# MonoNN: Enabling a New Monolithic Optimization Space for Neural Network Inference Tasks on Modern GPU-Centric Architectures

Donglin Zhuang, *The University of Sydney;* Zhen Zheng, *Alibaba Group;* Haojun Xia, *The University of Sydney;* Xiafei Qiu, Junjie Bai, and Wei Lin, *Alibaba Group;* Shuaiwen Leon Song, *The University of Sydney*

https://www.usenix.org/conference/osdi24/presentation/zhuang

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

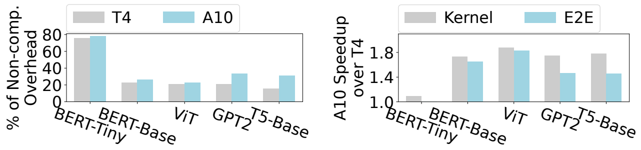
978-1-939133-40-3

# MonoNN: Enabling a New Monolithic Optimization Space for Neural Network Inference Tasks on Modern GPU-Centric Architectures

Donglin Zhuang [†*◇], Zhen Zheng [‡*], Haojun Xia [†◇], Xiafei Qiu [‡], Junjie Bai [‡], Wei Lin [‡]
Shuaiwen Leon Song [†]
[†]*The University of Sydney* [‡]*Alibaba Group*

## Abstract

In this work, we reveal that the kernel-by-kernel execution scheme in the existing machine learning optimizing compilers is no longer effective in fully utilizing hardware resources provided by the advances of modern GPU architectures. Specifically, such scheme suffers from severe non-computation overhead and off-chip memory traffic, making the optimization efforts from the state-of-the-art compiler techniques greatly attenuated on the newer generations of GPUs. To address this emerging challenge, we propose *MonoNN*, the first machine learning optimizing compiler that enables a new monolithic design and optimization space for common static neural network (NN) inference tasks on a single GPU. MonoNN can accommodate an entire neural network into a single GPU kernel, drastically reducing non-computation overhead and providing further fine-grained optimization opportunities from the newly formed monolithic optimization space. Most importantly, MonoNN identifies the resource incompatibility issue between various NN operators as the key design bottleneck for creating such a monolithic optimization space. Then MonoNN effectively tackles it by systematically exploring and exploiting the parallelism compensation strategy and resource trade-offs across different types of NN computations, and by proposing a novel schedule-independent group tuning technique to significantly shrink the extremely large tuning space. Finally, MonoNN provides a compiler implementation that incorporates our proposed optimizations and automatically generates highly efficient kernel code. Extensive evaluation on a set of popular production inference tasks demonstrates that MonoNN achieves an average speedup of 2.01× over the state-of-the-art frameworks and compilers. Specifically, MonoNN outperforms TVM, TensorRT, XLA, and AStitch by up to 7.3×, 5.9×, 1.7× *and* 2.9× in terms of end-to-end inference performance, respectively. MonoNN source code is publicly available at https://github.com/AlibabaResearch/mononn.



(a) Percentage of non-computation overhead on two generations of inference GPUs.

(b) GPU kernels-only speedup vs end-to-end (E2E) speedup by shifting hardware from T4 to A10.

Figure 1: Low hardware utilization for inference caused by growing non-computation overhead.

## 1 Introduction

In recent years, machine learning (ML) inference tasks have become one of real-world systems' most fundamental computation types. Existing optimization approaches [2, 7, 18, 24, 39, 41, 42] transform an ML computational graph into hundreds or thousands of computation kernels, and offload them onto high-performance AI accelerators, e.g., GPUs, for drastic latency reduction. However, with the increasing hardware advances of these complex GPUs on computation capability, the traditional kernel-by-kernel execution scheme is no longer effective in fully utilizing hardware resources.

Take XLA [2] as an example, which is one of the most popular and effective optimizers for ML workloads, Fig.1a shows the non-computation overheads (i.e., the end-to-end inference latency minus the pure kernel execution time on GPU) of five popular models on two generations of NVIDIA GPUs. Typically, the non-computation overhead mainly originates from frequent context switches between the host and GPU, e.g., framework scheduling and kernel launching. With the significant increase in computing power from T4 to A10, although the NN operators are executed faster, the non-computation

[◇] Work was done when interned at Alibaba.

[*] Equal contribution.

overheads tend to dominate the end-to-end performance, As illustrated in Fig.1b, the achieved end-to-end performance speedup can be far less than the kernel execution speedup when shifting across generations of hardware.

Moreover, the recent increase in computing power remains faster than that of memory bandwidth in recent generations of GPUs [42], making off-chip memory traffic among different GPU kernels within a model a significant performance bottleneck. Furthermore, there is a common scenario that often occurs in real-world systems and exacerbates the situation: CPUs are usually busy with data pre-/post-processing for real-time ML tasks, causing further delays in scheduling and launching their GPU kernels and subsequently increasing the non-computation overhead. To the best of our knowledge, although the kernel fusion scope and the corresponding techniques might be different, all the existing ML compilers [7, 18, 24, 39, 41, 42] suffer from performance issues discussed above due to the fundamental kernel-by-kernel execution scheme. Therefore, there is an urgent demand for a general solution with minimal non-computation overhead that can be widely applied to common ML inference tasks.

In this paper, we make a key observation that there exists *a monolithic design and optimization space* accommodating a wide spectrum of prevalent static DNN models in single GPU inference (Sec.3). MonoNN keeps the computation flow of the entire neural network on the GPU side without going back to the host to seek scheduling control. Such a scheme effectively avoids the non-computation overhead caused by the CPU-GPU context switch. With the structure of modern static DNNs consisting of repetitive layers, it would be more justified to aggressively enlarge the fusion scope, even result in a single kernel[1].

However, it is non-trivial to provide a general optimization scheme to consolidate all types of computations of an entire neural network into a *monolithic kernel*, while guaranteeing high performance and providing further fine-grained optimization opportunities from the newly formed monolithic optimization space. We observe that the main difficulty in forming such an optimization space is resource incompatibility between different types of neural network computations. On the one hand, a resource configuration that favors some operators can lead to a dramatic drop in performance on some other operators (e.g., low thread-level parallelism for GEMM computation is inefficient for element-wise operators). On the other hand, the resource configuration (e.g., parallelism configuration, register, and shared memory allocation) is fixed during the lifetime of a GPU kernel. Failure in reconciling such resource incompatibility in a monolithic kernel will result in poor performance. Furthermore, accommodating all operators of a complete NN into one GPU kernel results in an extremely large optimization space, making it very difficult for performance tuning on the whole computation graph.

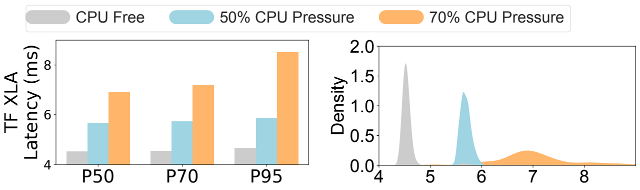
To address these emerging problems, we propose MonoNN, an ML optimizing compiler that enables a new monolithic optimization space for common NN inference tasks on modern GPU-centric architectures. Specifically, to address the significant resource incompatibility issue and accommodate the different resource requirements from various operators, we propose a *context-aware instruction rescheduling* technique (Sec.4.2.2). The key insight is to exploit the hidden instruction-level parallelism (ILP) for memory-intensive computations (e.g., element-wise, reduction) to compensate for the loss of the thread-level parallelism (TLP) under the monolithic kernel context. To further accelerate memory access, MonoNN classifies the memory access patterns into *streaming* and *temporal*, and comprehensively exploits all types of on-chip memory resources for the access patterns accordingly (Sec.4.3). It further exploits whole-graph level transformation inside the kernel to rearrange independent subgraphs together to reduce global thread barrier overhead (Sec.4.4). Finally, we systematically abstract the optimization space of the monolithic kernel and propose a *schedule-independent group tuning* approach to drastically compress the tuning space (Sec.5). Extensive evaluation on a set of neural network inference tasks demonstrates that MonoNN outperforms the state-of-the-art optimizers with an average of 2.01× speedup. Specifically, MonoNN outperforms TVM, TensorRT, XLA, and AStitch by up to 7.3×*,* 5.9×*,* 1.7× *and* 2.9× in end-to-end inference performance. To summarize, this work makes the following contributions:

- To the best of our knowledge, MonoNN is the first ML optimizing compiler that discovers a new monolithic optimization space for common static DNNs' inference scenarios that are served on a single GPU, and provides automatic high-performance kernel generation. This is also the first study that explores and evaluates this monolithic optimization design space and its limitations so that the community has a better understanding of the tradeoffs;
- It is the first optimizing compiler that explicitly exploits instruction-level parallelism optimization for memory-intensive operators to compensate for thread-level parallelism loss, enabling a new optimization dimension for neural network inference optimization;
- MonoNN enables a sophisticated compression mechanism to significantly shrink the tuning space for our proposed monolithic NN kernel;
- Extensive evaluation results have demonstrated the effectiveness of MonoNN on both single inference tasks as well as multi-inference processing scenarios.

## 2 Background and Motivation

**Emerging Challenges in Optimizing NN Inference.** From an optimization perspective, operators in neural network (NN) models can be classified into two categories, compute-intensive operators and memory-intensive opera-

[1] We also present a study on the fusion granularity under the monolithic optimization space in Sec.7.3



(a) Tail latency. (b) Latency distribution (ms).

Figure 2: T5 model latency statistics under each CPU pressure. The input sequence length is 128. (a): P50/P70/P90 tail latency. (b): latency distribution.

tors. Compute-intensive operators are typically composed of heavy arithmetic computations (e.g., GEMM and Conv), while memory-intensive operators are typically bounded by memory bandwidth (e.g., element-wise and reduction operations). Note that previous studies [18, 41, 43] have concluded both types of operators can dominate the execution time.

