Journal of Systems Architecture 160 (2025) 103359

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

GTA: Generating high-performance tensorized program with dual-task scheduling

Anxing Xie [a,1], Yonghua Hu [a,*], Yaohua Wang [b], Zhe Li [b,c], Yuxiang Gao [a], Zenghua Cheng [a]

[a] School of Computer Science and Engineering, Hunan University of Science and Technology, Taoyuan Road, Xiangtan, 411201, Hunan, China
[b] School of Computer Science, National University of Defense Technology, Deya Road, Changsha, 410073, Hunan, China
[c] Tianjin Institute of Advanced Technology, Huixiang Road, 300459, Tianjin, China

ARTICLE INFO

Keywords:
Mapping
Code generation
Compiler optimization
Tensor computation

ABSTRACT

Generating high-performance tensorized programs for deep learning accelerators (DLAs) is crucial for ensuring the efficient execution of deep neural networks. But, producing such programs for different operators across various DLAs is notoriously challenging. Existing methods utilize hardware abstraction to represent acceleration intrinsics, enabling end-to-end automated exploration of the intrinsics mapping space. However, their limited search space and inefficient exploration strategies often result in suboptimal tensorized programs and significant search time overhead.

In this paper, we propose GTA, a framework designed to generate high-performance tensorized programs for DLAs. Unlike existing deep learning compilers, we first coordinate intrinsic-based mapping abstraction with rule-based program generation strategy, followed by the application of resource-constrained rules to eliminate ineffective tensor program candidates from the search space. Second, we employ a dual-task scheduling strategy to allocate tuning resources across multiple subgraphs of deep neural networks and their mapping candidates. As a result, GTA can find high-performance tensor programs that are outside the search space of existing state-of-the-art methods. Our experiments show that GTA achieves an average speedup of more than 1.88× over AMOS and 2.29× over Ansor on NVIDIA GPU with Tensor Core, as well as 1.49× over Ansor and 2.76× over PyTorch on CPU with AVX512.

1. Introduction

Recently, the successful deployment of machine learning models has revolutionized diverse application domains, such as image recognition [1–3], natural language processing [4–6], and autonomous driving [7–9]. This rapid development has created a demand for generating high-performance tensor programs for deep learning accelerators (DLAs), such as Google TPUs [10], mobile devices [11–13], FPGAs [14–16], and more. To accelerate machine learning, hardware vendors have introduced domain-specific intrinsics for tensor computations, such as NVIDIA's Tensor Cores [17–19] and CPU's AVX512 [20]. This demand has led to the process known as **tensorization** [21], which involves transforming computations using these intrinsic instructions. However, hardware specialization complicates the task of generating high-performance tensorized programs.

To support hardware intrinsic instructions across different accelerators, existing methods [22–24] use unified hardware abstractions to enable end-to-end automatic mapping space exploration. These abstractions not only convert opaque intrinsics into an analyzable format but also bridge the gap between high-level tensor programs and low-level instructions, a process we refer to as **tensorized program generation with automatic mapping optimization**. However, generating high-performance tensorized programs for various DLAs remains challenging for several reasons.

Firstly, inefficient exploration of the intrinsic mapping space leads to substantial overhead in search time. For instance, mapping the 7 loops of a 2D convolution to the 3D of Tensor Core can involve 35 different ways [22]. Current strategies [22,23] treat each mapping candidate equally, generating a tensorized program for each and ultimately selecting the one with the best performance. This approach incurs significant time overhead and is inefficient, as it fails to prioritize more promising candidates during the exploration process. Our experiments reveal that many mapping candidates for a given subgraph ultimately fail to produce high-performance tensorized programs, indicating that a large portion of the explored mappings are ineffective in optimizing performance.

