IntelliGen: Instruction-Level Auto-tuning for Tensor Program with Monotonic Memory Optimization

Zixuan Ma*
Haojie Wang
Jingze Xing
Tsinghua University, China

Shuhong Huang
Liyan Zheng
Chen Zhang
Tsinghua University, China

Huanqi Cao
Kezhao Huang
Mingshu Zhai
Tsinghua University, China

Shizhi Tang
Penghan Wang
Jidong Zhai
Tsinghua University, China

Abstract

Tensor compilers play a critical role in optimizing deep neural networks (DNNs), with memory performance emerging as a key bottleneck in code generation for DNN models. Existing tensor compilers are constrained by inefficient auto-tuning algorithms. They either must deploy coarse-grained descriptions, thus miss potential optimization, or struggle with vast search spaces, rendering auto-tuning inapplicable. Tensor compilers require a more holistic optimization of memory performance to overcome these constraints.

To address this issue, we focus our optimization objective on memory performance, which allows us to design monotonic optimization methods, significantly enhancing the efficiency of auto-tuning and thus enabling auto-tuning on a fine-granularity description. Based on these observations, we propose IntelliGen, a tensor compiler with instruction-level auto-tuning and monotonic memory optimization. We design an instruction-level graph description, iGraph, and a monotonic optimization method for optimization on iGraph. Benefiting from auto-tuning techniques with fine-grained description, IntelliGen demonstrates significant speedup of up to 3.13×, 3.55×, and 16.9× (averaging 1.46×, 1.85×, and 2.30×, respectively) on NVIDIA GPUs, AMD GPUs, and Cambricon MLUs over the most efficient existing frameworks.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**.

Keywords: Auto-Tuning, AI Compiler, Domain-Specific Accelerator

ACM Reference Format:
Zixuan Ma, Haojie Wang, Jingze Xing, Shuhong Huang, Liyan Zheng, Chen Zhang, Huanqi Cao, Kezhao Huang, Mingshu Zhai, Shizhi Tang, Penghan Wang, and Jidong Zhai. 2025. IntelliGen: Instruction-Level Auto-tuning for Tensor Program with Monotonic Memory Optimization. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708967

*Zixuan Ma is also affiliated with Qingcheng.AI.



This work is licensed under a Creative Commons Attribution 4.0 International License.

CGO '25, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708967

1 Introduction

Deep neural networks (DNNs) have been widely used in many vital domains. The substantial computational demands of DNNs have led to the development of accelerators. While the computational performance has rapidly increased, memory performance is lagging far behind. For instance, the half-precision computation and memory performance of NVIDIA GPUs have evolved from 21.2 TFLOPS and 732 GB/s in 2016 (Tesla P100) to 756 TFLOPS and 2,000 GB/s in 2023 (Tesla H100). Over this period, the computation-to-memory performance ratio has grown by 13.1×. In some common models, memory utilization can reach 96.7%, while computation can only reach 51.6% on average [17] due to the imbalance of computation and memory performance. Memory efficiency has become the main bottleneck for DNNs.

Tensor compilers are designed to generate efficient code for various accelerators. There are two primary factors that influence memory performance. The first is the volume of memory access. By reusing data in high-level memory (closer to registers), tensor compilers can reduce memory access, especially low-level memory. The second factor is the utilization of memory bandwidth. Tensor compilers achieve higher efficiency by selecting more suitable memory instructions. When deploying auto-tuning, a critical aspect influencing these two factors is the **granularity of the description**. The granularity refers to the level of detail encompassed in the auto-tuning process. Finer granularity implies more detailed information, leading to more search states. Generally, a coarser granularity of description is easier to optimize but may yield a lower potential performance ceiling. Conversely, a finer granularity can offer higher potential performance but makes the optimization more challenging. Thus, existing work must compromise between the granularity of description and auto-tuning demands.

Figure 1 illustrates how prior auto-tuning approaches generate code across multiple operators and how granularity influences search space for memory instructions. The methods represented by TVM [4], XLA [8], and TorchInductor [2] follow an **operator-level** code generation strategy

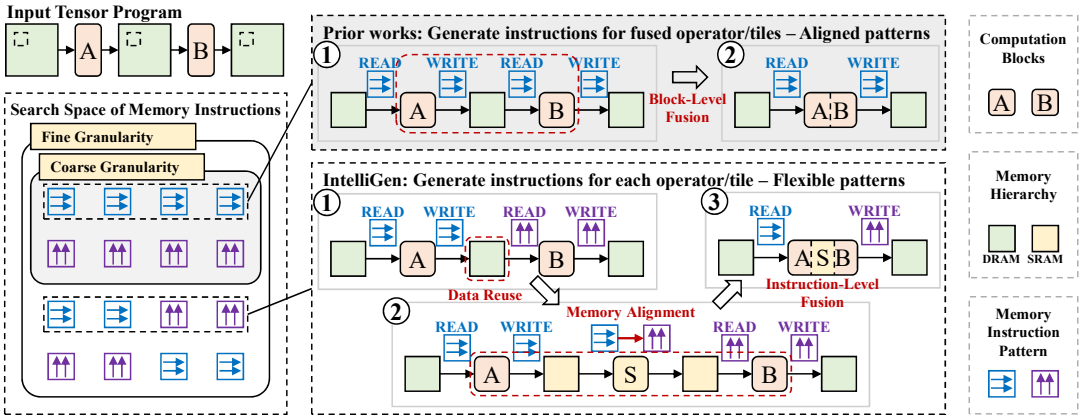


Figure 1. Illustration of how IntelliGen differs from previous work. Different description granularities lead to different search spaces they allow to fuse. Prior works (1) fuse operators or tiles first and then (2) generate instructions for them, which demands strictly aligned memory instructions. IntelliGen (1) generates instructions for each operator or tile, (2) analyzes diverse memory instructions and generates memory alignment for them, thus (3) providing more fusion opportunities.

that first fuse multiple operators into one and then generate codes for each. The techniques employed by Astitch [25] and Welder [17] consider memory hierarchy and use **tile-level** code generation. However, these strategies necessitate the fusion of operators or tiles before code generation, requiring strict alignment among the memory instructions of the operators or tiles to be fused. This requirement limits the search space and potential opportunities for optimization. A specific example of such limitation can be found in the operator group within the attention layer of GPT (to be discussed in a case study in Section 7.4). This group employs memory instructions that are not aligned across different operators, which cannot be fused by existing frameworks.

Instruction-level code generation offers descriptions with the granularity matching one or several vector instructions, aligning with the granularity of how hardware accesses memory. Therefore, theoretically, the search space at the instruction level can potentially cover the broadest range of possible implementations. As depicted in Figure 1, this approach can reuse data with flexible instructions within a specific hierarchy, leading to more potential optimizations. Existing approaches such as FreeTensor [18], Graphene [9], and ExoCompiler [11] have enabled writing tensor programs in instruction-level. However, finer granularity comes with the exponentially expanded search space. As a result, these works focused on creating specialized languages for writing efficient code. Auto-tuning techniques are still not fully explored within this extensive search space.

The primary challenge in enabling auto-tuning at fine granularity lies in the inefficiency of current techniques. Considering the complexity of both the programs and hardware, the transformations applied during auto-tuning are often blind, unable to guarantee speedup after transformation. Such approaches necessitate searching to identify the most optimal combination of transformations, as referenced in existing works [4, 12, 21]. The key to this challenge is the complexity of the optimization objectives. When targeting general performance, the influencing factors are multifaceted, which include computation performance, memory performance, and the efficiency of their overlap. Such complexity takes the unpredictability of performance and leads to complicated optimization strategies in existing works.

However, when focusing on memory-intensive programs, memory becomes the only bottleneck. Our observations reveal that the impact of transformations on tensor programs on memory performance is directional. For instance, actions such as transferring data from low-level to high-level memory or reducing the volume of memory access always enhance memory performance. Based on this observation, we propose an auto-tuning strategy that enables **monotonic** optimizations for a specific state. Such optimizations include eliminating redundancy among memory instructions or reusing intermediate data in high-level memory, which always positively impacts performance. This monotonic strategy enables fast optimization for specific instruction combinations and facilitates efficient exploration of the large search space for instruction-level auto-tuning. It has the potential to achieve higher performance by optimizing commonly recognized memory-intensive operators and those identified as highly optimized operators by DNN libraries in existing frameworks while also expanding the range of data layout selection to maximize library usage performance.