With the rapid growth of computing power for recent GPU generations[2], the execution time of compute-intensive operators decreases drastically. For example, Tensor Core brings an order of magnitude improvement in arithmetic unit throughput for compute-intensive operators since NVIDIA Volta architecture [3] (similarly, Matrix Core was also introduced in AMD GPUs since CDNA architecture [8]). However, there exists a disproportionate performance gain between hardware throughput improvement and end-to-end inference speedup. For instance, for the two common inference GPUs, NVIDIA A10 GPU has 1.9× more half-precision floating point throughput than its predecessor NVIDIA T4, while we only observe a 1.6× end-to-end inference speedup for the BERT model[3] [19] with XLA compiler optimization enabled [2]. Furthermore, we identified two emerging fundamental difficulties in optimizing inference scenarios on increasingly advanced modern GPUs:

***(i) Continuous advances in computation throughput leads to an increasing portion of non-computation overhead.***

Faster GPUs can offer shorter per-kernel execution in NN inference. However, the major portion of performance gains from hardware speed improvements for regular-size models begins to diminish as non-computation overhead becomes a notable portion of end-to-end latency. This new bottleneck mainly originates from frequent non-computation overhead which includes (1) context switch between host and GPU accelerator due to framework scheduling and kernel launch, and (2) off-chip memory traffic between operators.

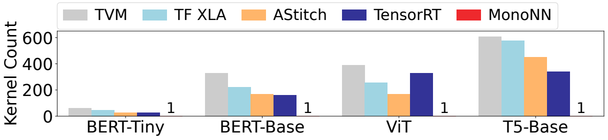


Figure 3: Number of inference GPU kernels for existing frameworks and MonoNN (1).

As the breakdown of the overall context switch overhead in Fig.1a, our measurements indicate that the framework scheduling accounts for 38.3% while the kernel launching overhead accounts for around 61.7% (see Sec.7.2.5 for more details). As for off-chip memory traffic, the memory bandwidth growth across hardware generations is generally slower than that for arithmetic throughput. Fig.1a demonstrates the non-computation inference overhead via XLA optimizations for five common models. It is worth noting that A10 suffers from more severe non-computation inference overhead than its predecessor T4 as newer generations of GPUs have much shorter per-kernel duration. Fig.1b illustrates that there is an average of 1.64× kernel execution speedup benefiting from shifting the underlying accelerator from T4 to A10. Unfortunately, such speedup decreases to 1.48× for the end-to-end latency as the non-computation overhead is not the highest optimization priority for the existing inference engines. Thus, the non-computation inference overhead for neural network models is becoming increasingly essential for the next generations of faster GPU hardware [6].

***(ii) Ever-present, non-negligible CPU workloads exacerbate non-computation overhead.***

Moreover, a commonly neglected factor is that CPU is usually busy with pre- and post-processing of input and output for NN tasks in real-world execution. Thus, CPU contention often further delays the scheduling and kernel launching of a large number of GPU kernels within a model execution. This further exacerbates the CPU-GPU context switch overhead and makes it a much more severe problem, causing an additional slowdown of model inference tail latency. In Fig.2a, when measuring the tail latency under XLA optimizations on a server with a 64-core CPU (128 threads) and an NVIDIA A10 GPU under 50% (70%) CPU utilization, the tail latency increases by 25% (52%), 26% (58%), and 26% (82%) at P50, P70, and P95, respectively, over the latency of an idle CPU. Fig.2b shows a detailed inference latency distribution of 1000 times of inference when CPU is under various utilization. With the increasing CPU contention, the end-to-end inference latency belongs to a wider range of much slower outliers. Note that it is impractical to designate a specific CPU core exclusively just for kernel launching in the datacenter because the CPUs are typically very busy performing pre-/post-processing. Besides, designating such a core requires a hardcoded list of CPU cores to be isolated from the default CPU scheduler in the system boot phase, resulting in rebooting for every new

[2] In this work, we focus our discussion on the most widely-adopted general-purpose AI accelerators: GPUs. Although the technical terminologies used in this paper are adopted from NVIDIA GPUs [4, 5], our proposed techniques aim to serve as general principles that are valuable for modern general-purpose machine learning system designs, and are applicable to other SIMT accelerators [8].

[3] Data is collected under TensorFlow XLA v2.7 with Tensor Core enabled, using 1 as the batch size and 128 as the sequence length.

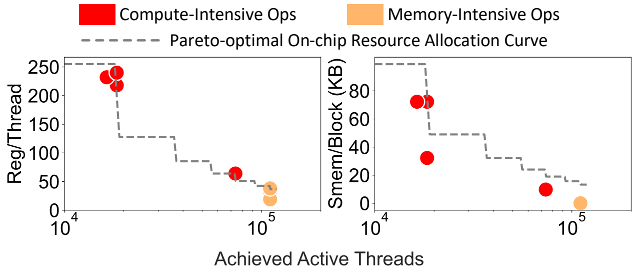


Figure 4: Different resource requirements between compute-intensive operators and memory-intensive operators for TensorRT BERT inference. Each data point may represent multiple GPU kernels with similar resource usage within a model.

inference service deployment.

Challenges in the State-of-The-Art Designs. TVM [18] applies a basic fusion strategy but still unnecessarily launches a large number of kernels. Some recent works propose more advanced fusion techniques to alleviate the problems above. AStitch [41, 42] leverages hierarchical GPU memory to fuse multiple memory-intensive operators with complex data dependencies into a single GPU kernel, named stitch optimization. TensorRT [11] also exploits a similar strategy since v8.

Although it helps reduce the kernel number to some extent, it still results in a large number of kernels since it is not capable to fuse globally along with all the compute-intensive operators, for which the bottlenecks that we discussed above still exist. As illustrated in Fig.3, TVM, XLA, TensorRT, and AStitch are all launching a large number of kernels during model inference. Furthermore, Rammer [24] partially addresses this problem with a persistent-thread technique [14, 17] to generate the schedule of multiple operators within one kernel. However, Rammer is incapable of handling the resource incompatibility between different operators in an entire neural network (see Sec.3.1). As a result, Rammer has to partition the neural network into separate GPU kernels for NN inference. For example, Rammer still launches `734` kernels on GPU for BERT-Large [19] model inference.

## 3 Monolithic Optimization Space

To address the emerging challenges discussed in Sec.2, we explore the *monolithic optimization space* where the entire neural network can be compiled into a single GPU kernel. This approach is appealing because it only incurs minimal non-computation overhead and enables the opportunities for whole graph optimization within the same kernel space. However, a general approach enabling this optimization space is non-trivial, especially when handling various NN models with very different execution patterns. Here we summarize two major challenges to auto-generate a highly-efficient GPU kernel containing all the operators of a given neural network.

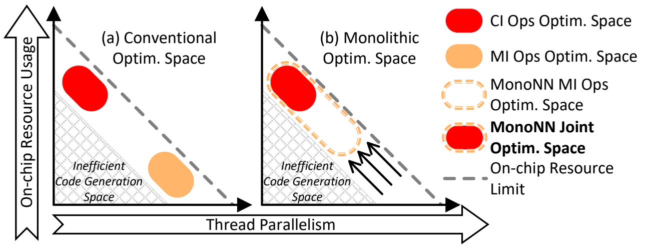

On-chip Resource Usage
(a) Conventional Optim. Space
(b) Monolithic Optim. Space
Inefficient Code Generation Space
Inefficient Code Generation Space
Thread Parallelism
CI Ops Optim. Space
MI Ops Optim. Space
MonoNN MI Ops Optim. Space
MonoNN Joint Optim. Space
On-chip Resource Limit

Figure 5: Optimization space comparison.

### 3.1 Main Challenges of Enabling A Monolithic Kernel Optimization Space

***Challenge 1: Resource incompatibility between compute-intensive and memory-intensive operators.*** The resource incompatibility between compute-intensive and memory-intensive operators hinders the state-of-the-art techniques to consolidate all operators into a monolithic kernel. Compute-intensive operators usually require a large amount of on-chip resources (e.g., registers and shared memory) whereas memory-intensive ops rely on massive concurrent threads to hide off-chip memory access. Thus, *it is extremely difficult to accommodate all types of operators by creating a GPU kernel with both high on-chip usage and massive concurrent threads due to the resource constraints on modern GPUs*. For example, the active TLP on an SM core will inevitably drop when a kernel uses a large number of registers and shared memory due to the limited on-chip resources. We illustrate this phenomenon quantitatively using GPU kernels from a TensorRT optimized BERT [19] and the Pareto-optimal on-chip resource allocation curve on an NVIDIA A10 GPU in Fig.4. Compute-intensive kernels in NN models tend to be closer to the upper-left corner, representing high on-chip resource allocation and relatively low achieved concurrently active threads. In contrast, memory-intensive kernels tend to be closer to the bottom-right corner, representing low on-chip resource allocation and massive concurrently active threads (or high TLP). All data points in Fig.4 are subject to resource constraints and thus will not be above the Pareto-optimal curve.

***Challenge 2: Extremely high implementation cost and huge tuning space.*** Modern ML models usually consist of thousands of operators with diverse computation patterns, resulting in intricate data dependencies. Manual implementation and optimization are no longer viable for developing a monolithic kernel. In terms of compiler optimization, the monolithic kernel approach significantly expands the optimization search space as all the operators coexist within the same kernel. Consequently, it becomes exceedingly challenging to identify suitable configurations and implementations for each operator to achieve optimal end-to-end inference efficiency within a monolithic kernel collectively.

## 3.2 A High-level Glance of MonoNN

The ultimate objective of MonoNN is to create an efficient joint optimization space for both compute-intensive operators (*CI Ops*) and memory-intensive operators (*MI Ops*). However, as elaborated in Fig.4 and Sec.3.1, these two types of operators naturally reside in disjoint optimization space in conventional solutions due to resource incompatibility. We conceptually illustrate this observation in Fig.5(a).

