory.

Proactive block-locality adaptation. For an op group that only contains element-wise ops, *AStitch* proactively adjusts the thread mapping schedule of this group to match the block-locality with its producer group. According to how many continuous elements each thread-block generates by the producer of this group, the thread mapping schedule can be determined in an element-wise manner for this group.

In summary, *reduce* ops prioritize parallelism since they require more computation. Thus, *reduce* dominated groups will perform passive block-locality checking without adjusting its thread mapping. On the other hand, element-wise ops typically prioritize locality due to their low-cost computation. Thus, groups dominated by them perform proactive block checking to enable more block-level locality.

4.4 Memory Usage Optimization

It is essential to use GPU shared/global memory moderately, e.g., high shared memory usage hurts kernel parallelism. *AStitch* operates under the consideration of memory resource limitations and their impacts on performance. For example, *AStitch* reuses previous allocated memory as much as possible to reduce unnecessary memory allocation requests. *AStitch* uses dominance tree algorithm[19] for memory data-flow analysis to maximize the memory reuse for operators. Additionally, *regional* scheme requires extra shared memory usage. If a shared memory request exceeds the hardware limit for a thread-block, *AStitch* alters the stitching scheme of *dominant* and *sub-dominant* ops from *regional* to *global* one by one within a *stitch op* until the memory usage is under the limit.

4.5 Resource-Aware Launch Configuration

AStitch requires a proper GPU kernel launch dimension for which the thread-block count does not exceed the max block count per-wave ($C_{blocks-per-wave}$) to meet the requirement of global barrier. Unfortunately, although *AStitch* needs the information (e.g., $C_{blocks-per-wave}$) to make thread mapping plan for GPU binary compilation, it can only be achieved after the compilation. We design an *assume-relax-apply* approach to address this issue. The basic idea is to assume a target $C_{blocks-per-wave}$ before optimization, then get a relaxed register usage, and finally apply the register usage limitation with CUDA compiler notations to achieve the target.

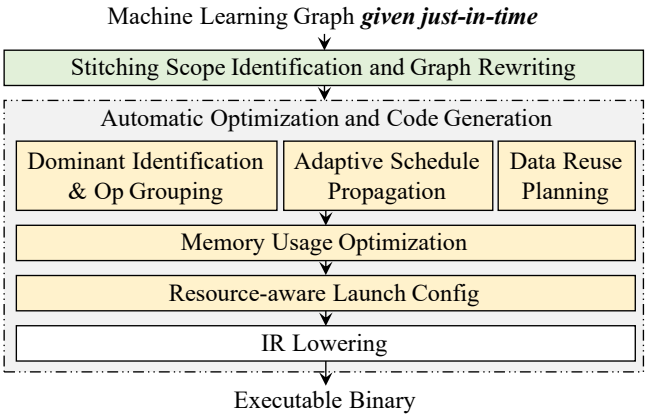


Figure 10: *AStitch* System Overview.

First, we assume the register usage bound is a very small value (i.e., 32). Second, we try to relax the bound if possible. The insight is that, the parallelism may be bounded by shared memory usage but not register usage, for which we can relax the register usage bound. We calculate $C_{blocks-per-wave}$ according to the assumed register usage bound, planned shared memory usage (Sec.4.4) and specified block size[4]. We specify the block size as the upper bound that CUDA allows (e.g., 1024) during the above process, as larger block size results in smaller $C_{blocks-per-wave}$, and in turn results in smaller global barrier overhead. Then we deduce the max register usage allowed according to $C_{blocks-per-wave}$. The max allowed register usage is the relaxed bound. Finally, *AStitch* adds annotation information to apply max register bound when lowering thread mapping schedule to GPU IR. We do not observe register spilling problem in our evaluation with this method.

5 IMPLEMENTATION

Figure.10 shows *AStitch* system overview.