* Corresponding author.
E-mail address: huyh@hnust.cn (Y. Hu).
[1] Part of this work was done at National University of Defense Technology.

https://doi.org/10.1016/j.sysarc.2025.103359
Received 23 November 2024; Received in revised form 8 January 2025; Accepted 30 January 2025
Available online 7 February 2025

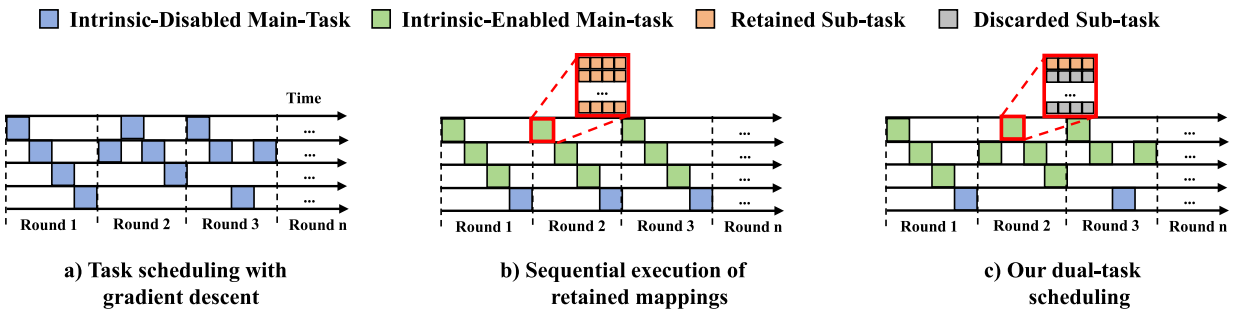

Fig. 1. Comparison of different task scheduling strategies. Part (a): task scheduling with gradient decent. In round 1, all $tasks_i$ are executed sequentially. In subsequent rounds, $tasks_i$ are selectively executed based on the performance gradients calculated from the feedback of each *task*. Part (b): sequential execution of sub-tasks without dual-task scheduling. Part (c): slice the time and prioritize important subgraphs and intrinsic mapping candidates, meaning that not all main-tasks and sub-tasks will be executed. For example, an intrinsic-enabled $main\text{-}task_i$ may contain both retained mapping and discarded mapping candidates. The former will proceed to subsequent tensor program optimization and tuning, while the latter will not participate in further optimization unless they are selected in the next scheduling round. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Secondly, existing rule-based tensor program exploration methods [25] lack the ability to perform automatic tuning and optimization tailored to domain-specific intrinsics. As a result, these methods often fail in auto-tuning and produce suboptimal tensorized programs. To overcome these limitations, there is an urgent need for more efficient exploration of subgraph mapping spaces, along with auto-tuning strategies that can effectively support domain-specific intrinsics, enabling the automatic generation of high-performance tensorized programs.

In this paper, we introduce GTA, a new compiler framework designed to generate high-performance tensorized programs. GTA automatically generates an extensive search space optimized for hardware intrinsics, simultaneously increasing the likelihood of selecting the most efficient mapping configuration. For generating the search space, we employ rule-based strategies to construct a large scheduling search space and apply pruning techniques based on hardware cache resource limitations to eliminate invalid program candidates. Finally, as shown in Fig. 1, for search strategy implementation, we use a dual-task scheduling algorithm to allocate tuning resources across all subgraphs ($main\text{-}task_i$ as shown by the blue box in Fig. 1) in the neural network and their intrinsic mapping candidates($sub\text{-}task_i$ as shown by the orange box and gray box). This algorithm prioritizes subgraphs with greater potential for performance improvement, allocating them more tuning opportunities, while reducing tuning efforts on less promising mapping candidates based on performance feedback, thereby minimizing overall tuning time. In summary, this paper makes the following contributions:

- We integrated intrinsic-based mapping abstraction with a rule-based program generation strategy to expand the search space significantly.
- We developed and implemented an efficient dual-task scheduling strategy for tensorized programs, effectively reducing tuning efforts while enhancing performance.
- We propose a compilation framework called GTA, which supports the generation of high-performance tensorized programs at both the operator level and the full network level on NVIDIA GPUs and CPUs.
- We implemented and comprehensively evaluated the GTA system, demonstrating that the aforementioned techniques outperform state-of-the-art systems across various deep neural networks (DNNs).