MonoNN enables a new monolithic optimization space that can effectively accommodate both *CI Ops* and *MI Ops*. The key idea is to align the optimization space of *MI Ops* as closely as possible with that of *CI Ops* (Fig.5(b)). Specifically, MonoNN leverages the hidden instruction-level parallelism (ILP) to offset the reduction in TLP for memory-intensive subgraphs, thereby achieving a similar resource allocation to *CI Ops* (Sec.4.2.2). Furthermore, MonoNN strategically utilizes abundant on-chip resources, including registers, shared memory, and cache, to buffer and prefetch off-chip data based on an analysis of memory access patterns (Sec.4.3). This approach enables both *CI Ops* and *MI Ops* to coexist within the same monolithic kernel efficiently. Additionally, MonoNN explores global optimization opportunities to minimize global synchronizations between computations, further enhancing the efficiency of neural network models (Sec.4.4).

# 4 System Design

## 4.1 Overview of MonoNN

Fig.6 illustrates the overview of MonoNN. MonoNN first formulates the input neural network into different subgraphs for subsequent optimizations (Fig.6(1), Sec.4.2.1). Then, MonoNN enables the hidden parallelism of memory-intensive subgraphs through context-aware instruction rescheduling (Fig.6(2), Sec.4.2.2), and comprehensively optimizes the usage of various on-chip resources according to memory access patterns (Fig.6(3), Sec.4.3). Next, it reorders and clusters subgraphs to reduce the required Global Thread Barriers (*GTBs*) to minimize the synchronization overhead (Fig.6(4), Sec.4.4). Finally, MonoNN abstracts, compresses, and tunes for the large monolithic optimization space with *schedule-independent group tuning*, and compiles the monolithic kernel into an executable binary (Fig.6(5)-(6), Sec.5).

## 4.2 Exploiting Hidden Parallelism for Memory-intensive Subgraphs

We address the resource incompatibility issue discussed in Sec.3 for memory-intensive computations with *context-aware instruction rescheduling* (the compute-intensive computations will be discussed in Sec.4.5.) MonoNN performs instruction rescheduling under the context of *monolithic optimization space* to recover potential TLP loss for memory-intensive computations with high-level instruction-level parallelism (ILP) enhancement. Thereby, MonoNN fully leverages the abundant registers under the new monolithic optimization space to unleash the hidden potential of reaching high performance.

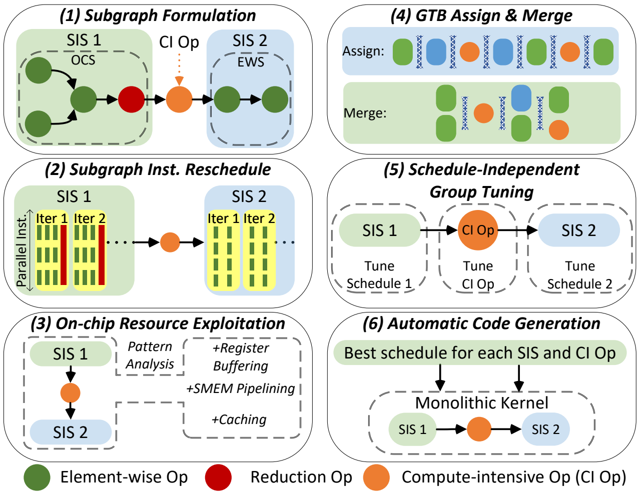


Figure 6: MonoNN overview. *SIS*: Schedule-independent subgraph. *EWS*: Element-wise subgraph. *OCS*: Output-only contracted subgraph.

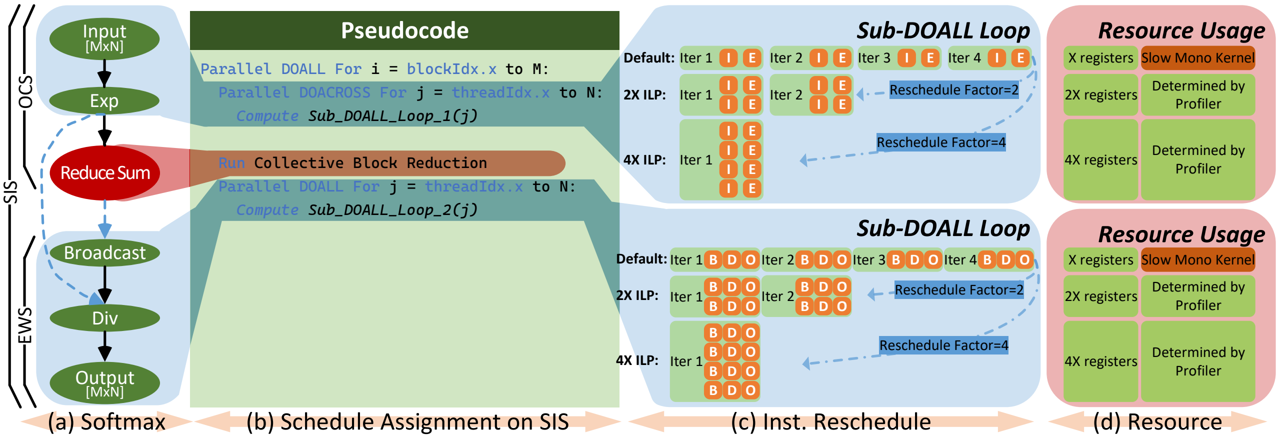
### 4.2.1 Memory-intensive Subgraph Formulation

Before unveiling the details of *context-aware instruction rescheduling*, we first present how subgraphs are formulated as the basic units of optimization exploration. MonoNN converts the whole graph of a model into one kernel. Instead of optimizing and generating the code of the whole graph all in one shot, MonoNN generates the schedules[4] of different partitions (i.e., subgraphs) of the graph separately under the same monolithic context, and then stitches them together with shared memory or global memory data buffering.

***Subgraph Formulation.*** The compute-intensive operators divide the whole computation graph into a set of memory-intensive subgraphs. We use the following criterion to further categorize memory-intensive subgraphs based on data dependencies between data elements in input and output tensors.

Formally, for a subgraph with $m$ input tensors $[\mathcal{X}_0, \mathcal{X}_1, \cdots \mathcal{X}_{m-1}]$ and $n$ output tensors $[\mathcal{Y}_0, \mathcal{Y}_1, \cdots \mathcal{Y}_{n-1}]$, the computation of the subgraph is: $[\mathcal{Y}_0, \mathcal{Y}_1, \cdots \mathcal{Y}_{n-1}] = f([\mathcal{X}_0, \mathcal{X}_1, \cdots \mathcal{X}_{m-1}])$. If each pair of $\mathcal{X}_i \in [\mathcal{X}_0, \mathcal{X}_1, \cdots \mathcal{X}_{m-1}]$ and $\mathcal{Y}_i \in [\mathcal{Y}_0, \mathcal{Y}_1, \cdots \mathcal{Y}_{n-1}]$ that $\frac{\partial \mathcal{Y}_i}{\partial \mathcal{X}_i} \neq 0$ satisfies $\forall ey \in \mathcal{Y}_i, \left|\{ex | \frac{\partial ey}{\partial ex} \neq 0, ex \in \mathcal{X}_i\}\right| \leq 1$, in which $ex$ represents a scalar data element in tensor $\mathcal{X}_i$ and $ey$ represents a scalar data element in tensor $\mathcal{Y}_i$. It indicates that all data elements in any of the output tensors rely on at most one data element in one input tensor. We call a subgraph with such property as an *element-wise subgraph* (**or EWS**).

[4] In code generation, *schedule* means how the threads are mapped to hardware to process the data (e.g., tiling size, on-chip resource configuration, parallelism configuration for GEMM code generation).



```
Parallel DOALL For i = blockIdx.x to M:
  Parallel DOACROSS For j = threadIdx.x to N:
    Compute Sub_DOALL_Loop_1(j)
  Run Collective Block Reduction
  Parallel DOALL For j = threadIdx.x to N:
    Compute Sub_DOALL_Loop_2(j)
```

Figure 7: Context-aware instruction rescheduling for `softmax` computation.

Otherwise, if there exists an output tensor element that relies on multiple input tensor elements, the subgraph contains contraction operations that combine several data elements into one (one-on-many element-level data dependency). We refer to such a subgraph as a *contracted subgraph* (**CS**). In machine learning graphs, contractions are often represented by `reduce` operations (e.g., reduce-sum) in the intermediate representation (IR). If all the `reduce` ops of a subgraph are the output operations, we call the subgraph an *output-only contracted subgraph* (**OCS**). Note that a *CS* is either an *OCS* or could be decomposed into *OCS* and *EWS*.

***Basic Codegen Scheme.*** MonoNN will first identify all the largest *OCS* through reverse traversal on the memory-intensive subgraphs. The remaining subgraphs are then *EWS*, which can be converted into DOALL loop [13] (i.e., loop with no inter-iteration dependency) for full parallelization.