The designs and insights of *AStitch* can be directly applied on all the current machine learning frameworks [12, 17, 36] and ML optimization compilers [11, 16, 18, 32, 55] for enhancement. Currently, we implement *AStitch* as a TensorFlow add-on. *AStitch* leverages TensorFlow built-in XLA engine for the system implementation. It retains all the optimizations of XLA except fusion strategies and code generation passes. *AStitch* accepts the computation graphs represented in XLA but replaces XLA's fusion and codegen passes.

We build *AStitch* as a stand-alone compiler engine that is portable to any version of TensorFlow. It leverages the custom graph pass API of TensorFlow to rewrite computation graph and generate GPU code for *stitch ops*. It then leverages custom op API of TensorFlow to plug *stitch ops* into the runtime sequence of operators in the computation graph.

To use *AStitch*, users only need to specify the path of *AStitch* engine along with an environment variable to initialize it. Users do not need to modify anything in the model script, making *AStitch* a very usable machine learning compiler.

Table 2: Workloads for evaluation.

Model	Field	Train batch-size	Infer batch-size
CRNN	Images	-	1
ASR	Speech	-	1
BERT	NLP	12	200
Transformer	NLP	4,096	1
DIEN	Recommendation	256	256

6 EVALUATION

In this section, we present the detailed evaluation results using a single NVIDIA V100 GPU with 16GB device memory with CUDA toolkit 10.0 and cuDNN 7.6.

6.1 End-to-End Evaluation

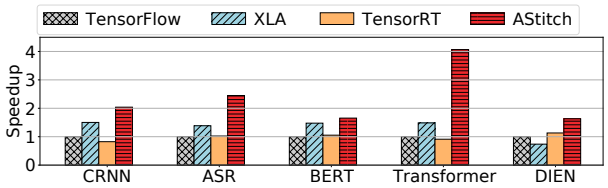
Workloads. We use a set of representative memory-intensive machine learning applications as our evaluation workloads, which include BERT[21] and Transformer[44] for natural language processing, DIEN[58] for recommendation, ASR[2][47] for automatic speech recognition and CRNN[40] for optical character recognition. These models are widely used in production. The building blocks of these workloads include perceptron, attention, convolution, RNN, and a broad range of memory intensive operators. Table.2 summarizes the fields and configurations of the evaluated applications. All of our workloads' batch sizes are selected from real-world production configurations.

Baselines. We compare *AStitch* with TensorFlow (v1.15), TensorFlow XLA (v1.15) and TensorRT (v7.0) to demonstrate the benefits of *AStitch*. Note that TensorRT is one of the most popular optimizer for inference. We evaluate the speedup of *AStitch* by comparing inference time or the training time of one iteration. We repeat 10 times and use the average performance to validate speedup. TensorRT does not support training, for which we only evaluate inference workloads. Ansor (i.e. TVM Auto-scheduler[55]) lacks support for most models we use. Thus, we present a detailed case study with Ansor in Sec.6.2. During our test, the accuracy are the same between *AStitch* and other techniques.

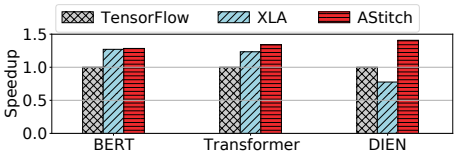
6.1.1 Overall Results. Figure.11 shows the end-to-end performance speedup for all workloads, where the execution time of TensorFlow is normalized to 1. *AStitch* outperforms all other techniques we compare. For the inference workloads, compared to TensorFlow, our approach achieves up to 4.06× speedup, with 2.37× on average. Compared to XLA, our approach achieves up to 2.73× speedup, with 1.84× on average. Compared to TensorRT, our approach achieves up to 4.46× speedup, with 2.47× on average. For the training workloads (Figure.11b), *AStitch* shows 1.34× average speedup compared to TensorFlow and 1.30× to XLA. Note XLA shows performance degradation for DIEN because it increases CUDA memcpy/memset activities for this model.

For the inference workloads, we have evaluate *AStitch* on NVIDIA T4 GPU, which is widely used for inference in production. Meanwhile, we also evaluate *AStitch* along with auto mixed precision (AMP) optimization[2] (Figure.12), which shows speedup similar with that in Figure.11. This means *AStitch* is applicable to more

[2]Porting from https://github.com/espnet/espnet



(a) Speedup of inference tasks.