In this work, we introduce IntelliGen (available at https://github.com/JohndeVostok/IntelliGen), an instruction-level automatic tensor compiler with monotonic memory optimization designed to generate memory-efficient code across various architectures. IntelliGen utilizes a graph rewrite algorithm, enabling rapid exploration of the vast search space

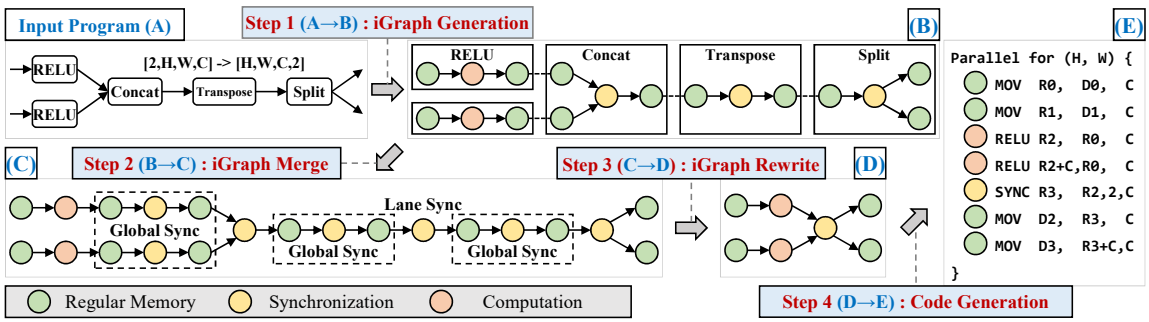


Figure 2. Overview and motivating example for IntelliGen.

provided by fine-grained descriptions. This method requires a graph-based instruction-level description, named iGraph, which outlines the dependencies among instructions. iGraph represents a tensor program with three primitives: computation, memory, and parallelism. The granularity aligns with hardware memory instructions, explicitly outlining the memory instructions and the synchronization requirements between them. Such a design enables IntelliGen to efficiently address memory-intensive programs, balancing the benefits of fine granularity and auto-tuning to achieve high-performance implementations for tensor programs.

IntelliGen is also a cross-platform tensor compiler. The description and optimization design allows for simple adaptation to various hardware architectures, including NVIDIA GPUs, AMD GPUs, and Cambricon MLUs. Our evaluations across different DNN models have demonstrated that IntelliGen can generate efficient code on various platforms, outperforming native computing systems such as TensorRT on NVIDIA GPUs and MagicMind on Cambricon MLUs, with speedup of average 1.45× and 2.30× respectively. IntelliGen's adaptability to diverse architectures is remarkable, requiring only up to 1,000 lines of code for each adaptation.

This paper makes the following contributions:

- By simplifying the optimization objective to memory performance, we designed a monotonic optimization method and an instruction-level graph description for the characteristics of such an optimization method.
- We proposed IntelliGen, an end-to-end memory-oriented tensor compiler, which leverages the techniques of instruction-level graph description and monotonic optimization to generate efficient code for various architectures.
- IntelliGen has been implemented across diverse architectures, including NVIDIA GPU, AMD GPU, and Cambricon MLU. It achieves speedup of up to 3.13×, 3.55×, and 16.91× (averaging 1.46×, 1.85×, and 2.30×), compared to the most efficient framework, respectively.

2 Background and Related Work

Single Operator Code Generation. Code generation frameworks such as TVM [4] and Ansor [24] adopt the compute/schedule separation idea and leverage auto-tuning techniques to explore various schedules using a loop-based description. These frameworks represent programs as perfectly nested loops, iterate through potential implementations by transforming loops, and map the optimized loops onto specific hardware. Works like Triton [19] also adopt a loop-based description. However, a significant drawback of these approaches lies in their inability to provide a direct description of memory instructions, impacting the optimization of memory bandwidth utilization. The absence of an explicit memory instruction description results in different schedules corresponding to the same memory instructions, leading to redundant search states and an unacceptable overhead. Furthermore, due to the blindness of existing auto-tuning algorithms, the effectiveness relies on developers' expertise. For instance, TVM requires users to write schedules for every operator on specific hardware. Such designs affect their effectiveness across various hardware platforms.

Polyhedral-based approaches, as exemplified by PPCG [20] and PLUTO [3], enable the analysis of element-level dependence by representing the program mathematically as points in a high-dimensional integer space. Such an approach is more fine-graphed compared to instruction. However, the overhead of dependence analysis makes it challenging to apply auto-tuning techniques. Additionally, the mismatch between element-level descriptions and how hardware accesses memory results in the mismatch between a transformation at the element level and the actual hardware execution process. Consequently, auto-tuning at the element level becomes redundant and blind, making it an unsuitable granularity for memory performance optimization. In summary, the optimization objectives constrain the efficient exploration of search space in existing auto-tuning approaches, thereby limiting the potential to achieve the best memory performance.

Multiple Operator Code Generation. Existing approaches commonly divide the code generation process for multiple operators into two steps: fusing operators or tiles and then generating code for them. For instance, TVM [4] performs operator-level fusion followed by auto-tuning of the merged operators. Welder [17] employs tile-level fusion followed by auto-tuning with Roller. TorchInductor [2] optimizes at the

operator level and then utilizes Triton to generate backend code. These methods explicitly separate these two processes.

A fine-grained description is required to analyze data dependencies and optimize data reuse. We utilize an instruction-level description for code generation. Furthermore, dependence analysis between instructions needs an initialization of instruction combinations. Considering that optimizing for data reuse impacts the selection of memory instructions. For instance, a specific reuse strategy can provide pruning conditions for auto-tuning. Thus, It is necessary to search memory instructions before data reuse optimization and integrate data reuse optimization and code generation within a unified framework. Overall, existing works, limited by their searching strategies, struggle to effectively analyze the data dependence and, thus, fail to get optimal performance.

3 Overview

Figure 2 provides an overview of IntelliGen and a running example of how it optimizes memory performance for a real-world DNN model. The given model fragment is from ShuffleNet. The input to IntelliGen is a tensor program formatted as a computation graph. IntelliGen transforms each operator into an instruction-level description, called iGraph, which will be introduced in Section 4. For each operator, IntelliGen will generate multiple iGraphs representing various implementations. Then, searches for the combinations of these implementations (step 1) and merges multiple iGraphs in each combination (step 2). For these merged iGraphs, a monotonic graph rewrite is employed to rapidly analyze and optimize data reuse (step 3), achieving less memory access volume. This process is called graph rewrite, which will be discussed in Section 5. Finally, IntelliGen utilizes a performance model-based approach to prune the optimization process, reordering the instruction sequence (step 4) and generating code for the iGraph with the highest performance (step 5). The optimization process is outlined in Section 6. By applying these techniques, we have developed IntelliGen as an end-to-end tensor compiler capable of auto-tuning with instruction-level description across different architectures.

4 Instruction-Level Description

In this section, we address the challenges of auto-tuning granularity by introducing an instruction-level description, denoted as **iGraph**. The iGraph serves as a description during the optimization, representing the implementation of an operation at the instruction level. Notably, the generation of iGraph is an internal process. Users do not need to write code with iGraph. iGraph represents a tensor program in terms of computation, memory, and parallelism, which describe the instructions and the dependencies between them under a specific parallelism. Figure 3 shows the description of iGraph and the device code for a RELU operation.

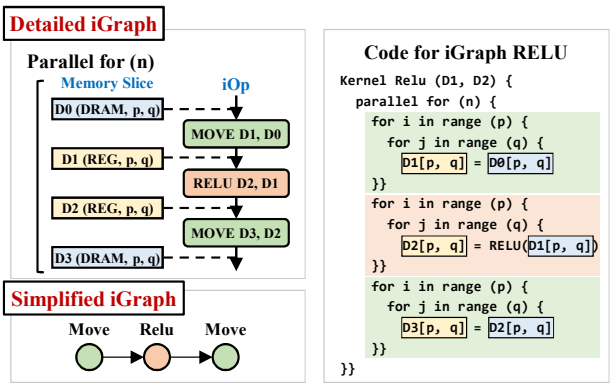


Figure 3. Illustration of the iGraph description and device code. For clarity and ease of understanding, iGraph are depicted in the form of *Simplified iGraph* in this paper.

Figure 4 illustrates the syntax of iGraph. Considering that the instructions can only be sequentially processed by a single unit (a unit refers to the smallest execution entity, corresponding to a warp on GPUs), it is necessary to include a **parallel** description to delineate how the program is partitioned across different units. Furthermore, to enable more accurate memory operation optimization, our approach diverges from descriptions used in prior works like TVM by explicit describe **memory** access, in addition to **computation**. We use a description called iOp to represent computation and memory instructions, forming the basis for iGraph.

To better facilitate data reuse optimizations for monotonic optimization, we structure iOps within a dataflow graph. In this graph, each node represents a iOp, while each edge represents the memory slices that iOp operated. This structure highlights the dependencies among instructions, thereby enabling graph algorithms for data reuse analysis and optimization. This approach reduces the complexity of designing and implementing the auto-tuning method. iGraph serves as a core component of the IntelliGen's optimization. This paper presents a description implementation suited to our instruction-level auto-tunning approach. The novelty of description design is not the focus of this paper.