An *OCS* contains the contraction computation in `reduce` op. The `reduce` ops in typical inference graphs are doing contraction over elements residing in a continuous address in memory (e.g., $Reduce([x,y]) => [x,1]$). For `reduce` ops on GPU, the non-contracted dimension (e.g., $x$ in the above example) forms a DOALL loop *without* inter-iteration dependency. Whereas the inner contracted dimension (e.g., $y$ in the above example) forms a DOACROSS loop [13] *with* inter-iteration dependency due to contraction computation. Note that in some cases, it might be beneficial to use a uniform schedule for adjacent subgraphs with loop fusion if certain locality constraints are met. For example, in Fig.7(a), an *OCS* followed by an *EWS* can use a uniformed schedule by fusing the outer loop, as the output of *OCS* can buffer on on-chip cache for subsequent read from *EWS*. This technique, also known as stitch fusion [42], is shown as dotted lines in Fig.7(a). We call subgraphs that have independent schedule *schedule-independent subgraphs*, *SIS* in short. Several subgraphs that use a uniform schedule after loop fusion are regarded as one *SIS* (e.g., Fig.7(b) shows an *SIS* after fusing an *OCS* and an *EWS*). The schedule within an *SIS* is constrained by loop structure and block locality, while the schedules among different *SIS* are independent. Different *SIS* with its own schedule is finally stitched together under MonoNN with global memory buffering for intermediate transferring.

### 4.2.2 Context-Aware Instruction Rescheduling

We illustrate how to enable the hidden parallelism given an *SIS* subgraph with the example in Fig.7(a). Note that the contracted dimension of the `reduce` op is *N*, which maps to the inner loop (i.e., parallel threads within a thread block), The non-contracted dimension is *M*, which maps to the outer loop (i.e., different thread blocks).

According to the property of *OCS*, it can be divided into a sub-*EWS* followed by a `reduce` op. Thus, the inner loop of *OCS*, which is a DOACROSS loop, can be converted to a sub-DOALL loop (Fig.7(c)) followed by the corresponding reduction. With the conversion above, the inner-loop of the *SIS* is converted to the computation sequence of "*sub-DOALL loop* ⇒ *reduction* ⇒ *sub-DOALL loop*" (Fig.7(b)-(c)).

The key insight of context-aware instruction rescheduling is to rearrange the instructions according to the property of the DOALL loop.

In Fig.7(c), the DOALL loop has no inter-iteration dependencies, allowing MonoNN to explore the default schedule as well as the ILP enhanced schedules (e.g., 2*X* ILP) by merging instructions from different iterations into parallel instructions within the same iteration to ensure stall-free instruction issue. Theoretically, all schedules shown in Fig.7(c) achieve near-maximum overall parallelization ($TLP \times ILP$), with the default schedule yielding varied TLP for different operators, thus necessitating numerous GPU kernels. In contrast, MonoNN can identify a schedule that maximizes overall parallelization and optimally fits TLP into a single monolithic kernel. The optimization space for utilization abundant registers (Fig.7(d)) and corresponding performance in the monolithic kernel will be explored, as determining the best scheduling factor involves balancing ILP and resource usage. We will discuss

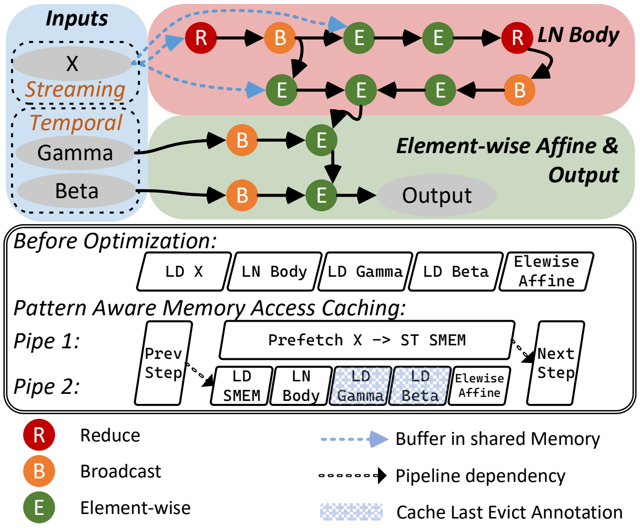


Figure 8: On-chip resource exploitation for layer norm.

how to find the optimal rescheduling factor for each *SIS* in Sec.5

## 4.3 On-Chip Resource Exploitation

For an SIS memory-intensive subgraph, global memory access often takes up a significant amount of execution time, particularly when thread-level parallelism (TLP) is limited in a monolithic context. Along with boosting parallelism through instruction rescheduling as discussed in Sec.4.2.2, MonoNN performs a comprehensive on-chip memory resource exploitation to maximize the use of memory resources based on access patterns.

We observe that there are two major memory access patterns for an *SIS* subgraph: (1) *Streaming*: Each element of the input tensor is accessed once in the computation graph. (2) *Temporal*, Each data item in the input tensor is read multiple times by its consumers, commonly due to `broadcast` operators in modern ML models. MonoNN implements a series of memory access optimizations based on these patterns.

***(I) Streaming Access Optimization.*** MonoNN leverages the abundant shared memory resource allocated by the compute-intensive computations in the monolithic kernel to pipeline the streaming global memory access with other computations. As mentioned, the outer loop of an *SIS* subgraph is a DOALL loop, where different iterations are independent. MonoNN organizes the computations between different iterations of the outer loops to form a computation pipeline and a memory copy pipeline. Particularly, during the computation of each outer loop iteration, it will prefetch the streaming accessed input data for the next iteration into the shared memory buffer. Fig.8 presents the input data access pipelining for layer norm [15]. *LN Body* represents the main layer norm computation and *Element-wise affine* represents the following element-wise affine transformation parameterized by *Gamma* and *Beta*. The memory access to input *X* is prefetched onto shared memory in the computation pipeline, fully overlapping the data fetching and computation. Note that the input *X* in Fig.8 is consumed by multiple operators. If the shared memory is not enough for the data buffering, MonoNN will not make a pipelined buffer *X*. Instead, MonoNN will buffer *X* on the register file (or local memory if facing register spills), for which the multiple consumers will reuse the data through the faster register file rather than the global memory.

***(II) Temporal Access Optimization.*** If the input data access is temporal rather than streaming, MonoNN will annotate cache hints to these memory operations to guide the cache behavior to preserve the data on the cache as long as possible (e.g., *evict_last* in NVIDIA GPU semantics). MonoNN will annotate memory read as temporal access from the node with a smaller tensor shape until an empirical value is reached to accommodate as many tensors as possible and prevent cache thrashing. As shown in Fig.8, *Gamma* and *Beta* have temporal locality because they are connected to the subsequent `broadcast` op. A load of *Gamma* and *Beta* will be annotated with *evict_last* for longer cache occupation, improving the temporal locality in *SIS*.

## 4.4 Global Thread Barrier Merging

As mentioned in Sec.4.2, there are cross thread block data dependencies between different *SIS* subgraphs. MonoNN inserts global thread barrier (**GTB**) between *SIS* subgraphs to ensure correctness. Note that *GTBs* are also required between compute-intensive operators and *SIS* subgraphs. Similar with [42], *GTB* in MonoNN is implemented in two stages: one-block-wait-all and one-block-notify-all. Each block has a flag in global memory (typically cached in GPU L2) to represent whether the corresponding thread block has arrived. The first thread block waits for all the remaining blocks to report waiting, and then notifies them to proceed. Furthermore, the inner-kernel *GTB* is much shorter than the non-computation overhead as the latter is composed of both kernel launching and framework scheduling overheads. The overhead measurement results of inner-kernel *GTB* [42] and kernel launching [37] from the previous studies are aligned with our observation that a typical kernel launching overhead is often multiples of a *GTB* length, e.g., a single kernel launching with framework scheduling is around $8 \sim 10$ microseconds which is $4 \sim 5\times$ of a *GTB* length.

***Longest-path based GTB merging.*** One *GTB* introduces minimal synchronization overhead, but this can accumulate when the number of *GTBs* is large. We have observed that some *SIS* subgraphs do not exhibit producer-consumer or topology dependencies. By clustering these independent *SIS* subgraphs in topological order, MonoNN can eliminate the need for *GTBs* between them. To address graph complexity, we propose the *longest-path based GTB merging* approach to find the optimal *SIS* clustering strategy. For example, in Fig.9, the nodes (A-E) represent the *SIS* subgraphs, with *GTBs*

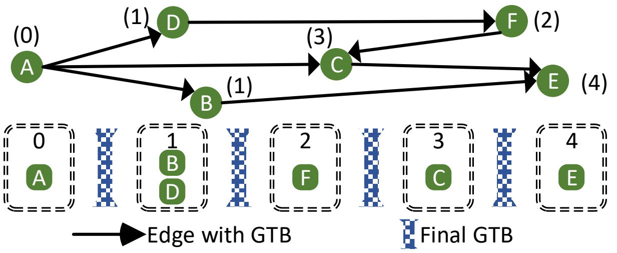


Figure 9: Global thread barrier merging. The numbers above nodes represent the longest distance from the source node to the current node.

required between them (indicated by edges). MonoNN calculates the longest path to each node from the first node. Nodes with the same longest path length (e.g., B and D in Fig.9) can be clustered together for *GTB* merging. Traditional topology ordering methods only order nodes and do not cluster them, making them inadequate for guiding *GTB* merging.

### 4.5 Optimizing Compute-Intensive Operators

While the memory-intensive subgraphs are effectively optimized to maximally leverage on-chip resources, the compute-intensive operators in MonoNN directly adopt the existing implementations from CUTLASS [1] as tunable basic building blocks: the tuning space of CUTLASS is included in the tuning space for the monolithic kernel.

## 5 The MonoNN Compiler

This section details the design and implementation of an optimizing compiler that automatically generates efficient monolithic kernels using the techniques outlined in Sec.4. Unlike previous works that focus on tuning single operators or subgraphs [18,42], MonoNN optimizes the entire graph, resulting in a vast optimization space. This complexity makes finding the optimal global configuration challenging. We explain how we systematically abstract this extensive optimization space in Sec.5.1 and how we reduce it to efficiently identify a suitable global configuration in Sec.5.2.