2. Background and motivation

2.1. Deep learning compilers

Deep learning compilers [21–32] have emerged as essential tools for bridging the gap between deep learning models and diverse hardware backends. These compilers take model definitions, expressed in frameworks like PyTorch [33] or TensorFlow [34], as inputs and generate efficient code implementations for specific hardware platforms, such as CPUs, GPUs. The compilation process often adopts a progressive multi-layer optimization approach. It begins with the front-end, where neural network models serve as input, and proceeds through intermediate representation (IR) stages. These include graph-level IR [35–39] for structural optimizations and loop-level IR [40–42] for fine-grained transformations. Finally, the back-end generates hardware-specific executable code using traditional compiler techniques, ensuring efficient execution on the target platform.

A key innovation in deep learning compilers is the **compute-schedule** separation first introduced by Halide [43] and adopted by frameworks like TVM [21]. Compute represents the mathematical description of tensor operations, such as addition, convolution, or matrix multiplication, while schedule defines how these operations are executed on hardware. Schedule specifies program transformations, including loop tiling, vectorization, and unrolling, to optimize performance for specific hardware architectures. This decoupling simplifies the representation of tensor computations, enabling flexible optimization strategies tailored to different backends.

Recent advancements [22–24,44] in deep learning compilers focus on leveraging hardware intrinsics to further optimize tensor programs. By integrating intrinsic-specific mapping abstractions, these compilers can directly utilize the specialized instructions of DLAs, such as NVIDIA's Tensor Cores or CPU's AVX512, to achieve higher computational efficiency. These developments mark a shift from general-purpose optimizations to hardware-aware designs, laying the foundation for intrinsic-based mapping strategies.

2.2. Intrinsic-based mapping abstraction

The development of DLAs has led to the creation of specialized instructions [45–48], known as **intrinsics**, designed to enhance the computational efficiency of tensor operations. These instructions serve as essential interfaces between hardware and compilers, enabling optimized execution of key operations like matrix multiplication and data movement.

Intrinsics provide an efficient mechanism for managing kernel operations in tensor programs, typically categorized into compute intrinsics for performing computations and memory intrinsics for data handling [22]. For example, NVIDIA Tensor Cores [17–19] and CPU AVX512 [20] offer specialized intrinsics that allow accelerated matrix and vector operations, respectively, facilitating high-performance computation across various accelerators.

Intrinsic-based mapping abstraction further unifies tensor program optimization by representing diverse intrinsic behaviors in a common, analyzable form. Frameworks like AMOS [22] and TensorIR [23]

Table 1
State-of-the-art compilers/mappings for hardware accelerators.

Name	Mapping Method
❶	
AutoTVM	Hand-written templates + Tuning
Triton	Hand-written templates
❷	
Tiramisu	Polyhedral model
AKG	Polyhedral model + Templates
❸	
Ansor	Generated rules + Tuning
XLA	Templates and rules
Heron	Constraint-based rules + Tuning
MetaSchedule	Generated rules + Tuning
❹	
UNIT	Analyzable abstraction + Tuning
ROLLER	Tile abstraction + Construction policy
AMOS	Analyzable abstraction + Tuning
TensorIR	Analyzable abstraction and generated rules + Tuning
❺	
Hidet	Task-mapping + Post-scheduling fusion
EINNET	Derivation-based + Tuning
TensorMap	Reinforcement learning + Tuning
GTA	**Analyzable abstraction and generated rules + Tuning**

leverage this approach to directly map software operations to hardware intrinsics, supporting automated generation and transformation of tensorized programs. This abstraction broadens the search space for high-performance configurations by identifying fundamental software-to-hardware mappings, thus enhancing optimization potential across different hardware backends.

2.3. Tensor program generation strategy

In Table 1, we summarize state-of-the-art compiler mapping techniques used to generate optimized tensor programs on hardware accelerators. Most existing compilers leverage programmable intrinsics as part of their mapping strategy, enabling developers to focus on high-level optimization while the compiler handles low-level architectural details. These mapping methods streamline tensor program generation by abstracting hardware-specific operations, thereby enhancing both efficiency and portability.