(b) Speedup of training tasks.

Figure 11: End-to-end performance speedup.

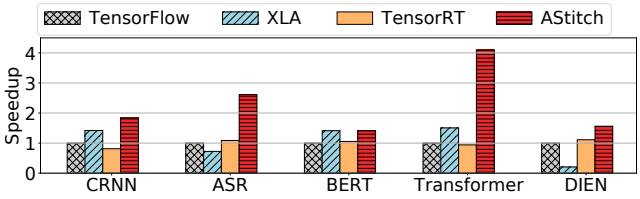


Figure 12: Inference speedup, for which baselines and *AStitch* are all with AMP optimization.

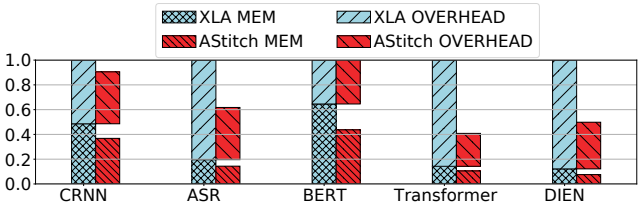


Figure 13: Performance breakdown, without showing the time of compute-intensive ops.

generations of NVIDIA GPUs and can be used in good combination with AMP.

For real-world training workloads, they usually contain less portions of memory-intensive computations than inference, thus demonstrating comparatively smaller speedup. It is also related to model structure itself, e.g.,DIEN gains higher training speedup than BERT and Transformer.

6.1.2 Performance Breakdown. We classify the execution time for a model into three parts: memory-intensive op execution (*MEM*), compute-intensive op execution and non-computation overhead (*OVERHEAD*). Note that we do not explore multi-stream execution and thus the total execution time is the summary of the three parts.

Figure.13 shows the timeline breakdown of *MEM* and *OVERHEAD* for XLA and *AStitch*. We collect the metrics with nvprof tool[10]. The total time of *MEM* and *OVERHEAD* in XLA is normalized to 1. *AStitch* significantly reduces both the execution time of memory-intensive ops (*MEM*) and non-computation overhead (*OVERHEAD*). For example, about 2/3 *OVERHEAD* time and 1/4 *MEM* time is saved for Transformer. The decrements of *OVERHEAD* mainly comes from kernel call decrements, and *MEM* benefits from parallelism increment.

Table 3: Kernel numbers. MEM: kernel of memory-intensive ops. CPY: CUDA memcpy/memset calls.

		CRNN	ASR	BERT	Transformer	DIEN
MEM	XLA	986	496	64	10,132	2,579
	AStitch	297	218	26	2,578	811
CPY	XLA	406	372	25	5,579	628
	AStitch	388	203	10	1,474	422

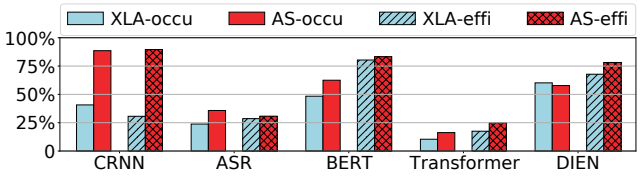


Figure 14: Average parallelism of top 80% memory-intensive computations. AS: *AStitch*. Occu: occupancy. Effi: SM-efficiency.

Kernel Call Decrements. Table.3 shows the number of memory-intensive kernel calls of XLA and *AStitch*. Thanks to the exhaustive stitching, 65.7% kernel calls of memory-intensive computations are saved on average. This leads to much fewer context switches and therefore reduces the non-computation overhead by a large margin. Moreover, *AStitch* reduces 43.2% CUDA memcpy/memset activities on average to further eliminate non-computation overhead.

Parallelism Increment. We profile the GPU kernels and collect performance counters to show that *AStitch* increases the parallelism and hardware utilization. We only focus on the top 80% memory-intensive kernels according to the execution time, which is enough to represent the overall effect.