### 5.1 Optimization Space Abstraction

First, we categorize the proposed optimizations into two types: (1) Deterministic optimizations are always beneficial. Including *comprehensive on-chip resource exploitation* (Sec.4.3) and *global thread barrier merging* (Sec.4.4). (2) Tunable optimizations: All other optimizations not included in the deterministic category are considered tunable. We classify the tunable factors of the monolithic kernel into three main classes:

***(I) Code generation schedule of each operator in a neural network.*** In the code generation process, element-wise operators follow the code generation schedule of `reduce` operators through input-inline. Thus, we only need to tune the schedule of `reduce` operators and compute-intensive operators (*CI Ops*). Note that a grid-stride loop will be used to iterate over its input elements if an element-wise operator cannot find a `reduce` or *CI Ops* that it associates with. There are two common schedules for row-major `reduce` operators. One is to reduce a row of elements with all threads in a thread block. The other one is to reduce a row with one warp. For *CI Ops*, MonoNN will jointly consider all the tunable factors, including tiling size, on-chip resource configuration, parallelism configuration, hardware intrinsic (e.g., Tensor Core instruction and CUDA async-copy), input prefetching, etc.

***(II) Context-aware instruction rescheduling factor.*** ILP is important for the *SIS* subgraphs to compensate for parallelism loss under the constraint TLP in a monolithic kernel. A too-small rescheduling factor may be insufficient to improve the overall parallelism. A rescheduling factor that is too large will use massive registers and may cause register spilling. MonoNN explores a spectrum of the rescheduling factors for each memory-intensive operator (*MI Op*). Specifically, for each *MI Op*, MonoNN explores up to 32*X* rescheduling factors [5] via *context-aware instruction rescheduling*. In Sec.7.2.1, we quantitatively evaluate how different ILP rescheduling factors impact *MI Ops* on performance.

***(III) TLP and on-chip resource of the overall monolithic kernel.*** TLP on GPUs is defined as the thread block size and number of blocks for a GPU kernel, and on-chip resource constraints are critical performance factors for efficient program execution. For the monolithic kernel, these factors not only affect the optimal execution configuration (e.g., tiling size) for *CI ops*, but also impact the optimal rescheduling for *context-aware instruction rescheduling* and optimal code generation schedule for memory-intensive subgraphs (e.g., warp reduction vs block reduction). The candidate block sizes for tuning are 128 and 256 for MonoNN, which are the main block sizes used in the existing machine learning compilers [2, 7] and CUTLASS for achieving good performance for both *CI Ops* and *MI Ops*. Other block sizes may also be trivially included to the optimization space. Our monolithic kernel requires that all thread blocks be able to be scheduled onto GPU concurrently in one wave to avoid deadlock in synchronization. Thus, the total thread block number should be no more than the max number of thread blocks that GPU can tolerate. Specifically, the number of candidate TLP choices is $N_{TLP} = \left|\{128,256\}\right| \times N_{blocks-per-sm} = 2 \times N_{blocks-per-sm}$, where $N_{blocks-per-sm} = \left|\{1,2,\ldots,N_{max-blocks-per-sm}\}\right|$ We empirically choose $N_{max-blocks-per-sm}$ as 5 because too many co-existing thread blocks will result in insufficient available on-chip cache per block and further slow down *CI Ops*.

[5] The range of ILP is constrained by on-chip resources and thus is up to 32 for hardware we evaluated.

## 5.2 Schedule-Independent Group Tuning

### 5.2.1 Extremely Large Tuning Space

The tuning complexity on the optimization space is up to: $O_{naive} = (S_C)^{N_C} \times (S_M \times N_{ILP})^{N_M} \times N_{TLP}$.[6] Indicates the code generation schedules for *CI ops* ($(S_C)^{N_C}$), schedules and ILP sizes for *SIS* ($(S_M \times N_{ILP})^{N_M}$), and candidate TLP sizes for the monolithic kernel ($N_{TLP}$). All operators in an *SIS* share the uniform schedule. If there exists a `reduce` operator in the *SIS*, we only need to enumerate the schedule of one reduce operator; otherwise, it will adopt a grid-stride loop as the schedule. A uniformed ILP will apply to all operators in the same *SIS* since all operators in the same *SIS* share similar resource and parallelism requirements. Unfortunately, this is an excessive tuning space and will have a size of approximately $10^{500}$ for a BERT-base model.

### 5.2.2 Tuning Space Compression

We make two important observations for the monolithic kernel. (1) A monolithic kernel is separated into a set of *SISs* and *CI Ops* by *GTBs*. The code generation schedules for different subgraphs are not interleaved. We call an *SIS* or a *CI Op* as a *schedule-independent group* (SIG). (2) The connection between two *schedule-independent groups* is the TLP and on-chip resource allocation. Meanwhile, the overall kernel's TLP and on-chip resource allocation are fixed throughout the monolithic kernel. According to the observations above, the code generation schedule of different *SIGs* can be safely tuned individually without missing the optimal solution.

Based on the above observations, we propose *schedule-independent group tuning* to compress the tuning space significantly. Different *SIGs* are tuned independently for each candidate TLP setting. Particularly, MonoNN concatenate the best-tuned configurations of all the *SIGs* to get the overall best configuration. Finally, we chose the TLP setting that performs the best and all its associated configurations.

For a *schedule-independent group* that is a *CI Op*, we enumerate $S_C$ code generation schedules. There are $N_C$ such groups, and the overall complexity is $N_C \times S_C$ under each overall TLP configuration. For a *schedule-independent group* that is an *SIS* subgraph, we enumerate the possible code generation schedules and overall ILP sizes. There are $N_M$ such groups, and the overall complexity is $N_M \times S_M \times N_{ILP}$ under each overall TLP configuration. As a result, the shrunken tuning complexity of our monolithic kernel is up to:

$$O_{opt} = (N_C \times S_C + N_M \times S_M \times N_{ILP}) \times N_{TLP}.$$

It is worth noting that MonoNN will check the *SIG* hash and reuse the tuning result if an identical *SIG* has been tuned previously. This will prevent duplicated tuning effort under repetitive neural network layers.

**Algorithm 1** Monolithic Kernel Tuning

```
 1: procedure GETTUNINGSPACETLP
 2:     $C_{block-size} \leftarrow \{128, 256\}$
 3:     $C_{blocks-per-sm} \leftarrow \{1, \ldots, N_{max-blocks-per-sm}\}$
 4:     return $C_{block-size}$ $X$ $C_{blocks-per-sm}$
 5: procedure OPTIMIZEMISIS(SIS, TLP)
 6:     $CandidateILPFactors \leftarrow \{1, 2, 3, \ldots, 32\}$
 7:     $BestSolution, BestTime \leftarrow NULL, \infty$
 8:     for $ILP \in CandidateILPFactors$ do
 9:         $S, Time \leftarrow ProfileAndOptimize(SIS, TLP, ILP)$
10:         if $Time < BestTime$ then
11:             $BestSolution, BestTime \leftarrow S, Time$
12:     return $BestSolution, BestTime$
13: procedure OPTIMIZEFORTLP(TLP)
14:     $Solution, TotalTime \leftarrow \{\}, 0$
15:     for $SIG \in GetAllSIG()$ do
16:         if $IsCIOp(SIG)$ then
17:             $S, Time \leftarrow ProfileAndOptimize(SIG, TLP)$
18:         else                                        ▷ Is MI SIS
19:             $S, Time \leftarrow OptimizeMiSIS(SIG, TLP)$
20:         $Solution \leftarrow Solution \cup \{S\}$
21:         $TotalTime \leftarrow TotalTime + Time$
22:     return $Solution, TotalTime$
23: procedure MONONNTUNE
24:     $BestSolution, BestTime \leftarrow NULL, \infty$
25:     for $TLP \in GetTuningSpaceTLP()$ do
26:         $S, Time \leftarrow OptimizeForTLP(TLP)$
27:         if $Time < BestTime$ then
28:             $BestSolution, BestTime \leftarrow S, Time$
29:     return $BestSolution, BestTime$
```

Algo.1 details the tuning procedure in MonoNN begins with sampling *MonoNNTune* in line 25. MonoNN takes Cartesian product between candidate block size $C_{block-size}$ and co-existing blocks per SM $C_{blocks-per-sm}$ (line 4). The optimal solution under each TLP will be tuned independently (line 26). MonoNN will optimize every *SIG* in the neural network (line 15). For memory-intensive subgraphs, the best solution across all rescheduling factors will be selected as the final solution of the current subgraph (line 10-12). MonoNN iterates over the solution under each distinct TLP and chooses the one with the shortest duration as the final solution (line 26-29).

## 5.3 Implementation

We implement MonoNN with 64k lines of C++ code on top of XLA compiler [2] and integrate it into TensorFlow [12] framework as a drop-in replacement to the backend execution engine. This allows MonoNN to accelerate existing TensorFlow models without requiring any code changes. Additionally, MonoNN can compile a neural network into a standalone assembly file that can be directly executed, potentially offering better performance by eliminating the runtime overhead from the deep learning framework. In Sec.7, we only report the performance number from the first mode for a fair comparison across frameworks. Unlike AStitch [42], MonoNN does not support cross-block reduction. We are not aware of any performance degradation on evaluated models as the

[6] $N_C$ (or $N_M$): number of *CI ops* (or *SIS*). $S_C$ (or $S_M$): possible schedules of *CI ops* (or SIS). $N_{ILP}$ (or $N_{TLP}$): possible ILP (or TLP) sizes.

reduction dimension for these popular models are all small.

## 6 Scope, Impact, and Limitations

The current optimization scope of MonoNN mainly focuses on general static DNNs where different layers have similar computational sizes. For example, MonoNN effectively supports various popular Transformer models without dynamic control flows, including Transformer encoder models such as BERT, encoder-decoder models such as T5, and every step of the decoder models such as GPT-like models. We compare the performance of different fusion granularity in Sec.7.3, ranging from basic element-wise fusion, stitch fusion, layer-wise monolithic kernel (i.e., one monolithic kernel per layer) to a single monolithic kernel of the entire model.