Specifically, we categorize the state-of-the-art compilers/mappers for DLAs into five main approaches:

❶ **Hand-written mapping**: Hand-written mapping [29,49] requires developers to manually define mappings for tensorized programs using compiler-provided tensorize interfaces. This approach enables fine-grained optimization, especially for specialized hardware like NVIDIA Tensor Cores. However, it demands significant expertise and high development costs, as developers must continually rewrite templates to support new operators and accelerators [50–52]. While hand-written mapping can achieve high performance for specific workloads, its lack of scalability and adaptability limits its effectiveness compared to more automated methods.

❷ **Polyhedral model mapping**: Polyhedral model mapping [28,32,53–56] provides a powerful strategy for optimizing tensor programs by restructuring execution and managing complex memory dependencies. In the realm of tensor program compilation, this approach plays a critical role in handling intricate memory structures and optimizing execution. For example, AKG [32] leverages polyhedral scheduling to restructure execution order through new linear relationships, effectively eliminating inter-loop dependencies. This method is particularly advantageous for hardware like TPUs, where enhancing parallel computation is essential. By exploring a broader range of affine transformations compared to methods such as TVM [21], polyhedral mapping optimizes performance for diverse workloads. However, the model's inherent complexity limits its general applicability, making it less feasible for simpler or less resource-intensive tasks.

❸ **Rule-based mapping**: Rule-based mapping [24–27,57] generates efficient tensor programs through predefined scheduling primitives, streamlining tensor program creation without user-defined templates. This approach leverages scheduling techniques like loop tiling, fusion, and vectorization, as demonstrated by frameworks like Ansor [25], which automatically create search spaces using these rules. This method simplifies tensor program generation in deep learning applications. However, it also has limitations: users must ensure that the predefined rules align with the specific operators and hardware, or the generated programs may fail to achieve optimal performance.

❹ **Analyzable abstraction mapping**: Analyzable abstraction mapping [22,23,44,58,59] unifies tensor program optimization by abstracting diverse hardware intrinsic behaviors into a common representation, facilitating efficient mapping and transformation of tensorized programs. Examples like AMOS and TensorIR establish direct mappings between software and hardware, guiding the automated generation of tensorized programs. This approach broadens the scope of exploration by identifying foundational software-to-hardware combinations, increasing the potential for discovering optimized mappings.

❺ **Other mapping**: Other mapping methods [13,40,60,61] reformulate deep learning optimization problems using strategies from other domains to enhance efficiency. For example, CoSA [56] and Heron [24] convert the scheduling space search into a constrained optimization problem and leverage solvers to rapidly explore the space. Alternatively, TLM [62] and Soter [63] treat tensor program exploration as a language model generation task, where tensor programs are represented as sequences and tunable parameters as language tokens. Specifically, they leverage a large language model (LLM) to generate these tokens for tunable parameters, enabling efficient exploration of mapping schemes and more effective optimization of tensor programs.

Building on this foundation, we reviewed five primary mapping approaches used for deep learning accelerators: hand-written, rule-based, polyhedral model, analyzable abstraction, and other mapping methods. Each approach brings unique advantages—hand-written and rule-based mappings allow fine-tuned performance but require extensive manual intervention or rigid predefined rules, while polyhedral and analyzable abstraction mappings offer more automated solutions but are challenged by complexity and limited applicability. Methods borrowing from other domains, such as optimization solvers and language models, open new directions but may lack consistency across diverse hardware. In summary, intrinsic-based mapping abstraction offers a unified framework for optimizing tensor programs across diverse hardware accelerators by abstracting hardware intrinsic behaviors into a common representation. Systems like AMOS and TensorIR leverage this approach to enable efficient and adaptable mappings for tensorized programs.

Despite these advances, significant challenges remain in achieving flexible, high-performance mappings that are adaptable to new hardware accelerators, such as the inefficiency of existing approaches in handling diverse architectural constraints and their inability to effectively explore large and complex search spaces. To better illustrate our motivation, we present an example to illustrate the specific challenges within existing analyzable abstraction mapping systems, motivating the development of our approach.