4.1 Parallelism Description

In IntelliGen, the smallest unit of parallelism is defined as the minimal unit that independently influences memory performance. This unit varies across architectures: on NVIDIA GPUs, it is a streaming multiprocessor (SM, akin to a warp in CUDA), while on Cambricon MLU, it corresponds to an IPU (equivalent to a thread in BANGC). We refer to this minimal unit as a **parallel unit**. The parallelism description not only contains the number of parallel units involved in execution. It also encompasses the mapping relationship between the data accessed by the program and these parallel units, which specifies which memory segments a particular parallel unit needs to access during execution. To describe this mapping

$$
\begin{aligned}
\mathit{Program} &::= (\mathit{iGraph};)^+ \\
\mathit{iGraph} &::= \textbf{parallel for } \vec{i} \text{ in } \vec{0} \text{ to } \vec{n} \\
&\quad \{\ (\textbf{input } \mathit{Name} \textbf{ on } \text{DRAM} : \mathit{Slice}(\vec{i}); \\
&\quad\ \mid \textbf{slice } \mathit{Name} \textbf{ on } \mathit{Hierarchy} : \mathit{Slice}(\vec{i}) = \mathit{iOp};)^+\} \\
\mathit{Name} &::= \text{String} \\
\mathit{Slice}(\vec{i}) &::= (\mathit{Offset}(\vec{i}), \mathit{Shape}, \mathit{Stride}) \\
\mathit{Offset}(\vec{i}) &::= (\mathit{Scalar\ affine\ function\ of\ integer\ vector}\ \vec{i}) \\
\mathit{Shape} &::= (\text{Int}, \text{Int}) \\
\mathit{Stride} &::= (\text{Int}, \text{Int}) \\
\mathit{Hierarchy} &::= \text{Register} \mid \text{SRAM} \mid \text{DRAM} \mid \cdots \\
\mathit{iOp} &::= \mathit{MemoryOp} \mid \mathit{ComputeOp} \\
\mathit{MemoryOp} &::= \texttt{Move}(\mathit{Name}) \\
&\quad \mid \texttt{Sync}(\mathit{Name}, \mathit{SyncScope}) \\
\mathit{SyncScope} &::= \text{Thread} \mid \text{Warp} \mid \text{Block} \mid \text{Grid} \mid \cdots \\
\mathit{ComputeOp} &::= \mathit{UnaryOp}(\mathit{Name}) \\
&\quad \mid \mathit{BinaryOp}(\mathit{Name}, \mathit{Name}) \\
&\quad \mid \mathit{ReduceOp}(\mathit{Name}, \text{axis} = ([0] \mid [1] \mid [0,1])) \\
&\quad \mid \texttt{Bcast}(\mathit{Name}, \text{axis} = ([0] \mid [1] \mid [0,1])) \\
\mathit{UnaryOp} &::= \texttt{Exp} \mid \texttt{Log} \mid \texttt{ReLU} \mid \cdots \\
\mathit{BinaryOp} &::= \texttt{Add} \mid \texttt{Sub} \mid \texttt{Mul} \mid \cdots \\
\mathit{ReduceOp} &::= \texttt{ReduceSum} \mid \texttt{ReduceMean} \mid \cdots
\end{aligned}
$$

Figure 4. Syntax for iGraph.

relationship, each parallel unit should have a unique index, as denoted by $\vec{i}$ in Figure 4. Moreover, considering the parallel structure of hardware, such as the grid-block-warp hierarchy in CUDA and the IPU-MTP relationship in Cambricon MLU, these indices should also contain hierarchical information.

The parallelism is described as an inherent parameter for iGraph. Considering that current hardware organizes their parallel structure as perfectly nested, we can map each parallel unit to an index space, ensuring that the indices of included parallel units are contiguous within a specific parallel structure. For instance, in CUDA, the warps within the same block have contiguous indices. This design enables determining the relationships between different parallel units by comparing their indices. Due to the correspondence between parallel structures and memory hierarchy, this characteristic is crucial for data reuse analysis, which we will expound upon in Section 5. In addition, this approach eliminated the impact of various architectures by representing different hardware's nested parallel structures with a unified description, enabling the establishment of a unified framework for optimizing tensor programs on various hardware.

4.2 iOp Description

The design principle of iOp is to align with the granularity of hardware memory access. As a result, the operand of iOp is defined as a two-dimensional **memory slice**, which contains a group of contiguous memory segments. This granularity is consistent with the common practice of issuing multiple memory access instructions on GPUs and aligning with the strided DMA instructions on specific hardware platforms. For such a memory slice, we define several parameters, including *shape*, *stride*, *offset*, and *hierarchy*. As illustrated in Figure 4, *shape* and *stride* are expressly set to be two-dimensional. The *offset* is an affine function of the parallel for indices, which specifies where each parallel unit addresses its own slice on a specific memory hierarchy. Hierarchy denotes the memory's structural level (higher means closer to register), which will be modified by the graph rewrite algorithm as Section 5.

With the description based on memory slices, we can define an iOp. An iOp describes instruction-level operations on memory slices, which contain both computation and memory instructions, representing a single or a group of instructions. The main attributes of an iOp are displayed in Figure 4. There are additional parameters for certain types of iOps, such as the axis for reduce and broadcast iOps. This unified description enables the design of optimization algorithms across various hardware.

Memory iOp. Memory iOp is utilized for delineating instruction-level memory access. Such access may occur within the same level of the memory hierarchy or across different levels. Memory iOp has a single input and output with congruent shapes. To demonstrate the optimization methods of IntelliGen more clearly, we categorize memory iOp into two types. The first type is **regular memory iOp**, or move iOp, where different parallel units operate independently. In IntelliGen, for convenience of optimization, memory iOp between levels of the memory hierarchy must be a move iOp. This requirement implies that any conflicts must be resolved within the memory hierarchy through other memory iOps.

Furthermore, as the iGraph describes the parallel execution process of a program, such parallelism needs data synchronization. To address this, we introduce the second category of memory iOp, known as **synchronization memory iOp**, or sync iOp. This category describes data movement within the same memory hierarchy, with the synchronization scope required among involved parallel units. For example, synchronization within a block is needed to ensure program correctness when accessing shared memory on a GPU. Sync iOp is crucial not only for maintaining program correctness but also as an analytical tool for assessing data reuse within specific memory hierarchies. Thus, it forms a crucial component to support monotonic optimization strategies. A more detailed discussion of this will be presented in Section 5.

Computation iOp. The IntelliGen utilizes computation iOps to define the computations within the iGraph. A computation iOp delineates a computation block in hardware execution aligned with the granularity of memory access. Typically composed of a single or a group of instructions,

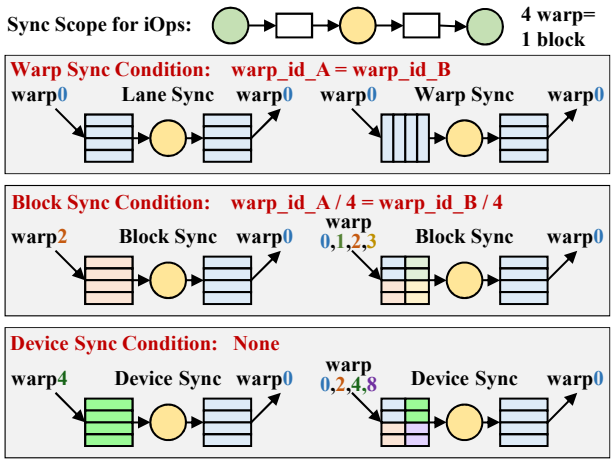


Figure 5. Demonstration of how IntelliGen determine synchronization scope.

it is designed to describe a basic computational function, similar to what is widely known as micro kernels. Common computation iOps for memory-intensive operations include **element-wise**, **reduce**, and **broadcast** operations. Element-wise iOps are responsible for managing individual computations on each element, encompassing tasks like arithmetic operations or activation functions. On the other hand, reduce and broadcast iOps handle dimension expansion and contraction, effectively reshaping data as necessary. For most memory-intensive operations, combining these computation iOps is enough for describing program function.

Figure 4 also shows the functional types of iOps. All iOps, including both memory and computation iOps, are aligned with the granularity of hardware instructions. This alignment enables organizing iOps into a graph format, which allows rapid optimization through graph algorithms. An additional benefit of this granularity is the consistent and uniform representation of computation and memory operation across diverse hardware platforms, thereby rendering the optimization of the iGraph platform-independent. When adapting to different hardware, only the implementation of iOp code generation is needed without hardware-specific considerations. This strategy simplifies optimizing tensor programs across various hardware systems.