MonoNN established a new monolithic optimization space for common static DNN inference scenarios by resolving the long-existing global optimization challenge within a single kernel, addressing the resource incompatibility problem of various operators. MonoNN introduce key contributions such as *Context-Aware Instruction Rescheduling* (Sec.4.2.2), *On-Chip Resource Exploitation* (Sec.4.3), and *Global Thread Barrier Merging* (Sec.4.4). Moreover, MonoNN is designed to be forward-looking, performing even more effectively for the upcoming GPU architectures. The increased computing power of future GPUs will likely exacerbate issues related to off-chip memory access and CPU-GPU context switch overhead. Moreover, as supported by [6], distributed shared memory access can enable more flexible and efficient intermediate data buffering for large-scale operator fusion.

Despite the contribution of MonoNN, several potential limitations should be noted. (1) MonoNN mainly addresses common static DNN inference scenarios rather than the models with dynamic control flows [22, 36]. However, users can still optimize the subgraphs separated by control flow operators using MonoNN techniques. (2) MonoNN focuses on DNN inference scenarios that fit within a single GPU, covering a wide range of real-world inference service cases. Extending MonoNN to incorporate collective communication primitives is beyond the scope of this work, but users can still optimize the subgraphs separated by the communication operators using MonoNN. (3) MonoNN may be less effective for DNNs with varied tensor sizes in different layers due to the imbalanced workloads in the single monolithic kernel. While our experiments did not show significant performance regression, this potential limitation in the monolithic kernel should be highlighted.

## 7 Evaluation

***Model specifications:*** We use a set of representative machine learning applications as our evaluation workloads, including BERT-Base, BERT-Large [19], Transformer T5-Small, T5-Base [30] for natural language processing, ViT [20] for image recognition (with both Convolution and Transformer components), CLIP [29] for computer vision and text, OPT [38] for text generation (OPT-125M version). All the models are publicly available from Huggingface [34]. For all BERT-like and Transformer-like models, we used sequence length equal to 128 unless specified elsewhere.

***Software specifications:*** We compare MonoNN against TensorFlow [12] (v2.7), XLA [2] (v2.7), TensorRT v8.2[7] (via TF-TRT integration [7]), TVM (commit f6f9056) [18], AStitch [42], Rammer [24], PyTorch [28] (v1.12.1), and CUDAGraph [9] (via PyTorch integration). We use CUDA v11.6 and cuDNN 8 for all the experiments[8]. We enable Tensor Cores for all the frameworks we evaluated except in Sec.7.5 cause Rammer [24] only supports SIMT cores.

***Hardware Platforms:*** *A10 server*: NVIDIA A10 GPU (Ampere), and two Intel(R) Xeon(R) Platinum 8369B CPUs. *T4 server*: NVIDIA T4 GPU (Turing), and two Intel(R) Xeon(R) Platinum 8163 CPUs. *A100 server*: NVIDIA A100 80GB SXM (Ampere), and two Intel(R) Xeon(R) Platinum 8369B CPUs.

### 7.1 End-to-End Performance Comparison

#### 7.1.1 Overall Results

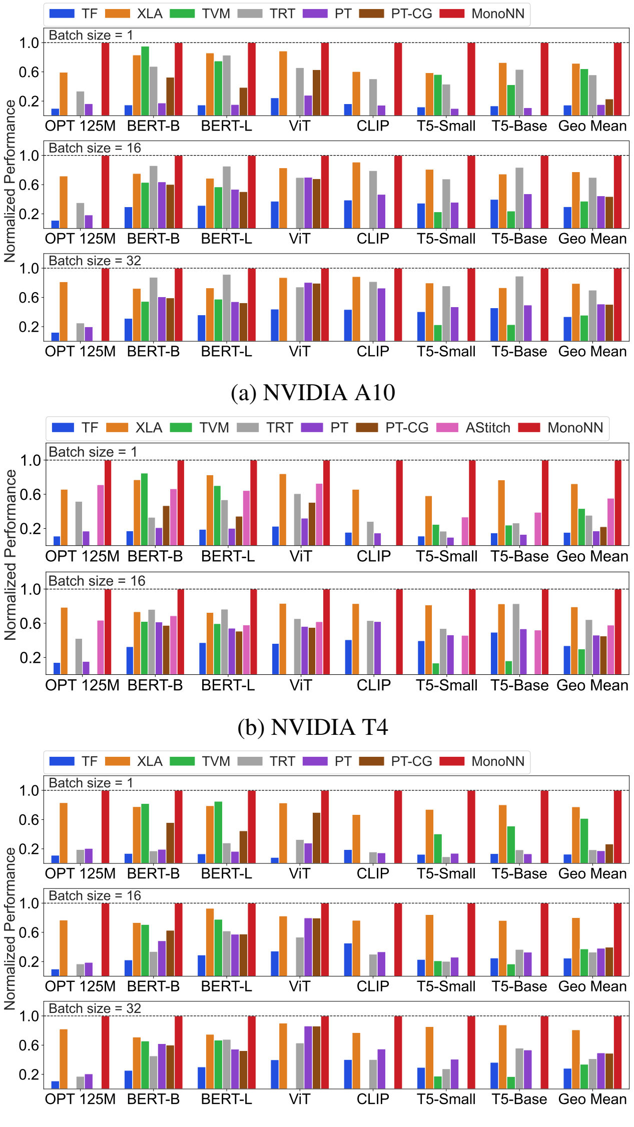
Fig.10 shows the end-to-end performance speedup on NVIDIA A10, T4, and A100 GPU for all experiments with three batch size variations. *Geo Mean* refers to the geometric mean across all models. All the execution time is normalized against the best optimizer in the group. TVM failed to optimize OPT, ViT, and CLIP due to incomplete operator support. PyTorch-CUDAGraph failed to optimize OPT, CLIP, and T5 for unsupported operations in the graph-capturing phase.

As demonstrated in Fig.10, on A10 GPU, MonoNN achieve 6.9×, 1.4×, 1.6×, 1.8×, 6.6×, and 2× average speedup over Tensorflow, XLA, TVM, TensorRT, PyTorch, and PyTorch-CUDA Graph on batch size 1, respectively. In addition, for batch size 16, and 32, MonoNN achieve on average 1.88×, and 1.80× speedup over baselines. On NVIDIA T4, MonoNN achieve 6.5×, 1.4×, 2.3×, 2.8×, 5.8×, 2.3×, and 1.8× average speedup over Tensorflow, XLA, TVM, TensorRT, PyTorch, PyTorch-CUDA Graph, and AStitch on batch size 1, respectively. In addition, for batch size is 16, MonoNN achieves on average 1.91× speedup over all baselines. We also have comprehensively tested MonoNN on A100 as detailed in Fig.10c.

Given the diverse set of baselines we compare against, the extent of performance improvements can vary. MonoNN consistently achieves the best performance across all baselines, with significant improvements observed for all testing batch sizes. It is important to note that reduced performance gains with larger batch sizes are anticipated, as

[7] TensorRT 8 is the latest version at the time of submitting the paper (Dec. 2022), with much performance improvement compared to TensorRT 7.

[8] AStitch is using its released artifact (CUDA 10.2 and cuDNN 7).



(a) NVIDIA A10

(b) NVIDIA T4

(c) NVIDIA A100

Figure 10: MonoNN End-to-End speedup (higher is better).

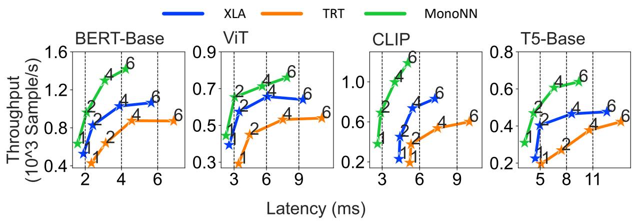


Figure 11: MPS Performance.

larger batches generally lead to better device utilization, leaving less room for performance enhancements. For example, XLA/TensorRT/MonoNN show on average 4.9×->2.6×->2.3× / 3.8×->2.3×->2.0× / 6.9×->3.4×->3.0× performance gain over the TF baseline when expanding the batch size from 1 -> 16 -> 32 on A10 respectively.

We test CUDA Graph via Pytorch integration. Despite the failure in some of the models in our benchmark, CUDA Graph achieves on average 2.6×, 0.95×, 0.98× speed up over PyTorch when batch size is $1,16,32$ on A10. Obviously, the performance gain of the CUDA Graph diminishes drastically (even with no performance gain) when the batch size is larger than one. The average speedup is far less than the achieved performance speedup of MonoNN. Specifically, MonoNN outperforms PyTorch-CUDA Graph by an average of 2× (batchsize=1) and continues to outperform it when batchsize is larger than 1. We attribute the reason as follows. On the one hand, MonoNN can perform various optimizations in the monolithic optimization space that CUDA Graph cannot, e.g., whole graph-level optimizations, instruction rescheduling, on-chip resource exploitation, and *GTB* merging. On the other hand, as pointed out by previous literature [37], a new GPU kernel has several types of overhead (e.g., kernel launching, kernel initialization) but CUDA Graph can only optimize kernel launching.

XLA achieves the best average speedup among our baselines. But XLA can only explore register-level data buffering rather than multi-dimensional optimization techniques in MonoNN. We only run AStitch [42] experiment on T4 GPU (with CUDA 10.2) because the artifact released does not support newer NVIDIA A10 architecture.