Mapping intrinsic instructions onto hardware accelerators poses significant challenges due to the vast number of possible configurations and their impact on performance. The process of selecting the optimal mapping for intrinsic instructions, such as those used in Tensor Cores, is complex, given the numerous potential mapping candidates. Each mapping choice can critically affect performance factors like data locality and parallelism. For example, as shown in Table 2, AMOS identified 35 distinct ways to map the seven loops of a 2D convolution onto the 3D loops of the Tensor Core. Exhaustively exploring all configurations is inefficient and rarely yields substantial performance gains. Thus, a more efficient approach is required, one that prioritizes the most promising mappings to reduce search overhead and maximize performance.

Table 2
Mapping candidates choices. This example maps a 2D convolution index to Tensor Core index (type: float16). Space loops: *n, k, p, q,* i_1, i_2; Reduction loops: *rc, rr, rs,* r_1. The mapping choices can be categorized into basic mapping and complex mapping. Basic mapping means selecting only one choice at a time, while complex mapping allows multiple choices to be combined for mixed mapping.

	mapping1	mapping2	mapping3	mapping4	mapping5	mapping6	mapping7
i1	n	n	n	p	p	q	q
i2	k	k	k	k	k	k	k
r1	rc	rr	rs	rc	rs	rc	rr
Choices	0/1	0/1	0/1	0/1	0/1	0/1	0/1

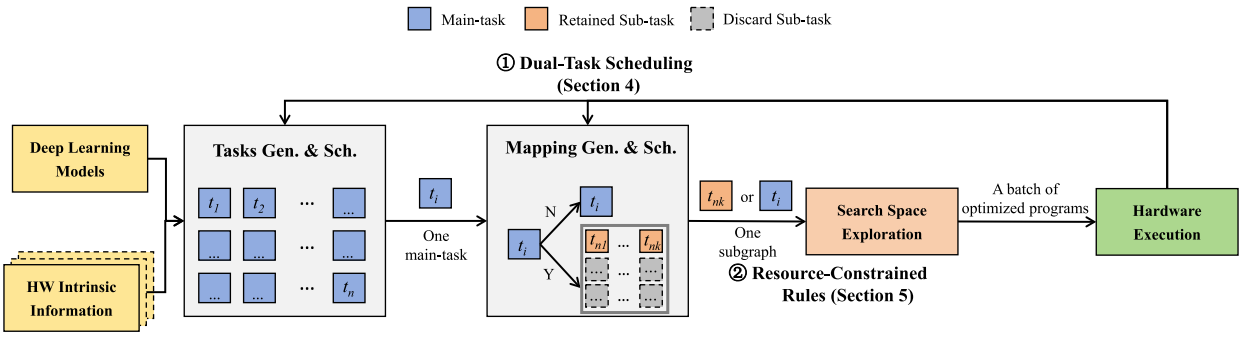

Fig. 2. The compilation flow of GTA. t_n denotes the nth non-intrinsic main-task (blue box), and t_{nk} denotes the kth mapping candidate of nth intrinsic-enabled main-task (orange box). All mapping candidates are ranked and executed based on performance feedback. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

A second challenge lies in the scheduling of tensor programs, which often lacks consideration for DLAs intrinsics. Existing systems do not sufficiently incorporate these intrinsics when generating the scheduling search space, limiting their ability to optimize tensorized programs for specialized hardware. To address this, a more comprehensive approach to scheduling is needed, integrating primitives like tiling, fusion, and vectorization that are tailored to the unique characteristics of DLAs. Without such a targeted approach, the scheduling search space cannot fully leverage the potential of available mappings, thereby constraining the system's capacity to produce high-performance programs.

3. GTA overview

To address the aforementioned issues, we propose GTA, a compilation framework designed to automatically generate high-performance tensorized programs for specialized hardware. As shown in Fig. 2, it takes deep neural networks (DNNs) as input, converting them into computation graphs represented as directed acyclic graphs (DAGs). In these graphs, each node corresponds to a tensor operation, and each edge denotes a producer–consumer relationship between operations. To handle the complexity of large computational graphs, GTA partitions the DNN's computation graph into smaller, manageable subgraphs using Relay's operator fusion algorithm, which has minimal performance impact due to the layer-by-layer structure of DNNs ($t_1, t_2, \ldots, t_n$ in Fig. 2).