We collect two GPU performance counters with nvprof profiling tool: *achieved_occupancy* and *sm_efficiency*. *Occupancy*[3] shows whether enough threads are scheduled in parallel, indicating parallelism. *Sm_efficiency*[10] shows the percentage of elapsed cycles that GPU SM is busy, indicating GPU utilization. The higher the two metrics, usually indicating the better utilization.

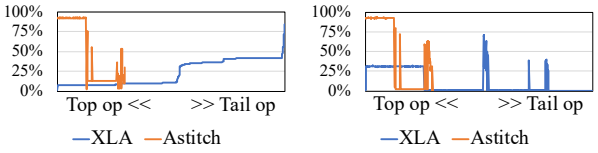
Figure.14 shows the average value of the two metrics for the top 80% memory-intensive kernels. Thanks to the adaptive thread mapping, *AStitch* increases the parallelism and GPU utilization overall. *DIEN* shows a little decrements of *occupancy* with only 2%, but increases *sm_efficiency*, indicating higher GPU utilization.

6.1.3 A Comprehensive Case Study. We make a case study with *CRNN* to illustrate the performance benefit.

Ablation Study. We enable the optimizations in *AStitch* one by one to justify and prove the design choice of *AStitch*. The baseline is XLA. We first enable the *adaptive thread mapping* based on the

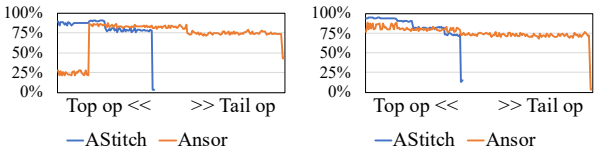
Table 4: Ablation study for *CRNN*.

	XLA	ATM	HDM	AStitch
Time (ms)	23.95	21.98	20.45	17.64



(a) achieved_occupancy (b) sm_efficiency

Figure 15: CRNN occupancy and SM efficiency trend. X axis indicates memory-intensive ops in descending order of execution time. Note *AStitch* has less ops.



(a) achieved_occupancy (b) sm_efficiency

Figure 16: BERT occupancy and SM efficiency trend. Axis has the same meaning with Figure.15. Note *AStitch* has less ops.

Table 5: Total performance counters of all memory-intensive ops in *CRNN*. DR_transactions: dram_read_transactions. DW_transactions: dram_write_transactions

	DR_transactions	DW_transactions	inst_fp_32
XLA	104,056,236	63,793,690	1,700,113,391
AStitch	104,022,389	16,302,582	1,675,090,268

stitching scope of XLA (*ATM*). Note that XLA has smaller fusion scopes than *AStitch*. Then, we apply *exhaustive stitching* with *hierarchical data management* onto *ATM*, without *dominant merging* (*HDM*). Finally, we apply the complete function *AStitch*.

Table.4 shows the ablation result. The adaptive thread mapping technique based on the fusion plan of XLA contributes 8.9% speedup, by increasing the parallelism and GPU utilization. The exhaustive stitching technique, without dominant merging, contributes another 8.2% speedup, by reducing context switch overhead and off-chip memory traffics, and exploring element-level data-reuse. The dominant merging technique contributes the final 18.7% speedup, by enabling op-level data-reuse.

Performance Counter Analysis. We analyze the GPU performance counters for memory-intensive ops in three aspects: **1) Parallelism.** Figure.15 shows the *achieved_occupancy* and *sm_efficiency* of kernels generated by XLA and *AStitch*. The top time-consuming kernels of *AStitch* shows higher parallelism (*achieved_occupancy*) and hardware utilization (*sm_efficiency*) than XLA. **2) Reduced off-chip memory traffic.** Table.5 shows that *AStitch* reduces the total GPU global memory load transaction (*dram_read_transactions*) and store transaction (*dram_write_transactions*) than XLA. The reduced global memory access is benefited from the hierarchical data management, which buffers a large portion of intermediate values on-chip. **3) Reduced instructions.** Table.5 shows that *AStitch* reduces the fp32 instruction count (*inst_fp_32*), due to reduced redundant computations.