5 Monotonic Graph Rewrite

The key to enabling auto-tuning at the instruction level in IntelliGen is designing a monotonic optimization strategy. Monotonicity refers to the certainty of each step in the optimization process. The algorithm does not require searching to determine whether each iGraph transformation should be performed, which ensures high efficiency in optimizing iGraphs. This certainty is crucial for IntelliGen to provide fine-grained auto-tuning. However, when directly targeting performance for any operator, the optimization becomes challenging due to the complexity of influencing factors, making it difficult to predict the direct impact of each transformation, which prevents existing methods from monotonic optimization. In this work, since the optimization target is limited to memory-intensive operators, we can focus on memory performance as the optimization target. Compared to the performance of any operator, memory performance is primarily influenced by low-level memory access volume and access bandwidth. When the implementation of each operator is fixed, only the low-level memory access volume affects memory performance. Thus, the optimization effect can be determined, offering an opportunity to achieve monotonic optimization. So, for a specific iGraph, we only need to analyze the dependencies between iOps, aiming to upgrade lower-level memory reads and writes to higher levels whenever possible, thus reducing low-level memory access.

To achieve this, we employ graph rewrite techniques. Graph rewrite contains a series of rewriting rules and a rewrite strategy. A rewrite rule describes a transformation on iGraph, and a rewrite strategy describes how to deploy rewrite rules. In IntelliGen, to ensure the performance of optimized iGraph, each rewrite rule is designed not to hurt the program's memory performance. To maintain the monotonicity of the optimization process, the rewrite strategy is designed to prevent cycles. Therefore, we proposed a series of graph rewriting rules and a rewrite strategy for iGraph. Such an algorithm requires only a $O(N)$ complexity and enables rapid optimization on iGraphs. It is important to note that this algorithm focuses more on speed than achieving the optimal result, representing a tradeoff between theoretical optimality and practical feasibility.

5.1 Rewrite Rules

Graph rewrite in IntelliGen is required to ensure the program's correctness, performance, and monotonicity. The design of the rewrite rule follows several guiding principles: 1) Any modification that affects the program's output is not permitted. The results of the program must be maintained throughout the rewriting process. 2) For memory iOps, adding any extra move iOps is not permitted. Only sync iOp addition is permitted. Modifications to memory iOps are limited to upgrading the scope of sync iOps. Memory iOps are allowed to be eliminated. This ensures that the memory performance is not hurt. 3) The rules must be unidirectional and forbid reversible transformations. This principle is crucial to prevent the rewriting process from undoing its own optimizations, thereby maintaining the monotonicity of the optimization. With these principles, we designed three specific rewrite rules. These rules ensure that the rewrite process in IntelliGen is effective and consistent with our optimization goal.

Hierarchy Upgrade. The hierarchy upgrade rule defines the method for upgrading the memory slice hierarchy to improve data reuse. When two memory slices are connected

by a sync iOp, this rule upgrades both the synchronization scope and memory hierarchy to the highest level, determined by analyzing the mapping relationship between the slices.

As mentioned in Section 4, the program's parallel structure is represented using nested indices. Therefore, whether the indices of two parallel units fulfill certain constraints decides whether they operate on the same parallel structure. For example, in a CUDA kernel with four warps per thread block, the synchronization scope condition is depicted in Figure 5. Each square symbolizes a memory slice, with its color representing the mapping and pattern representing the shape and stride. The synchronization scope is governed by the parallel units involved in the operations. If the writing unit is identical to the reading unit, the synchronization scope is confined to the warp level. If the result of dividing the indices of the unit that reads or writes by 4 is consistent, then the synchronization scope is confined to the block level. Otherwise, the scope is restricted to the device level. We define a hierarchy upgrade rule according to this analyzing process. We upgrade the memory hierarchy for the memory slices to the highest available level depending on the parallel units involved and modify the synchronization scope between them. This rule ensures the correctness of the program while maintaining that the memory performance does not decrease.

Memory iOp Merge. The memory iOp merge rule defines a method for reducing memory access in a iGraph by merging memory iOps. This rule covers two scenarios: 1) When two move iOps are connected by a sync iOp, and the output of the first and input of the second share identical memory slice patterns (including shape, stride, and offset). In this case, the iOps can be merged into a single memory iOp, with the input from the first iOp and the output from the last iOp. If the input and output memory slice patterns of the merged iOp are identical, the iOp can be eliminated, replacing the output slice with the input. Note that if the first move iOp has other successors, a copy of the iOp is retained to maintain connections with these successors. 2) When two move iOps share the same input memory slice. If their outputs also have identical memory slice patterns, one of the move iOps is retained, and the output of the other is replaced with the retained iOp's output. This rule efficiently merges multiple memory iOps into a single operation, reducing memory access in the iGraph without introducing extra move iOps.

iOp Swap. The iOp swap rule defines how to reorder iOps. When a swappable iOp is followed by a memory iOp, this rule will swap their order. This rule expands the applicability of the previous two rules. Consider a iGraph, as shown in Figure 6, bounded by two global sync iOps, which the previous rules cannot further optimize due to the disconnection between move and sync iOps. To overcome this limitation, we introduce the iOp swap rule. Specifically, certain computation iOps, such as element-wise operations (e.g., arithmetic

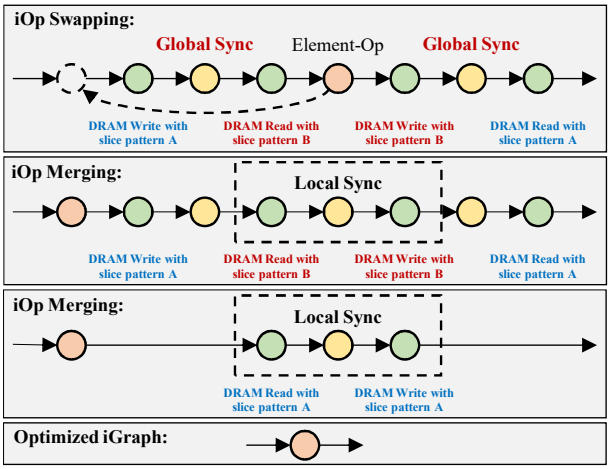


Figure 6. Example of iGraph rewriting.

and activation iOps), can be swapped with memory, broadcast, or certain reduction iOps without affecting the final result. These are referred to as swappable iOps. Figure 6 provides an example of a iGraph optimization using iOp swap. To ensure the process is directional, the rule enforces forward swapping of swappable iOps, considering the reversibility of the swap. By applying the iOp swap, the iGraph can then benefit from the hierarchy upgrade and iOp merge rules.

5.2 Rewrite Strategy

The goal of graph rewriting is to optimize the iGraph fast. The rewrite rules ensure the correctness, performance, and monotonicity of the algorithm. We adopt a greedy strategy that guarantees the optimal application of rewrite rules, resulting in the best possible memory performance. The implementation of this strategy is shown in Algorithm 1, and its optimality is proven in Appendix A.1. This strategy first applies the iOp swap rule wherever possible, followed by the iOp merge rule, and finally the hierarchy upgrade rule.

As each node is traversed at most once across the iGraph, the time complexity is $O(N)$, N represents the node. Compared to the time taken to search for possible status, the execution time of this rewrite process is negligible. Therefore, this algorithm enables us to explore a broader range of search space introduced by instruction-level description and achieve higher potential performance.

6 Efficient Auto-Tuning Methodology

As noted in Section 2, IntelliGen adopts a search-then-optimize approach. The major challenge in instruction-level auto-tuning lies in its massive search space. IntelliGen requires an auto-tuning methodology that significantly enhances search efficiency, enabling a thorough exploration within a reasonable time. Thus, search efficiency is the guiding principle in the design of IntelliGen's optimization methods. To achieve this, a more aggressive optimization

Algorithm 1 Graph rewrite algorithm.

```
Input: An input iGraph G
Output: The optimized iGraph G

function Upgrade(G, n_0)
    if n_0 is synchronization then
        e_0, e_1 = getInputAndOutput(n_0)
        h_0 = getHierarchy(e_0, e_1)
        update e_0, e_1, n_0 's hierarchy with h_0 to G

function MergeOp(G, n_0, n_1)
    if n_0, n_1 is mergable then
        merge n_0, n_1 and insert merged node to G

function SwapOp(G, n_0, n_1)
    if n_0, n_1 is swappable and n_1 is element-wise iOp then
        swap n_0 and n_1 in G

function Rewrite(G)
    for n_0, n_1 in G.nodes do
        SwapOp(G, n_0, n_1)
    for n_0, n_1 in G.nodes do
        MergeOp(G, n_0, n_1)
    for n_0 in G.nodes do
        Upgrade(G, n_0)
    return G

G = Rewrite(G)
```

strategy is employed, which is a tradeoff between search efficiency and algorithm optimality. However, the evaluation result shows that the benefits from the larger search space help IntelliGen achieve higher performance than prior works, validating the effectiveness of this strategy.