To maximize performance across multiple subgraphs, GTA dynamically prioritizes subgraphs and mapping candidates most likely to enhance end-to-end efficiency. It uses a **dual-task scheduling** approach (detailed in Section 4) that allocates tuning time across both subgraph and mapping candidate levels. By allocating varying amounts of time to different subgraphs and probabilistically discarding less efficient candidates based on performance feedback, dual-task scheduling helps avoid wasted tuning resources on low-impact mappings.

Additionally, **resource-constrained rules** (explained in Section 5) guide program generation on both DLAs and general-purpose accelerators. GTA designs these rules by abstracting common architectural characteristics across DLAs, such as coarse-grained hardware intrinsics (e.g., WMMA in Tensor Core) and dedicated scratchpad memory (e.g., Unified Buffer in TPU). This design allows GTA to efficiently leverage hardware-specific features, optimizing tensorized programs to fully exploit the underlying hardware capabilities.

4. Dual-task scheduling

Most existing compiler frameworks adopt a performance-aware tuning strategy to fine-tune generated programs, a method proven effective by systems such as Ansor and AMOS. For example, Ansor refines its cost model by updating task weights based on feedback from each search iteration, while dynamically allocating subgraph trials. Building on this approach, when multiple intrinsic instruction mapping options are available, feeding performance results of each mapping back into the front-end further enhances the framework by enabling seamless co-design between the front and back-end stages.

To optimize tuning resource allocation, a DNN can be decomposed into multiple independent subgraphs (e.g., conv2d + ReLU). For some subgraphs, spending time on tuning may not significantly improve the overall network performance. This may occur when a subgraph is not a performance bottleneck, or when any tuning yields only marginal gains. Similarly, a subgraph may have multiple intrinsic mapping candidates, but further tuning on certain mappings may not result in meaningful improvements. This is often because certain mapping schemes exhibit inefficient memory access patterns, limiting their ability to leverage the unique features of the underlying hardware and thereby restricting the potential for significant performance gains.

To illustrate the dual-task scheduling (DTS) process, we use ResNet18 as an example. After splitting ResNet18 into subgraphs, there are 24 unique subgraphs, most of which are convolution layers with varying shape configurations (e.g., input size, kernel size, stride). Following Ansor's task scheduling methodology, we define a task as the process of generating high-performance programs for each subgraph. Thus, optimizing a single DNN like ResNet18 requires completing multiple tasks (e.g., 24 tasks for ResNet18).

To efficiently allocate tuning resources across these tasks, GTA employs a **DTS** approach. This method dynamically assigns varying amounts of time to different subgraphs and probabilistically discards inefficient mapping candidates based on program performance feedback. DTS operates on two levels: the subgraph level and the mapping candidate level, helping GTA focus tuning resources on the most impactful configurations and avoid spending time on low-impact mappings.

As shown in Fig. 1, the DTS iteratively allocates tuning resources to different tasks. In each round, the first step selects a subgraph for program generation, GTA generates a set of intrinsic-compatible mapping

Algorithm 1: Dual-Task Scheduling

Input:
G: native deep learning neural network
target: target hardware platform
trials: total tuning counts
MEASURE_NUM: number of measures per round
Output: *best_tasks*: best performance tasks

1 **Function** *dual_scheduling*
2 Initialize local variables $B_{latency}$, B_{task}, $T_{latency}$, C_{task}, $C_{samples}$;
3 $tasks = extract\_tasks(G, target)$;
4 **while** $C_{trials} < trials$ **do**
5 $tid = gradient\_scheduling(tasks, T_{latency})$;
6 $M_{candi} = match\_intrinsic(tasks[tid], target)$;
7 **if** M_{candi} *not NULL* **then**
8 **for** $C_{mapping}$ ***in*** M_{candi} **do**
9 **if** $C_{samples}$ **then**
10 **if** $C_{mapping}$ ***not in*** $C_{samples}$ **then**
11 continue;
12 **end**
13 **end**
14 $latency = tasks[tid].\text{tune}(C_{mapping})$;
15 $T_{latency}.append(latency)$;
16 **if** $latency < B_{latency}$ **then**
17 $B_{latency}[tid] = latency$;
18 $B_{task}[tid] = tasks[tid]$;
19 **end**
20 **end**
21 $C_{sample} = probability\_sample(T_{latency})$;
22 C_{trials} += *MEASURE_NUM*
23 **end**
24 **end**
25 **return** B_{task};

candidates for the intrinsic-enabled $task_i$. This effectively breaks the main-task into several sub-tasks (as shown by the orange box in Fig. 1). The second step then generates a batch of promising programs for these sub-tasks and measures their performance on hardware. Each round is defined as one unit of time resource. When a time resource is allocated to a task, the task gains the opportunity to generate and measure new programs, increasing the chance of discovering better-performing ones.