6.2 Comparing with TVM Ansor: A Case Study

We present detailed analyses with Ansor[55] on BERT inference case. Note that Ansor fails to run other models shown in Table.2 due to limited operator support. We run Ansor auto-tuning for 2000 measurement trials and choose the best-tuned model for comparison.

The End-to-end evaluation shows that *AStitch* takes 31.75ms and Ansor takes 42.02ms for one inference. *AStitch* achieves 1.3× speedup.

AStitch forms 53% less GPU kernels for memory-intensive ops than Ansor, introducing much lower context switch overhead. Meanwhile, *AStitch* brings 1.4× speedups for all memory-intensive computations comparing with Ansor.

Figure.16 shows the *achieved_occupancy* and *sm_efficiency* of kernels generated by Ansor and *AStitch*. The top time-consuming kernels of *AStitch* shows higher parallelism (*achieved_occupancy*) and hardware utilization (*sm_efficiency*) than Ansor, indicating better thread mapping ability of *AStitch*. We also collect the GPU performance counter of global memory transactions for this case. The memory-intensive computations in Ansor requires 49,826,724 global memory read transactions and 47,262,821 global memory write transactions. While the metrics of *AStitch* is 33,038,694 and 28,432,641 respectively, reducing nearly 40% total off-chip memory transactions.

6.3 Production Evaluation

AStitch has been deployed into a production cluster and has saved around 20,000 GPU hours on 70,000 tasks within a week. This demonstrates the robustness of *AStitch*. Our previous trial using XLA fails to produce satisfactory results due to its negative optimization on many models.

The workloads *AStitch* optimizes effectively on cluster mainly include transformer based models, recommendation models, and RNN models. About 23% jobs are distributed jobs, consuming 56% total GPU time among all machine learning jobs. Others are single GPU jobs.

The method we estimate the saved GPU hours is that, we run the deployed model with TensorFlow for several iterations at the beginning, and optimize with *AStitch* in later iterations. The execution time is logged in every iteration, except for the first 10 iterations that include initialization overhead. We compute the total time saved by multiplying the number of iterations and time saved per iteration.

Table 6: Overhead of inlined global-bar.

#block	20	40	60	80	100	120	140	160
Time(us)	2.53	2.53	2.59	2.59	2.66	2.66	2.69	2.72

6.4 Overhead Analysis

6.4.1 Optimization Overhead. The optimization overhead of *AStitch* comes from the process of exhaustive stitching, thread mapping, and data management planning. We measure the overhead time from accepting the input graph until just before lowering LLVM IR to CUDA binary. We measure the overhead on computation graphs with 5,000 to 10,000 nodes and the results show that *AStitch* introduces an overhead of 90s in average where originally XLA requires 30s in average. Just like compilation overhead, the overhead of *AStitch* is introduced only once for all following iterations of training/inference. The overhead, although non-negligible, is still much more efficient than searching and tuning-based optimizations.

6.4.2 Global Barrier Overhead. Table 6 shows the overhead of global barrier under different GPU block numbers, with a block size of 1024. Time is the duration of a kernel consists of only a global barrier, without any other computation. We measure the kernel time with nvprof. A V100 GPU can accommodate at most 160 such thread blocks concurrently. Thus the overhead of global barrier is no more than 2.72us in *AStitch*, less than the kernel launch overhead on the order of 10 microseconds.

To justify the overhead of global barrier in the real model, we remove the global barrier in *AStitch* and measure the end-to-end performance of *CRNN* model. Note that the result is not correct in such a test. We do not observe obvious performance improvement when removing the global-bar. It shows that the global barrier is not the bottle-neck of *CRNN* case.