The optimization process in IntelliGen is structured into three key stages: generation of iGraphs for each operator, merging of these iGraphs, and optimization of iGraphs with monotonic graph rewrite techniques. This entire process is a search process. Benefiting from targeting memory performance, IntelliGen's method significantly improves the efficiency of the search in terms of directional optimization and searching strategy. On the one hand, monotonic graph rewrite techniques are directional and ensure better execution time while guaranteeing effective optimization. On the other hand, to leverage memory performance characteristics, we introduce a performance model-based pruning strategy that markedly enhances search efficiency. These two aspects make it feasible for IntelliGen to conduct auto-tuning at the instruction level. The following sections will focus on the design of IntelliGen's optimization methodology.

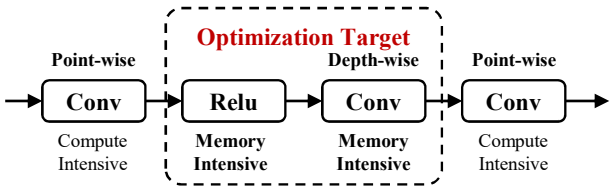


Figure 7. Example of a model fragment in ShuffleNet.

6.1 Preprocessing for Tensor Programs

The input of IntelliGen is a tensor program represented as a computation graph. IntelliGen aims to optimize memory performance. So, computation-bound operators within the tensor program are managed using specialized DNN libraries, such as cuDNN [5] on NVIDIA GPUs. The decision to utilize a DNN library is based on a comparative analysis of each operator's computation volume (C) and memory access volume (M), which can be calculated according to the operator's configuration. This comparison is quantified through the ratio C/M, which is then evaluated against a pre-established, hardware-specific threshold (H). This threshold reflects the computation-to-memory access performance ratio of different hardware architectures. The DNN library is employed when C/M exceeds H. Otherwise, generate code for them.

An illustrative example of a subgraph within ShuffleNet is shown in Figure 7. The depth-wise convolution, which is memory-intensive, is optimized along with the `RELU` operator. The point-wise convolution, which is computation-intensive, uses the DNN library. Preprocessing partitions a tensor program into multiple subgraphs, each containing only memory-intensive or only computation-intensive operators. These subgraphs are typically moderate in size. IntelliGen will independently auto-tune and generate code for each memory-intensive subgraph and then integrate them into the final compiled code.

6.2 iGraph Generation for Operators

The subsequent step in IntelliGen generates iGraphs for each subgraph (computation graph). This process unfolds in two primary stages: initially generating iGraphs for individual operators, followed by attempts to merge the iGraphs corresponding to different operators within the subgraph. First, complex operators need to be decomposed into basic operators. For example, the `SILU` operator will be split into `SIGMOID` and `MUL` operators. IntelliGen divide such basic operators into four types: element-wise, broadcast, reduce, and transpose. For each operator, we use multiple code templates to generate iGraph. Each template represents an implementation approach. There are multiple specific parameters for each template. A iGraph will be generated when the parameters are determined. We take reduce operator as an example. Different computation and memory iOps are required for reduction within one or multiple parallel units. These

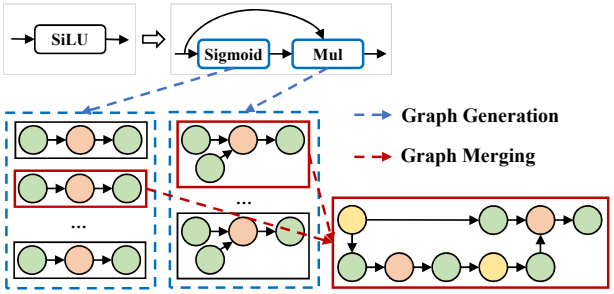


Figure 8. Illustration of iGraph generation and merging.

various implementations correspond to different code templates. Moreover, parameters within the code template, such as parallelism and tile size, can be searched. Divergent code templates and their corresponding parameters encapsulate various memory patterns, engendering performance differences and unlocking potential optimization opportunities. IntelliGen searches as many potential implementations of each operator as possible. The graph generation process for both the `SIGMOID` and `MUL` operators, which result in multiple iGraphs, is depicted in Figure 8.

After searching implementations for each operator, It is necessary to search various implementation combinations for operators to explore potential opportunities for data reuse. Multiple operators must be represented within a single iGraph. The merging of two iGraphs requires the same parallelism. The iOps within these graphs are connected according to the data corresponding to the memory slices, e.g., two slices in different iGraphs that correspond to the same tensor. Then, a synchronization iOp with global scope is inserted between them. Figure 8 illustrates how two iGraphs were successfully merged in this manner. Following the generation of iGraphs, IntelliGen employs graph rewriting algorithms to optimize data reuse within the iGraph. The optimized iGraphs thus become candidates for the final result.

6.3 Search Strategy Based on A*

The iGraph generation and merging process is a search process that explores all possible implementations for each operator and applies graph rewriting optimizations to every combination. The search space is extremely large: assuming each operator has, on average, T different implementations, and the graph contains N operators, the complexity of the search is $O(N^T)$. Conducting a brute-force search over such a space within a reasonable time frame is impractical. To address this challenge, we designed an A*-based search algorithm [10], as shown in Algorithm 2, which estimates the shortest execution time for each search state. The algorithm uses a priority queue to select the state with the lowest predicted execution time, allowing for efficient pruning and reducing the searching time.

The predict function is critical to this approach. To balance pruning effectiveness with the computation cost, we divide it into two parts. For the operators that have been searched and have certain implementations, we apply graph rewrite, generate code, and measure actual execution time. For the remaining operators, we predict the upper bound of memory performance after optimization. During prediction, we merge each operator into the existing iGraph only based on dependencies, without considering specific implementations. Operators that cannot be merged are assigned to new iGraphs. The system then estimates the change in execution time based on the following method.

Assume that the searched operators consist of n iGraphs, denoted as G_i. After merging the remaining operators, there are m iGraphs, denoted as G'_i, where $n \le m$. Given that the device's maximum memory bandwidth is B. The memory footprint of iGraph G is $M(G)$. The execution time of G'_i for $i < n$ can be estimated as:

$$Time(G'_i) = \begin{cases} Time(G_i) \times \frac{M(G'_i)}{M(G_i)} & \text{if } M(G'_i) < M(G_i) \\ Time(G_i) + \frac{M(G'_i) - M(G_i)}{B} & \text{if } M(G'_i) \ge M(G_i) \end{cases}$$

For newly merged iGraphs, i.e., $i \ge n$, the execution time is estimated as:

$$Time(G'_i) = Time(G_i) + \frac{M(G'_i) - M(G_i)}{B}$$

This method combines the accuracy of measurements and bandwidth-based estimates to reduce the computation cost of the predict function, significantly improving search efficiency. Additionally, since the estimate portion uses the maximum bandwidth, the predicted time is always less than the actual time, ensuring the optimality of the A* algorithm [10]. The proof of this property is provided in Appendix A.2. Note that The accuracy of the predict function is driven by the measurement, meaning that deeper search states yield more precise predictions. For the unmeasured portions, we focus on estimating upper bounds and the relative differences between states rather than absolute accuracy.

Moreover, since a significant search overhead comes from compilation and evaluation, the algorithm optimizes the number of intermediate states, which is also an advantage of A* [10]. By applying prediction-based pruning, we can efficiently explore a vast search space quickly, enabling exploration of the instruction-level auto-tuning.