In the following section, we introduce the formulation of the scheduling problem and our solution.

4.1. Problem formulation

In defining the scheduling problem, we divide DTS into two types of tasks: **main-tasks** and **sub-tasks**. In this framework, a DNN can be split into several subgraphs (main-tasks). If the computation type, data type, and computation shape of a main-task meet the limitations required for utilizing hardware intrinsic resources, multiple intrinsic mapping candidates will be generated for the main-task. Each of these intrinsic mapping candidates is referred to as a sub-task. A main-task represents a process performed to generate high-performance programs for a subgraph, meaning that optimizing a single DNN requires completing dozens of main-tasks. And related notions used in this paper are shown in Table 3.

We define $m_i(t)$ as the minimum execution time required for the ith main-task at time t, and $m_{ik}(t)$ as the execution time of the kth mapping scheme for the ith main-task. The optimal execution time for subgraph i is represented as $\min(m_{i1}(t), m_{i2}(t), \ldots, m_{ik}(t))$. The end-to-end execution time of the entire network, denoted by $G(m_1(t), m_2(t), \ldots, m_n(t))$, represents the aggregate time across all main-tasks. Our objective is to minimize this function to achieve the lowest possible overall execution time for the DNN. Thus, the objective function is

Table 3
Notations.

Notation	Description/Definition
Main-task	Subgraph process for generating high-performance programs
Sub-task	Intrinsic mapping candidate satisfying hardware constraints
Δt	Small backward window size
N_i	The set of similar task of i
C_i	The number of floating point operation in task i
V_k	The number of floating point operation per second we can achieve in task k
$B_{latency}$	Best mapping latency set of tasks
B_{task}	Best mapping tasks set of DNN
C_{sample}	Samples selected from all mappings
C_{trials}	Current number of trials
$C_{mapping}$	Current mapping selection
G	Native neural network
$m_i(t)$	Minimum execution time for$i$th task
$m_{ik}(t)$	Execution time of kth mapping for $m_i(t)$
$T_{latency}$	Latency set of all tasks
M_{candi}	Set of all mapping candidates
α_k	Sampling probability of mapping k
β	Hyperparameter for increasing probability
ω_i	Number of appearances of task i in the network

defined as:

$$f(G) = \sum_{i=1}^{n} (\omega_i \times max(\beta(\alpha_1 \cdot m_{i1}(t), \alpha_2 \cdot m_{i2}(t), ..., \alpha_k \cdot m_{ik}(t)))) \quad (1)$$

Let ω_i denote the number of appearances of main-task i in the network, where i is the main-task index. If a main-task has already met its latency requirement, no additional tuning resources are allocated to it. The variable α_k represents the sampling probability assigned to sub-task k. Unlike other frameworks, our approach introduces probabilistic allocation for intrinsic mapping candidates (sub-task). Once performance feedback for all mapping candidates of a subgraph is received, sampling probabilities are assigned based on time cost, candidates with lower time costs are assigned higher probabilities, while those with higher time costs receive lower probabilities. We also introduce a hyperparameter β to adjust sampling probabilities for specific mapping candidates, helping to avoid convergence on locally optimal solutions.