7 RELATED WORK

Many current machine learning compiler optimizations mainly address compute-intensive operations [14, 16, 18, 20, 28, 33, 39, 42, 43, 55], while paying limited attention to the performance issue of memory-intensive computations. Some compilers [11, 14, 16, 18, 39, 43, 55, 59] apply fusion optimization for memory-intensive ops. These works lack to tackle the complex dependency problem and suffer from insufficient fusion. *AStitch* addresses this problem and widens the fusion scope upon previous works.

There are some works study about fusion for machine learning specifically. Li et al.[31] discusses how to fuse operators with no data dependencies horizontal to improve parallelism. Abdolrashidi et al.[13] study about fusion strategy rather than how to generate efficient kernel for a fusion. Wang et al.[46], Sivathanu et al.[41] and Ashari et al.[15] explore compute-intensive computation related fusion optimization. ia et al.[25] tackle the problem with graph equivalent transformation at operator-level and apply fusion of compute-intensive operators (i.e., GEMM and Convolution). However, it does not investigate how to generate efficient code for memory-intensive computation graphs. HuggingFace[48] provides APIs to build transformer structures and relies on other frameworks (e.g., XLA) for kernel fusion. *AStitch* is orthogonal with the above studies in that it focuses on generating high performance GPU kernels given a large group of memory-intensive operators just-in-time. Niu et al.[34] make studies about fusion optimization for the inference on mobile devices, while *AStitch* targets both training and inference on industrial GPU vendors, showing different targets and techniques. Zheng et al.[57] explore operator stitching with shared memory, and use a two-level cost-model based method for fusion pattern decision and codegen schedule selection. *AStitch* enlarges the optimization space with *global scheme* stitching, and avoids expensive cost-model based searching thanks to the *adaptive thread mapping* ability.

There are ad-hoc optimizations focusing on specific structures, such as *LayerNorm* [9, 23], CNNs [26, 52] and RNN cells [22]. DeftNN [24] applies model compression and data fission to speedup CNN networks. *AStitch* provides a general JIT compiler rather than model-specific optimizations. There are some works study about data preprocessing. POCLib [53] is a high-performance framework enabling near orthogonal processing on compression for a wide range of applications. G-TADOC [51, 54] is an efficient GPU-based text analytic system without decompression. *AStitch* is orthogonal to these data processing works and can be combined with them.

There are some works addressing the issue of non-computation overhead. Ma et al.[32] and Zheng et al.[56] leverage persistent-thread technique to schedule tasks and reduce kernel launch overhead, which do not explore how to generate efficient GPU kernel under a set of to-be-fused memory-intensive operators with complex data dependencies (e.g., one-to-many). Meanwhile, comparing with [32, 56], *AStitch* is totally automatic and capable to leverage the highly tuned libraries for compute-intensive computations (cuDNN[8], cuBLAS[7]). Kwon et al.[30] address the framework overhead with op schedule planning, but not focus on kernel launch overhead studied in *AStitch* . CUDA Graph[5] binds, but not fuses, GPU kernels to reduce kernel launch overhead, which still suffers from off-chip memory traffic. Furthermore, it results in high GPU memory consumption to store all the graph metadata of every kernel[35]. AStitch does not have these problems and explores a larger optimization scope beyond CUDA Graph.

8 CONCLUSION

We reveal that memory-intensive computation is a rising performance critical factor in recent machine learning models. We propose *hierarchical data reuse* technique to address the complex dependencies to enlarge fusion scope, reducing non-computation overhead. We propose *adaptive thread mapping* technique to deal with the problem of irregular tensor shapes. We develop a JIT compiler named *AStitch* integrating the optimizations with high usability. Results show that *AStitch* outperforms state-of-the-art compilers with up to 2.73× speedup. We believe *AStitch* fills a long-overlooked gap of machine learning compilers.