6.4 Code Generation

The design principle of IntelliGen involves abstracting different hardware into a unified structure and extracting hardware features as parameters, which allows us to optimize programs and generate code for iOps specific to the hardware being used. By decoupling hardware and optimization, the system becomes more extensible. During the code generation, IntelliGen generates parallel structure code related to hardware information first and then sequentially generates device codes for computation and data movement iOps in order. The generated code is compiled using a native

Algorithm 2 Search strategy based on A*.

```
Input: ops: Operators of input graph, sorted by topology
Output: target: Optimized iGraphs

struct State
    idx, graphSet, time
end struct

function Merge(graphSet, nextGraph)
    if ∃graph_i ∈ graphSet, matched nextGraph then
        newGraph = merge graph_i and nextGraph
        return graphSet − graph_i + newGraph.
    else
        return graphSet + nextGraph

function Predict(GS, idx)
    time = evaluated time for GS after rewrite
    for i ∈ range(idx, n) do
        newGS = Merge(GS, op_i)
        time = time+ predicted Δtime of GS and newGS
        GS = newGS
    return time

function Search
    bestTime = +∞
    queue =Priority queue of State, sorted by State.time
    queue.push({0, ∅,Predict(∅, 0)})
    while queue ≠ ∅ do
        curState = queue.pop()
        if curState.time > bestTime then
            continue
        if curState.idx = ops.size then
            bestTime = curState.time
            target = curState.graphSet
        else
            i = curState.idx + 1
            GS = curState.graphSet
            impls = iGraphs for op_{i−1}
            for nextGraph ∈ impls do
                newGS = Merge(GS, nextGraph)
                state = {i, newGS,Predict(newGS, i)}
                if state.time < bestTime then
                    queue.push(state)
    return target

target =Search
```

compiler and evaluated for optimal performance. This design enables the low porting cost of IntelliGen to new hardware, as porting only requires mapping the parallel abstraction and primitives for computation and data movement. For example, porting to NVIDIA GPU and AMD GPU requires less than 1, 000 lines of new code for each. We will explain how IntelliGen adapts to different platforms and generates efficient code using NVIDIA GPU and Cambrian as examples.

NVIDIA GPU. The GPU is the most commonly used accelerator in the AI domain, with vendors such as NVIDIA and AMD having similar hardware architectures and programming models. To illustrate how IntelliGen is ported to GPUs, we show how it maps to CUDA on NVIDIA GPUs. CUDA's parallel structure consists of three layers: thread block, warp, and thread. To map the parallel unit of iGraph onto CUDA, we map it to the warp, as described in Section 4. The warp is the smallest scheduling unit in CUDA with independent performance. In contrast, threads in CUDA have dependent memory access and scheduling and correspond to SIMD lanes in the SIMD architecture, which makes them unsuitable as parallel units in iGraph. During the search process, the number and size of thread blocks in CUDA can be configured and used as search parameters. For the instructions of computation and memory access operations, the CUDA program represents the program executed on a specific thread. Thus, in iGraph, an implementation that says "read contiguous memory containing $n \times 32$ elements" would generate "read n element with stride 32".

Cambricon MLU. The architecture of Cambricon MLU differs from that of GPUs. It has two layers of scratch-pad memory, NRAM and SRAM, located in the scope of an IPU or an MTL Cluster. However, data movement operations on the MLU platform require direct memory access (DMA), which can introduce overhead and limit the memory performance achievable by synchronously performing computation and data movement operations on each IPU. To improve memory efficiency, IntelliGen applies pipeline parallelism, which executes multiple pipelines on a single IPU simultaneously, enabling data movement to overlap with computation. During code generation, IntelliGen selects the parallel unit as a pipeline on the IPU and interleaves multiple pipelines to generate execution code. This pipeline parallelism optimization introduces a new parallel dimension under the MTP Cluster and IPU and does not affect any existing optimization on iGraph. The successful porting of IntelliGen to the Cambricon MLU platform demonstrates its strong extensibility.

7 Evaluation

7.1 Evaluation Setup

Hardware Platforms. We evaluated IntelliGen on three different hardware platforms. For NVIDIA GPUs, we use Tesla A100-40GB PCIe with CUDA 11.7.0 and set the memory and application clocks to their maximum values. For AMD GPUs, we use INSTINCT MI100 with ROCm 6.2. For Cambricom, we use MLU-370x4 with BANGC 1.0. We set the clocks to the maximum value on all platforms. These hardware platforms were chosen to represent a diverse range of

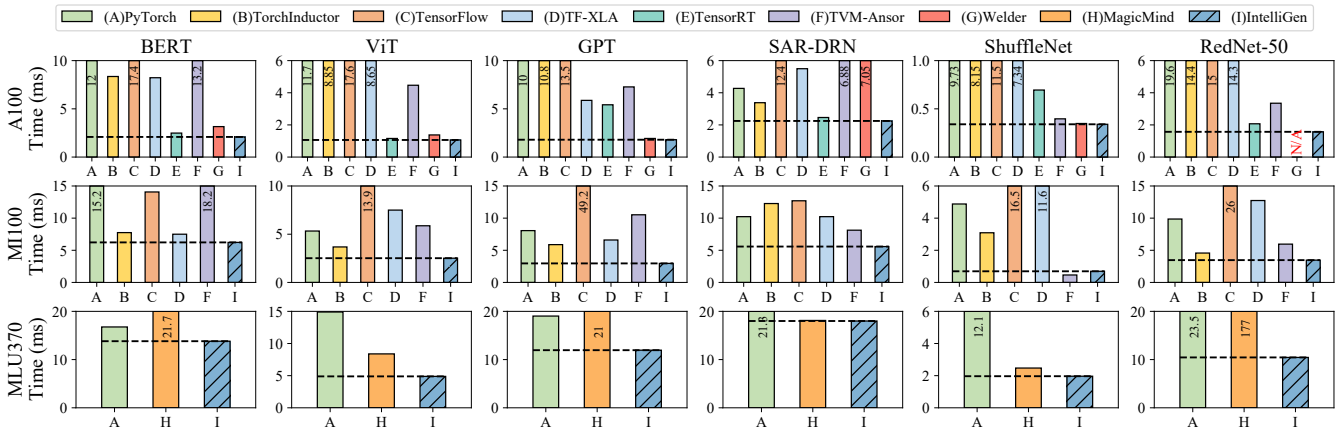


Figure 9. End-to-end performance of batch size 1 (GPT's batch size is 128).

architectures and to provide a comprehensive evaluation of IntelliGen's performance and portability.

DNN Models. We evaluated IntelliGen on six models, including both transformer and CNN models for diversity, including BERT [6], GPT [16], ViT [7], SAR-DRN [22], ShuffleNet [23] and RedNet-50 [13]. In the evaluation, we set the batch size of all models to 1 and 16 except GPT, whose batch size is 128 as an autoregressive model.

Baselines. TensorFlow [1] and PyTorch [2] are widely used machine learning frameworks. We evaluated PyTorch and TensorFlow on NVIDIA and AMD GPUs, and PyTorch on Cambricon MLU, using PyTorch 2.1.1 and TensorFlow 2.10.0. TorchInductor [2], a PyTorch library for converting models into a more efficient serialized format. It uses Triton [19] to generate device code. It was evaluated with the version aligned with PyTorch 2.1.1. TensorFlow-XLA [8], a linear algebra compiler within TensorFlow, was employed to automatically parallelize and vectorize computations, corresponding to TensorFlow 2.10.0.

TensorRT [14] and MagicMind are vendor-developed deep learning inference optimizers, leveraging high-performance kernels. The version of TensorRT is 8.4.0. TVM [4], a machine learning compiler based on loop descriptions, and Ansor [24], an automated scheduling tool representing the state-of-the-art for tensor compilers, were evaluated with TVM 12.0 on NVIDIA and AMD GPUs.

Welder [17] is an ML compiler with tile-level code generation. Note that the evaluation in [17] deployed on V100 GPU, while this paper is on A100 GPU. Due to the performance differences between TensorRT on the V100 and A100, the performance comparison between Welder and TensorRT varies. All the evaluations are deployed with Tensor Core or Matrix Core with TF32 or FP32.

Table 1. End-to-end performance of batch size 16 on A100. Execution times are in ms. Speedup is over the best baseline. GPT of batch size 128 is shown in Figure 9.

Models	BERT	ViT	SAR-DRN	shuffleNet	RedNet-50
PyTorch	34.4	10.5	64.7	8.83	18.5
TorchInductor	28.7	7.41	50.9	5.19	11.9
TensorRT	27.2	3.08	37.6	1.06	7.16
Welder	35.5	3.39	114	1.06	N/A
IntelliGen	**22.0**	**2.87**	**33.1**	**0.734**	**2.79**
Speedup	**1.23×**	**1.07×**	**1.14×**	**1.44×**	**2.57×**

7.2 End-to-End Performance

The end-to-end evaluation results for batch size 1 (Figure 9). All results are averaged over 1024 independent runs, demonstrating that IntelliGen outperforms all baselines on A100. IntelliGen achieves an average speedup of 7.88×, 6.37×, 11.07×, and 6.08× over PyTorch, TorchInductor, TensorFlow, and TensorFlow-XLA, respectively. Additionally, compared to TensorRT, IntelliGen accelerates models by 50% on average and achieves a speedup of 2.45× on the GPT model. Moreover, compared to TVM, IntelliGen achieves an average speedup of 3.06×. Compared to Welder, IntelliGen achieves a speedup of 46% averagely. For BERT and ViT, IntelliGen has similar performance as TensorRT because GEMM takes up most of the execution time. Meanwhile, for shuffleNet, IntelliGen achieves similar performance as TVM and Welder. This is because IntelliGen spends more time on convolution operations.