In addition, we evaluate MonoNN and the baselines on A10 using longer input for the BERT model. We use an input sequence length equal to 512, which is the maximum sequence length supported by the model's pre-trained positional embedding. As illustrated in Tab.1, MonoNN achieves on average 1.94×, 1.52×, and, 1.48× speedup over baselines when batch size is 1/16/32 respectively.

| | TF | XLA | TRT | PT | PT-CG | MonoNN |
|---|---|---|---|---|---|---|
| BS=1 | 0.27 | 0.90 | 0.44 | 0.39 | 0.57 | 1 |
| BS=16 | 0.40 | 0.87 | 0.75 | 0.63 | 0.62 | 1 |
| BS=32 | 0.42 | 0.90 | 0.79 | 0.63 | 0.63 | 1 |

Table 1: Normalized performance.

Among the evaluated models, the OPT-125M model has the smallest computation shape; only a single output token is generated at each step. This results in severe non-computation overhead, making MonoNN especially advantageous.

### 7.1.2 Impact on Throughput with MPS

This section is to demonstrate that MonoNN's optimizations can perform well for GPU-shared scenarios for higher throughput. In the real-world inference scenario, a common approach is to share a single GPU with multiple inference tasks to improve inference throughput and hardware utilization. NVIDIA Multi-Process Service (MPS) [10] is one of the most widely adopted solutions for GPU sharing. We test our solution with *MPS* for BERT-Base, ViT, CLIP, and T5-Base on A10 and plot the latency-throughput curve in Fig.11. The

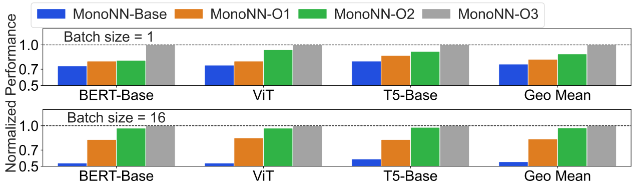


Figure 12: Ablation study on NVIDIA A10.

numbers on the line indicate how many instances are used in *MPS*. The batch size is one in this experiment. MonoNN consistently outperforms baselines, achieving 1.5 × −2× QPS throughput under the same latency constraint. It demonstrates that a monolithic kernel is capable of delivering meaningful speedup in GPU-shared inference. In practice, we divide TLP for each instance by the number of instances co-existing in *MPS* to ensure all instances can run concurrently on a single GPU.

### 7.1.3 Ablation Study

Fig.12 dissects the main optimizations in MonoNN. We build *MonoNN-Base*, a lightweight monolithic kernel generator that has all the optimization techniques of MonoNN except for the three: *context-aware instruction rescheduling*, *comprehensive on-chip resource exploitation* and *global barrier merging*. Note that *MonoNN-Base* is different from the baselines in Fig.10. *MonoNN-Base* is already a strong baseline that has many basic optimization techniques. It is a single monolithic kernel with minimal non-computation overhead, achieving better performance than TensorFlow and on par with XLA. We then build *MonoNN-O1-O3* by gradually applying the above optimizations one by one in order. *MonoNN-O3* is the full MonoNN. We observe 5%, 6%, and 12% speedup for *O1*, *O2*, and *O3* optimization on batch size 1, and 35%, 15%, and 3% speedup on batch size 16. *Context-aware instruction rescheduling* shows much more performance gain for batch size 16 because larger tensor shapes need higher parallelism, thus requiring ILP compensation more. In addition, we observe instruction rescheduling does not improve performance on *OPT-125M* model as the text generation model only produces a single token in each inference and a small tensor shape does not need a larger rescheduling factor. *Comprehensive on-chip resource exploitation* also shows higher performance gain on batch size 16 as larger tensors need more comprehensive solutions to accelerate off-chip memory access. *GTB Merging* shows larger performance gain when batch size is one because synchronization overhead is invariant to batch size and thus will take a larger portion when kernel duration is short.

## 7.2 MonoNN Optimization Breakdown

In this section, we dissect our optimization techniques proposed in Sec.4 and present a deep-dive into the solution generated by MonoNN with both conceptual and quantitative analysis to help understand optimizations in monolithic kernel better. Fig.13 shows the identified *CI Ops* and *SIS* in a BERT-Base model. We present a detailed analysis when the inference batch size is 16 on NVIDIA A10.

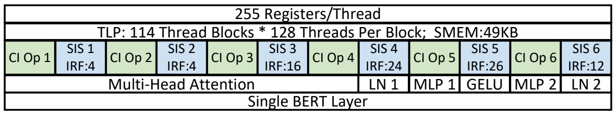
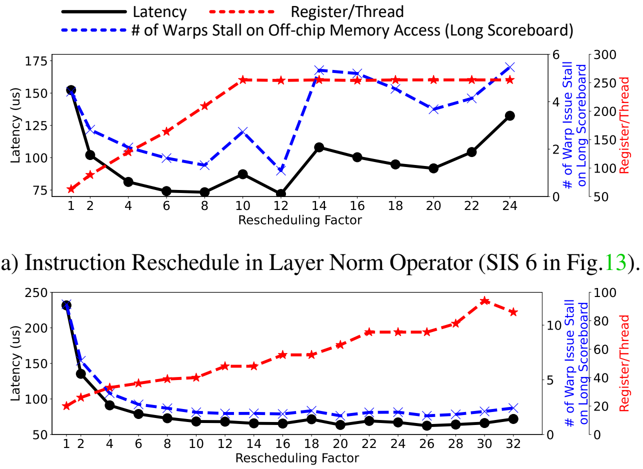

<table>
<tr><td colspan="12">255 Registers/Thread</td></tr>
<tr><td colspan="12">TLP: 114 Thread Blocks * 128 Threads Per Block; SMEM:49KB</td></tr>
<tr><td>CI Op 1</td><td>SIS 1<br>IRF:4</td><td>CI Op 2</td><td>SIS 2<br>IRF:4</td><td>CI Op 3</td><td>SIS 3<br>IRF:16</td><td>CI Op 4</td><td>SIS 4<br>IRF:24</td><td>CI Op 5</td><td>SIS 5<br>IRF:26</td><td>CI Op 6</td><td>SIS 6<br>IRF:12</td></tr>
<tr><td colspan="7">Multi-Head Attention</td><td>LN 1</td><td>MLP 1</td><td>GELU</td><td>MLP 2</td><td>LN 2</td></tr>
<tr><td colspan="12">Single BERT Layer</td></tr>
</table>

Figure 13: Identified *SIS* and *OCS* on a BERT layer, including TLP and on-chip resource usage of the monolithic kernel and instruction rescheduling factor (IRF) for each *SIS*.



(a) Instruction Reschedule in Layer Norm Operator (SIS 6 in Fig.13).

(b) Instruction Reschedule in GELU Operator (SIS 5 in Fig.13).

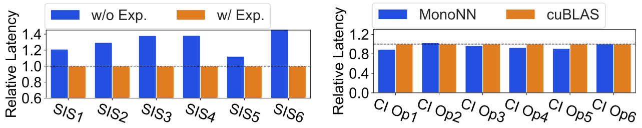
Figure 14: Context-aware instruction rescheduling analysis.

### 7.2.1 Context-Aware Instruction Rescheduling Analysis

*Context-Aware Instruction Rescheduling* (Sec.4.2.2) can increase ILP with more register usage. We show the rescheduling analysis of two subgraphs in Fig.14. As demonstrated in Fig.14a, when the rescheduling factor is too low, the average number of warps per SM per cycle that stall on off-chip memory access is high due to the low parallelism (both TLP and ILP), resulting in high inference latency. On the other hand, a too-high factor will cause register pressure and even register spilling. Slight register pressure often does not indicate degradation in performance, but register spilling often results in drastic performance degradation. We observe register pressure when the rescheduling factor is 10 and register spilling when the rescheduling factor is larger than 22. The best factor for *SIS6* is 12. *SIS5* in Fig.14b has less register usage compared to *SIS6*, for which the best rescheduling factor is 26.

### 7.2.2 On-chip Resource Exploitation Analysis

*Comprehensive On-chip Resource Exploitation* (Sec.4.3) can further exploit the on-chip cache and shared memory based on the data access pattern of the subgraph. Fig.15a shows performance improvement after applying this optimization



(a) *SIS* latency after Exploitation. (b) *CI Ops* latency.

Figure 15: Operator latency breakdown.

for subgraphs corresponds to Fig.13, achieving 1.3× speedup on average. Note that this is additional performance gain over *Context-Aware Instruction Rescheduling*.

#### 7.2.3 Performance of Compute Intensive Operators

We also detailed the performance of *CI Ops* in the monolithic kernel. All *CI Ops* need to follow the same TLP setting but the tensor shape for each *CI Op* could be different. Thus, handling different tensor shapes with a unified TLP setting is critical. MonoNN achieves this by leveraging intra-thread block tuning choices from CUTLASS. Through extensive evaluation, we found that intra-block tuning can generate satisfactory solutions for *CI Ops*. We illustrate the performance of *CI Ops* in monolithic kernel in Fig.15b. With the highly-tuned open-source vendor code (i.e., CUTLASS), all operators achieve on-par performance with the cuBLAS. Surprisingly, in some cases, the *CI Op* found by MonoNN is slightly better than cuBLAS. The reason we judiciously suspect is MonoNN performs an exhaustive search over all possible solutions whereas cuBLAS uses heuristics.

#### 7.2.4 Global Thread Barrier Merging Analysis

*GTB* is necessary for a monolithic kernel to ensure correctness. But each *GTB* involves a small overhead, approximately 2*us* based on our evaluation. Thus we need to minimize such overhead with *longest-path based GTB merging* (Sec.4.4). We compare *GTB* number before and after merging optimization. We observe 516, 658, 319, 710, and 366 *GTBs* in BERT-Base, CLIP, OPT-125M, T5-Base, and ViT model respectively. The *GTB* number reduced to 146, 185, 185, 315, and 184 respectively after *GTB* merging.