ACKNOWLEDGMENTS

We thank anonymous reviewers and our shepherd, Dr. Peng Wu, for their extensive suggestions. We thank Prof. Feng Zhang (RUC) and Prof. Tzu-Mao Li (UCSD) for their proofreading. We also want to acknowledge University of Sydney (USYD) faculty startup funding, SOAR faculty fellowship and ARC DP210101984.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains the necessary software components to validate the main results in *AStitch* paper. We provide a docker image to ease the environment setup. The docker image contains the compiled binary of *AStitch*, scripts to evaluate the inference and training performance, and scripts to draw the figures. It requires a Linux system with NVIDIA driver (capable to run CUDA 10.0) running on a NVIDIA V100 GPU equipped x86_64 machine to create the docker container. After launching the docker container, people can run one script to collect all performance numbers. It requires some manual finishing to fill the performance numbers into several python scripts to draw the most important figures in the paper, showing the speedup of *AStitch* and breakdown information.

A.2 Artifact Check-List (Meta-Information)

- **Binary:** The docker image of *AStitch*.
- **Run-time environment:** A Linux system with NVIDIA driver (capable to run CUDA 10.0).
- **Hardware:** NVIDIA V100 GPU.
- **Output:** Performance results and figure to show speedup (need some manual finishing).
- **Experiments:** For all inference workloads evaluated in the paper, it contains the evaluation of naive TensorFlow, XLA, TensorRT and *AStitch*. For BERT and Transformer training, it contains the evaluation of naive TensorFlow, XLA, and *AStitch*.
- **How much disk space required (approximately)?:** 12GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** It requires dozens of minutes to download the docker image. You can then run a script once to collect all performance results. The execution takes about 2.5 hours and you can do something other while waiting for the results. Finally, it requires about 20 minutes for manual finishing to draw the figures to show speedup and breakdown.
- **Publicly available?:** Yes. The docker image is public, which contains the compiled binary. Source code is in the process of open source, which we will release at early 2022.
- **Code licenses (if publicly available)?:** Apache-2.0.
- **Data licenses (if publicly available)?:** Apache-2.0.
- **Archived (provide DOI)?:** 10.5281/zenodo.5733989

A.3 Description

A.3.1 How to access. We provide the docker image at both docker-hub and zenodo.

Docker-hub URL: https://hub.docker.com/r/jamesthez/astitch/tags.

Zenodo URL: https://zenodo.org/record/5733989.

A.3.2 Hardware dependencies. NVIDIA V100 GPU equipped x86_64 machines.

A.3.3 Software dependencies. Linux system with NVIDIA driver capable to run CUDA 10.0.

A.4 Installation

You just need to pull the docker image and launch a container:

```
docker pull \
    jamesthez/astitch:astitch_asplos_ae
docker run \
    --gpus all --net=host --pid=host -it\
    --name <your-container-name> \
    jamesthez/astitch:astitch_asplos_ae bash
```

Alternatively, you can download the tar file of docker image and import it. The download URL is https://zenodo.org/record/5733989. You can run the following command to launch the container:

```
gzip -d astitch_asplos_ae.tar.gz
docker import - astitch_asplos_ae < \
    astitch_asplos_ae.tar
docker run \
    --gpus all --net=host --pid=host -it \
    astitch_asplos_ae bash
```

Use *sudo* to run docker if necessary.

A.5 Evaluation and Expected Results

We have provided a read-me file on how to reproduce the key results within our provided docker image. You can find it at */root* after launching the docker container.

REFERENCES

[1] Cited July 2021. AMD GPU-Powered Machine Learning Solutions. https://www.amd.com/en/graphics/servers-radeon-instinct-deep-learning.

[2] Cited July 2021. Automatic Mixed Precision for Deep Learning. https://developer.nvidia.com/automatic-mixed-precision.

[3] Cited July 2021. CUDA Achieved Occupancy. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[4] Cited July 2021. CUDA Occupancy Calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[5] Cited July 2021. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/.

[6] Cited July 2021. GPU Dominates AI Accelerator Market. https://www.informationweek.com/ai-or-machine-learning/gpus-continue-to-dominate-the-ai-accelerator-market-for-now.

[7] Cited July 2021. NVIDIA cuBLAS. https://developer.nvidia.com/cublas.

[8] Cited July 2021. NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[9] Cited July 2021. NVIDIA FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[10] Cited July 2021. Nvprof Profiling Tool. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview.