On MI100, IntelliGen outperforms all baselines, achieving an average speedup of 2.83×, 1.90×, 6.62×, and 3.11× over PyTorch, TorchInductor, TensorFlow, and TensorFlow-XLA, respectively. Compared to TVM, IntelliGen has an average speedup of 1.85×. The lower acceleration ratio on the AMD GPU is due to the lower ratio of peak computation performance to memory bandwidth than that of the NVIDIA

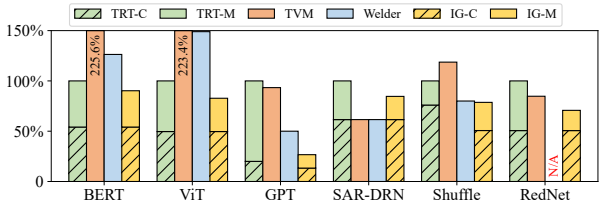


Figure 10. Kernel number breakdown

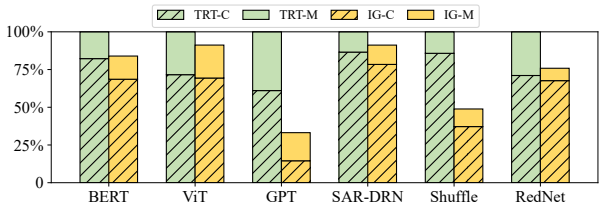


Figure 11. Kernel execution time breakdown

GPU. The memory access operation takes less time, resulting in less significant optimization results.

As demonstrated in Figure 9(c), on Cambricon MLU-370, IntelliGen achieves an average speedup of 2.40× over PyTorch and a speedup of 2.30× over MagicMind. This result indicates that IntelliGen supports different architectures and has good cross-platform optimization capabilities.

Due to space constraints, we only list the batch size 16 results for NVIDIA GPUs. The trends are similar to batch size 1. The results are shown in Table 1. Compared to PyTorch, TorchInductor, TensorRT, and Welder, IntelliGen exhibits an average speedup of 3.89×, 2.75×, 1.41×, and 1.75× respectively. This demonstrates our system's consistent and significant acceleration across different batch sizes.

7.3 Breakdown

To clarify how IntelliGen outperform prior works and how memory optimizations affect end-to-end performance, we evaluate the breakdown of execution time and kernel counts for TensorRT (TRT), TVM, Welder, and IntelliGen (IG). The results are shown in Figure 10 and Figure 11. Specifically, we define MatMul and Conv-2D operations that use DNN libraries as computation kernels (denoted as -C), while other operations are categorized as memory kernels (denoted as -M). Since TVM and Welder generate code for all operations with unified optimizations, we cannot categorize its kernels. So, we only compare the total kernel counts with TVM and Welder. For the breakdown of TensorRT, we used NVIDIA Nsight Systems to obtain the execution time of each kernel. To deal with the overhead of profiling, we aligned the total execution time with the end-to-end execution time of TensorRT. Despite these corrections, profiling will still introduce inaccuracies in the execution times across different kernels.

We identify the optimizations of IntelliGen into three categories:

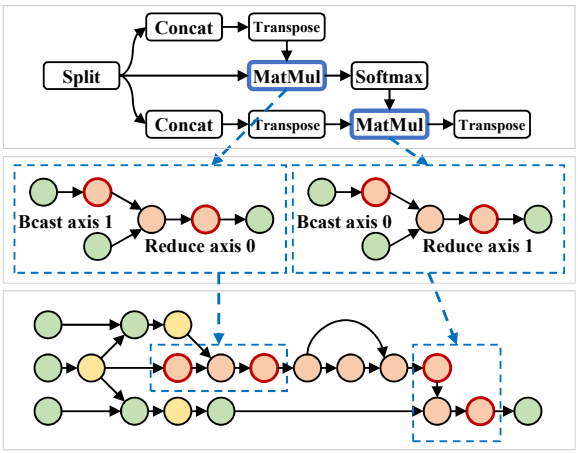


Figure 12. Case study of GPT.

1) The results show that for BERT, ViT, and RedNet50, IntelliGen and TensorRT use the same number of computation kernels, but IntelliGen generates fewer memory kernels, especially in RedNet50, where the number of memory kernels is reduced by 59% compared to TensorRT. In the BERT model, IntelliGen separates Transpose from MatMul during optimization and fuses them with other memory operators. This optimization enables IntelliGen to adopt a different data layout than TensorRT and improve the execution time for computation kernels. TVM and Welder use more kernels than IntelliGen and TensorRT. The results of these models demonstrate IntelliGen's ability to fuse memory kernels and highlight how memory optimizations influence the performance of computation kernels.

2) For SAR-DRN, TVM and Welder use the same number of kernels as IntelliGen and TensorRT for computation kernels. This indicates that TVM can fuse all memory kernels into computation kernels. However, IntelliGen and TensorRT are constrained by vendor-provided libraries that will leave unfused memory kernels. Compared to TensorRT, IntelliGen applies a different fusion strategy for computation kernel epilogues(e.g., activation). IntelliGen fuses them with memory operations and thus reduces the execution time of the computation kernel. This optimization is the crucial factor of the performance improvements over TensorRT in this model.

3) In contrast, for GPT and ShuffleNet, IntelliGen reduces the number of computation kernels by identifying some of them as memory kernels. Through further fusion optimizations with memory operations, IntelliGen significantly reduces the total number of kernels and execution time. For instance, in GPT, the number of kernels generated by IntelliGen is reduced by 73%, 71%, and 25% compared to TensorRT, TVM, and Welder, respectively. Compared to TensorRT, IntelliGen decreases the execution time of computation kernels by 76% and 57% in these two models.

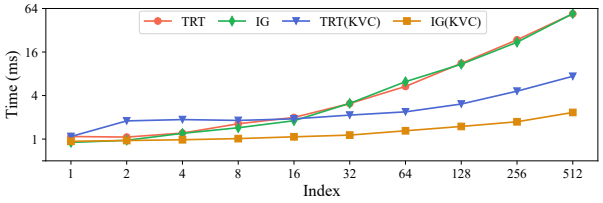


Figure 13. Performance of GPT with different indices.

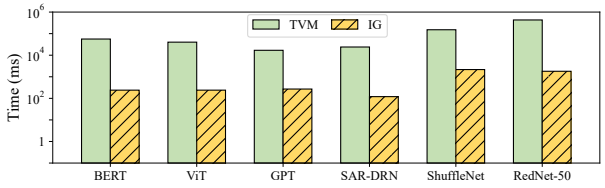


Figure 14. Searching time of TVM and IntelliGen.

7.4 Case Study

To illustrate how IntelliGen identifies the optimization for memory operators and outperform existing frameworks, we use GPT as an example. Figure 13 presents the performance of GPT inference on A100 GPU. It shows that without KVCache, IntelliGen's performance is comparable to that of TensorRT. TensorRT optimizes the three fixed memory-intensive operators in the computation graph, and IntelliGen proposes similar optimization strategies. KVCache optimization is widely deployed in generative models [15]. It leverages the characteristics of the GPT model by storing intermediate results in every iteration and reusing them later. KVCache introduces two Concat operators in the attention layer, which concatenate the current and previous results. Thus, it reduces computation time by avoiding recomputation and maintains the equivalence of results. As shown in Figure 13, the execution time of TensorRT significantly decreases after applying the KVCache technique, especially for large input indices.

IntelliGen can apply further optimization to GPT with KVCache. As each iteration only requires the computation of one token, the two MatMul operations in attention layers degenerate into GEMV, which is memory-intensive. Thus, using IntelliGen to jointly optimize all operations of attention and generate a memory-efficient kernel significantly improves program performance. Figure 12 demonstrates how IntelliGen optimizes the attention layer, where the two MatMul have identical shapes and computation patterns. However, the program with the best performance after optimization uses distinct iGraphs for them. These iGraphs differ in their data slice pattern and axis of computation operation, making such optimizations challenging in other approaches. Therefore, TensorRT, TVM, and Welder will generate at least 3 kernels for the attention layer. IntelliGen exhibits a more significant performance advantage than TensorRT after applying KVCache, improving performance by up to 3.16×.

7.5 Searching Time

We compare the search times of IntelliGen and TVM. AsFigure 14, the search time for IntelliGen is significantly shorter than TVM's. This outcome is attributed to two factors. (1) Our system directly uses the native library's implementation for computation-intensive operations without searching. (2) We leverage memory performance model pruning during the optimization, which prevents the exploration of vast useless search space and consequently decreases search overhead. Our evaluation results demonstrate that IntelliGen can generate code for models with less time.