#### 7.2.5 Dissecting Non-computation Overhead

| | Bert-Base | ViT | T5-Base | OPT-2 |
|---|---|---|---|---|
| Framework | 0.41 | 0.57 | 0.44 | 0.61 |
| Kernel Launch | 0.71 | 0.51 | 0.72 | 1.74 |

Table 2: Context switch overhead breakdown (in ms).

Framework scheduling overhead and kernel launching overhead are two major sources of non-computation overhead. Tab.2 shows the separated framework scheduling and kernel launching overhead after optimization with XLA. It shows that the kernel launch overhead accounts for 61.7% of the overhead on average, larger than that of framework overhead. To measure the two kinds of overhead, we build two XLA variants. The first variant *XLA-framework* executes all framework scheduling logic as XLA, except it does not launch GPU kernel but returns immediately for each operator. Therefore, the inference latency of *XLA-framework* is pure framework overhead. The second variant *XLA-framework-and-kernel* has the same functionality as XLA, except that it launches empty GPU kernels (GPU kernels that do nothing) rather than the original kernels. The inference latency of *XLA-framework-and-kernel* is the summation of framework overhead and kernel launch overhead.

### 7.3 Fusion Granularity Analysis

| | EleWise | Stitch | Layer | Layer+CUDAGraph | Monolithic |
|---|---|---|---|---|---|
| OPT | 0.68 | 0.73 | 0.93 | 0.93 | **1** |
| MultiModal | 0.49 | 0.61 | **1.10** | **1.10** | 1 |

Table 3: Relative performance at each fusion granularity

We further analyze how kernel fusion granularity impacts inference performance on the models we evaluated to have a better understanding of the optimization space we proposed. Specifically, we control the fusion scope of MonoNN to generate code at different fusion granularity. From small to large, 1) **EleWise**: Element-wise fusion [2, 18]. 2) **Exhaustive memory-intensive fusion (Stitch)**: perform exhaustive fusion optimization on memory-intensive subgraphs using shared memory and global memory. This scope is similar to TensorRT [11] and AStitch [42]. 3) **Layer**: each layer of the neural network will be generated into a kernel with monolithic optimization. Note that from this scope, efficient code generation is unrealistic without the techniques proposed in this work. 4) *Layer+CUDAGraph* additionally apply CUDA Graph to the generated kernels. 5) *Monolithic*: the entire neural network is fused into a single kernel.

We choose OPT-125M and a customized multimodal model and benchmark them on A10 with a batch size equal to one. The multimodal model contains a transformer-based text encoder and a CNN+transformer-based image encoder. The setting with the best performance is highlighted in bold. We observe for a regular model like OPT with repetitive layers, monolithic kernel trend to achieve the best performance because all the optimization choices are essentially the same across layers. But for the multimodal model with complex structure, we observe the text encoder and image encoder trend to explore different optimization spaces due to divergence in computation tensor shape.

### 7.4 MonoNN Tuning Speed

MonoNN uses a grid search tuner with caching to tune the entire network. The modern neural network usually has many

|  | Bert-Base | Bert-Large | T5-Small | T5-Base |
|---|---|---|---|---|
| MonoNN | 22 | 70 | 17 | 68 |
| TVM | 172 | 220 | 51 | 116 |

Table 4: MonoNN end-to-end compilation time in minutes.

repetitious layers so that MonoNN avoids tuning them redundantly by caching the result from previous layers. We further apply many engineering-level optimizations to speed up tuning, which will not be highlighted in this paper. To this end, we found that MonoNN tuner can provide satisfactory tuning speed. We detailed quantitative numbers in Tab.4 collected from A10 GPU.

### 7.5 Comparison with Rammer

We compare MonoNN with Rammer [24] on BERT-Large inference. Rammer failed in optimizing all Huggingface public models in Fig.10 due to unsupported operators (e.g., Einsum, BroadcastTo). The only common inference model that we can find is the BERT-Large model from Rammer's official repository using fixed batch size 1. Thus, we cannot test other batch sizes on Rammer. In addition, Rammer does not support Tensor Cores. We thus compare with Rammer on NVIDIA T4 GPU after disabling Tensor Core usage for MonoNN. MonoNN shows 1.28× speedup over Rammer on BERT-Large model when using batch size equal to one and sequence length equal to 512.

## 8 Related Work

Most of the popular ML compilers focus on either single-operator or subgraph-level kernel generation. [16, 18, 26, 31, 32, 39, 41, 43] focus on compute-intensive operators optimization, with basic fusion support for memory-intensive ops, whereas [27, 41, 42] explore the stitch optimization of memory-intensive subgraphs. From graph level, [23, 33] explore graph transformation optimizations to accelerate neural network execution, which is orthogonal to our work.

Notably, holistic optimizations for machine learning workloads have received increased attention in recent years. VersaPipe [40] utilizes persistent-thread technique [14, 17] to execute a computation graph in a pipelined manner, in which the large kernel is spitted into several small kernels to avoid resource incompatibility problem. This approach is not sufficient to support computation graphs with massive operators, like machine learning graphs. Rammer [24] utilizes the persistent-thread technique to support large scope fusion, in which the task re-slicing and scheduling help to fill up execution units. Rammer does not touch the incompatibility problem and cannot support the monolithic optimization of an entire neural network efficiently. For example, the demoed BERT model of Rammer consists of 734 kernels on GPU. The persistent thread scheduling of VersaPipe and Rammer also introduces extra scheduling overhead, while MonoNN applies effective static scheduling to avoid such overhead. Moreover, neither VersaPipe nor Rammer explores the optimizations of on-chip resource exploitation and GTB merging like in MonoNN. BOLT [35] can fuse GEMM and its following operations into single kernels under restricted locality constrain. It cannot generate the monolithic kernel due to the incompatibility problem. Müller et al. [25] manually fuse all operators of a tiny MLP, small enough to fit on-chip, into a single GPU kernel for accelerated execution. In contrast, MonoNN explores a general approach for automatically high-performance code generation for common-sized models. There are ad hoc solutions to speed up single operator (e.g., LayerNorm) with instruction level parallelism on GPU [21]. None of the above work tackles the challenge of monolithic kernel generation.

## 9 Conclusion

We reveal that the kernel-by-kernel execution scheme is no longer effective in fully utilizing modern GPUs for various machine learning workloads, causing notable non-computation overhead and off-chip memory traffic. We propose the *monolithic kernel* execution scheme to tackle these problems, providing a vast new optimization space. We propose *context-aware instruction rescheduling* and *comprehensive on-chip resource exploitation* techniques to cope with the incompatibility problem between compute-intensive and memory-intensive operators. We systematically abstract the monolithic optimization space and propose *schedule-independent group tuning* approach to compress the extremely large tuning space. We develop a compiler integrating the optimizations automatically. Extensive evaluation on a set of inference tasks demonstrates that MonoNN outperforms state-of-the-art optimizers with on average 2.01× speedup.

## ACKNOWLEDGMENT

We appreciate the guidance from the OSDI reviewers and our shepherd during the review and revision process. We also extend our gratitude to Minmin Sun, Feiwen Zhu, Kai Zhu, Zaifeng Pan, Bohua Chen, and Wenyi Zhao at Alibaba Group for their valuable suggestions. This research is funded by Alibaba Group through the Alibaba Innovative Research Program. Donglin Zhuang and Haojun Xia are affiliated with the School of Computer Science at the University of Sydney, Australia, and conducted this work during their internship at Alibaba. Shuaiwen Leon Song is the corresponding author of this paper.

## References

[1] Cutlass: Cuda templates for linear algebra subroutines. https://github.com/nvidia/cutlass.

[2] Xla: Optimizing compiler for machine learning. https://www.tensorflow.org/xla.

[3] Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[4] Nvidia turing gpu architecture. https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[5] Nvidia ampere ga102 gpu architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2021.

[6] Nvidia hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022.

[7] Accelerating inference in tf-trt. https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html, Cited Dec 2022.

[8] Amd cdna architecture. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf, Cited Dec 2022.

[9] Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, Cited Dec 2022.

[10] Multi-process service (mps). https://docs.nvidia.com/deploy/mps/index.html, Cited Dec 2022.

[11] Tensorrt. https://developer.nvidia.com/tensorrt, Cited Dec 2022.

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[14] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149, 2009.

[15] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

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

[17] Michael Boyer, David Tarjan, Scott T Acton, and Kevin Skadron. Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. In *2009 IEEE international symposium on parallel & distributed processing*, pages 1–12. IEEE, 2009.

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

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *ICLR*, 2021.

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

[22] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. Cortex: A compiler for recursive deep learning models. In Alex Smola, Alex Dimakis, and Ion Stoica, editors, *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 47–62. ACM, 2019.

[24] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[25] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *ACM Trans. Graph.*, 40(4):36:1–36:16, 2021.

[26] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 883–898. ACM, 2021.

[27] Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, Wei Lin, and Xiaoyong Du. Recom: A compiler approach to accelerating recommendation model inference with massive embedding columns. In Tor M. Aamodt, Michael M. Swift, and Natalie D. Enright Jerger, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 268–286. ACM, 2023.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[29] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021.

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

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

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

[33] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: optimizing tensor programs with partially equivalent transformations and automated corrections. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 37–54. USENIX Association, 2021.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[35] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap be-

tween auto-tuners and hardware-native performance. *arXiv preprint arXiv:2110.15238*, 2021.

[36] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 681–699. USENIX Association, 2023.

[37] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. Understanding the overheads of launching cuda kernels. *ICPP19*, 2019.

[38] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022.

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

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

[41] Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, Xiaoyong Du, Jidong Zhai, and Wei Lin. Bladedisc: Optimizing dynamic shape machine learning workloads via compiler approach. *Proc. ACM Manag. Data*, 1(3):206:1–206:29, 2023.

[42] 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. 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*, pages 359–373, 2022.

[43] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: fast and efficient tensor compilation for deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 233–248. USENIX Association, 2022.