[11] Cited July 2021. TensorFlow XLA. https://www.tensorflow.org/xla.

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

[13] Amirali Abdolrashidi, Qiumin Xu, Shibo Wang, Sudip Roy, and Yanqi Zhou. 2019. Learning to fuse. In *NeurIPS ML for Systems Workshop*.

[14] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[15] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices* 50, 8 (2015), 173–182.

[16] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[17] 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. *arXiv preprint arXiv:1512.01274* (2015).

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

[19] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[20] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 305–316.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[22] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*. PMLR, 2024–2033.

[23] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.

[24] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. 2017. Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 786–799.

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

[26] Wonkyung Jung, Daejin Jung, Sunjung Lee, Wonjong Rhee, Jung Ho Ahn, et al. 2018. Restructuring batch normalization to accelerate CNN training. *arXiv preprint arXiv:1807.01702* (2018).

[27] Konstantinos Kamnitsas, Christian Ledig, Virginia FJ Newcombe, Joanna P Simpson, Andrew D Kane, David K Menon, Daniel Rueckert, and Ben Glocker. 2017. Efficient multi-scale 3D CNN with fully connected CRF for accurate brain lesion segmentation. *Medical image analysis* 36 (2017), 61–78.

[28] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and Ponnuswamy Sadayappan. 2019. A code generator for high-performance tensor contractions on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 85–95.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[30] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. *arXiv preprint arXiv:2012.02732* (2020).

[31] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2020. Automatic Horizontal Fusion for GPU Kernels. *arXiv preprint arXiv:2007.01277* (2020).

[32] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 881–897.

[33] Matthew W Moskewicz, Ali Jannesari, and Kurt Keutzer. 2017. Boda: A holistic approach for implementing neural network computations. In *Proceedings of the Computing Frontiers Conference*. 53–62.

[34] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. 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*. 883–898.

[35] Alberto Parravicini, Arnaud Delamare, Marco Arnaboldi, and Marco D Santambrogio. 2020. DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime. *arXiv preprint arXiv:2012.09646* (2020).

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[37] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2018. Automatic kernel fusion for image processing DSLs. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. 76–85.

[38] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2019. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 242–253.

[39] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2020. Nimble: Efficiently compiling dynamic neural networks for model inference. *arXiv preprint arXiv:2006.03031* (2020).

[40] Baoguang Shi, Xiang Bai, and Cong Yao. 2016. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE transactions on pattern analysis and machine intelligence* 39, 11 (2016), 2298–2304.

[41] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong 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*. 909–923.

[42] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana 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*. 209–223.

[43] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[45] Mohamed Wahib and Naoya Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 191–202.

[46] Xueying Wang, Guangli Li, Xiao Dong, Jiansong Li, Lei Liu, and Xiaobing Feng. 2020. Accelerating Deep Learning Inference with Cross-Layer Data Reuse on GPUs. In *European Conference on Parallel Processing*. Springer, 219–233.

[47] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. 2018. ESPnet: End-to-End Speech Processing Toolkit. In *Proceedings of Interspeech*. 2207–2211. https://doi.org/10.21437/Interspeech.2018-1456

[48] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[49] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 107–118.

[50] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[51] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. G-TADOC: Enabling Efficient GPU-Based Text Analytics without Decompression. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1679–1690.

[52] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2016. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (2016), 905–918.

[53] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. POCLib: A High-Performance Framework for Enabling Near Orthogonal Processing on Compression. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2021), 459–475.

[54] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.

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

[56] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. Versapipe: a versatile programming framework for pipelined computing on GPU. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 587–599.

[57] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2020. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924* (2020).

[58] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 5941–5948.

[59] Kai Zhu, Wenyi Zhao, Zhen Zheng, Tianyou Guo, Pengzhan Zhao, Junjie Bai, Jun Yang, Xiaoyong Liu, Lansong Diao, and Wei Lin. 2021. DISC: A Dynamic Shape

Compiler for Machine Learning Workloads. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 89–95.