8 Conclusion

We propose IntelliGen, a tensor compiler with instruction-level auto-tuning and monotonic memory optimization. Targeting memory performance, IntelliGen enables rapid and efficient optimizations through a larger search space. The evaluations on NVIDIA GPU, AMD GPU, and Cambricon MLU show IntelliGen can outperform existing frameworks by up to 3.13×, 3.55×, and 16.91×, and averages of 1.46×, 1.85×, and 2.30×, respectively.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work is supported by the National Key R&D Program of China under Grant 2023YFB3001501, Beijing Natural Science Foundation (L242017), NSFC for Distinguished Young Scholar (62225206), National Natural Science Foundation of China (U23A6007), the Young Student Basic Research Fund of the National Natural Science Foundation of China (623B2061), and Tsinghua University Initiative Scientific Research Program. Jidong Zhai is the corresponding author of this paper.

A Appendix

A.1 Proof of Graph Rewrite Strategy

Our graph rewriting strategy employs a greedy algorithm, ensuring that the rewrite rules are applied in an optimal sequence to achieve the best memory performance. Let us assume that the optimal rule sequence is $R_0, R_1, ..., R_n$, denote as R, and the sequence derived from the graph rewriting strategy is $R'_0, R'_1, ..., R'_m$, denote as R'. Proving the optimality of the graph rewriting method is equivalent to proving that R and R' are equivalent.

Only iOp Merge. When R consists of iOp merge operations, R' represents the order in which iOp merge rules are applied as much as possible. We assign an index to all move iOps in the graph, and each rule R_i describes merging R_i to its predecessor. For any i, consider swapping the order of R_i and R_{i+1}.After applying the rules $R_0, R_1, ..., R_{i-1}$ to the original graph, we obtain a graph G. Applying R_i and R_{i+1} to

G produces G', and applying R_{i+1} followed by R_i produces G''. Swapping these two rules will not change the result. So $G' = G''$.Thus, applying the subsequent rules R_{i+2}, R_{i+3}, ..., R_{n-1} to both G' and G'' will yield the same result. Hence, for a rule sequence R consisting only of iOp merge operations, the order of any two adjacent rules can be swapped. It follows that the order of R does not affect the outcome.

If $\exists R_i \in R, R_i \notin R'$, then after applying R', R_i could still be applied, contradicting the assumption that R' applies iOp merge as much as possible. Therefore, $\forall R_i \in R, R_i \in R'$. Similarly, if $R'_i \in R', R'_i \notin R$, then after applying R, R'_i could still be applied. Since iOp merge always reduces memory access, applying only R would not yield the optimal result, contradicting the assumption. Therefore, $\forall R'_i \in R', R'_i \in R$. Since R and R' contain the same set of rules, and their order does not affect the outcome, R and R' are equivalent, confirming that a rule sequence consisting only of iOp merge operations satisfies the optimality of the greedy algorithm.

iOp Merge and Upgrade. For rule sequences R and R' that involve both iOp merge and upgrade operations, where R represents the optimal sequence and R' applies iOp merge as much as possible and then applies iOp upgrade as much as possible. Considering the upgrade rule applies to sync iOps without affecting the graph structure, we define an upgrade rule by the sync index.

Let G be the graph resulting from applying R, and G' the graph from applying R'. Since iOp merge does not depend on memory hierarchy information, G and G' have the same structure, differing only in the sync iOps and memory slice hierarchies. If there exists a sync iOp $n_i \in G$, there must be a corresponding sync iOp $n'_i \in G'$, with memory hierarchies $H(n_i)$ and $H(n'_i)$, respectively. If $\exists i, H(n_i) < H(n'_i)$, then R' could still apply the upgrade rule to sync iOp n'_i, contradicting the assumption that the upgrade rule was applied as much as possible. Conversely, if $\exists i, H(n_i) > H(n'_i)$, then G would have longer memory times than G', which contradicts the assumption. Therefore, $\forall i, H(n_i) = H(n'_i)$, and thus $G = G'$. This demonstrates that the greedy algorithm can be applied for the iOp merge and upgrade rules.

All Rewrite Rules. For rule sequences R and R', where R represents the optimal sequence and R' applies the rules in the following order: as many swaps as possible, followed by as many merges as possible, and finally, as many upgrades as possible. We construct R'' by first applying all possible swap rules, then following the sequence of non-swap rules from R. For $i < m$, R''_i corresponds to swap rules, and for $i \geq m$, R''_i corresponds to other rules.

Since swap rules move swappable operations to the edge of the connected block formed by memory iOps, any two adjacent memory iOps x and y in the original graph remain adjacent after applying the R''_{m-1} rules. For $i \geq m$, R''_i corresponds to R_j in R. If G is the graph obtained by applying R_0, R_1, ..., R_{j-1} and G'' is the graph obtained by applying R''_0, R''_1, ..., R''_{i-1}, any adjacent memory iOps x and y in G will also be adjacent in G''. Therefore, if G allows the application of R_j, G'' must allow the application of R''_i, and the connectivity change between x and y remains the same. By induction, all rules in R'' can be applied, meaning R'' is equivalent to R and also an optimal sequence. For $i < m$, $R' = R''$, and for $i \geq m$, R' and R'' contain only upgrade and merge operations, making them equivalent. Therefore, R' is equivalent to R'', and R' is also equivalent to R. This proves the optimality of the greedy algorithm.

A.2 Proof of Optimization Method

Our optimization method guarantees finding the optimal solution within the search space. To simplify, we map the optimization search problem to a single-source shortest path problem on a directed graph. Each node in the graph corresponds to a search state, where the first i operators are fixed to specific implementations, denoted as state S_i. When the $(i+1)$th operator is fixed, the resulting state is denoted as S_{i+1}. There is an edge between nodes S_i and S_{i+1}, with the edge weight representing the difference in the best execution time between S_i and S_{i+1} after optimization. By adding edges from the source node to the state where no operators are searched and from the state where all operators are searched to the sink node, the shortest path from the source to the sink corresponds to the optimal solution. The length of the shortest path is the execution time of the optimal strategy.

Proof of Reachability. First, we prove that the method is guaranteed to find at least one solution. We assume that a brute-force method can always find a solution. If our method fails to find any solution, then the variable $bestTime$ remains unchanged. In this case, the pruning condition is $time \neq +\infty$, meaning the search space is equivalent to that of the brute-force method, which guarantees a solution exists. This contradicts the assumption that no solution was found. Therefore, our method must find at least one solution.

Proof of Optimality. Next, we prove that the solution found by our method is optimal. Assume there exists a path R that our method did not explore and that the execution time of R, denoted $time(R)$, is less than the best time found by our search, $bestTime$, i.e., $time(R) < bestTime$. This implies that the A* algorithm failed to find the optimal solution.

For this, a pruned state S_i on path R must exist, such that at the point of pruning, the condition $time(S_i) > bestTime$ was met. Otherwise, path R would have been explored. However, due to the nature of the heuristic function, any extension of S_i must have an execution time greater than $time(S_i)$. Thus, $time(R) > time(S_i) > bestTime$, which contradicts the assumption that $time(R) < bestTime$. Therefore, the A* algorithm, when the prediction function is always less than the actual result, is guaranteed to find the optimal path.

References

[1] 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 G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. https://doi.org/10.1145/3620665.3640366

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759 [cs.NE] https://arxiv.org/abs/1410.0759

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[7] 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. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy

[8] Google. 2017. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla.

[9] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3582016.3582018

[10] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[11] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[12] 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* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[13] Duo Li, Jie Hu, Changhu Wang, Xiangtai Li, Qi She, Lei Zhu, Tong Zhang, and Qifeng Chen. 2021. Involution: Inverting the Inherence of Convolution for Visual Recognition. arXiv:2103.06255 [cs.CV] https://arxiv.org/abs/2103.06255

[14] NVIDIA. 2017. NVIDIA TensorRT: Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt.

[15] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 606–624. https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf

[16] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. OpenAI blog.

[17] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling Deep Learning Memory Access via Tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 701–718. https://www.usenix.org/conference/osdi23/presentation/shi

[18] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. 2022. FreeTensor: A Free-Form DSL with Holistic Optimizations for Irregular Tensor Programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 872–887. https://doi.org/10.1145/3519939.3523448

[19] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[20] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[21] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association, 37–54. https://www.usenix.org/conference/osdi21/presentation/wang

[22] Qiang Zhang, Qiangqiang Yuan, Jie Li, Zhen Yang, and Xiaoshuang Ma. 2018. Learning a Dilated Residual Network for SAR Image Despeckling. arXiv:1709.02898 [cs.CV] https://arxiv.org/abs/1709.02898

[23] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices . In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 6848–6856. https://doi.org/10.1109/CVPR.2018.00716

[24] 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. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[25] 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3503222.3507723