4.2. Optimizing with gradient and probability

Inspired by the gradient descent-based task scheduling approach presented in [25], we propose a DTS algorithm (Algorithm 1) that combines gradient descent with probability-based selection to efficiently optimize the objective function. Starting from the current allocation t, the algorithm approximates the gradient of the objective function, $\frac{\partial f}{\partial t_i}$, and identifies the primary task i by maximizing the absolute gradient, defined as $i = \arg\max_i |\frac{\partial f}{\partial t_i}|$. This gradient approximation serves as the foundation for selecting the main-task with the highest potential impact.

$$\frac{\partial f}{\partial t_i} \approx \frac{\partial f}{\partial m_i} \left(\alpha \frac{\Delta m}{\Delta t} + (1-\eta) \left(\min \left(-\frac{m_i(t_i)}{t_i}, \theta \frac{C_i}{\max_{k \in N(i)} V_k} - m_i(t_i) \right) \right) \right) \quad (2)$$

where $\Delta m = m_i(t_i) - m_i(t_i - \Delta t)$ and other variables are defined in Table 3. The parameter η and θ control the weight to trust some predictions.

GTA initializes the algorithm with $t = 0$ and begins with a round-robin warm-up phase, resulting in an initial allocation vector of $t = \{1, 1, \ldots, 1\}$. After the warm-up, as shown in line 5 of Algorithm 1, the gradient for each main-task is computed, and the main-task with the maximum absolute gradient, $i = \text{argmax}_i |\frac{\partial f}{\partial t_i}|$, is selected. A tuning time unit is then allocated to this main-task, updating its allocation to $t_i = t_i + 1$. The optimization process continues until the tuning time budget is exhausted.

Afterward, GTA searches for a hardware intrinsic that matches the specified main-task. Once a suitable set of hardware intrinsics is identified, tensor programs are generated for all mapping candidates, serving as a warm-up for the sub-task. This warm-up allows GTA to select the most promising mapping candidates by assigning probabilities based on their performance feedback. In subsequent rounds, only mapping candidates prioritized by their previously assigned probabilities are executed. This selective exploration avoids spending time on inefficient candidates, enhancing tuning efficiency and allowing higher-potential candidates more opportunities for optimization.

The *probability_sample* algorithm, as called in line 21 of Algorithm 1, is designed to probabilistically select mapping candidates for further analysis and optimization. We first introduce the notation: let $R = \{r_1, r_2, \ldots, r_n\}$ represent the set of all mapping results, where r_i denotes the ith result with a performance value $V(r_i)$.

The total weight W is calculated by considering each result's inverse performance value, normalized with respect to the maximum performance in R, as follows:

$$W = \sum_{r_i \in R} \frac{1}{V(r_i)} \cdot \frac{1}{\max_{r_j \in R} \frac{1}{V(r_j)}}$$

This ensures that weights are scaled and relative to the most performant candidate in the result set R. Using this normalized total weight W, the initial probability assigned to each result r_i is given by:

$$P(r_i) = \frac{\frac{1}{V(r_i)} \cdot \frac{1}{\max_{r_j \in R} \frac{1}{V(r_j)}}}{W}$$

To encourage exploration, the algorithm applies a probability increase factor β to selected results. The probability adjustment is defined by weighting the original probability $P(r_i)$ with an exploration boost:

$$P'(r_i) = \frac{(1+\beta) \cdot P(r_i)}{\sum_{r_j \in R}(1+\beta_j) \cdot P(r_j)}$$

Here, β_j is a task-specific exploration factor, applied selectively to candidates r_j, where $\beta_j = \beta$ for selected candidates and $\beta_j = 0$ otherwise. The inclusion of the initial probability $P(r_j)$, derived from each candidate's performance value $V(r_j)$, serves as the foundation of the adjusted probabilities. This ensures that $P'(r_i)$ retains the relative importance of each candidate while allowing selective exploration through β_j.

The normalization term, $\sum_{r_j \in R}(1 + \beta_j) \cdot P(r_j)$, ensures that the adjusted probabilities remain valid and sum to 1. By combining the task-specific exploration factor with the initial performance-weighted probability $P(r_j)$, this formula balances exploitation of high-priority candidates with exploration of less performant options. Furthermore, $P(r_j)$ prevents the adjustment from overly concentrating on a small subset of candidates, promoting diversity and fairness across the result set R.

Finally, the algorithm selects the top N results based on the adjusted probabilities $P'(r_i)$. The selection process is expressed as:

$$\{r_i\}_{i=1}^{N} = \mathrm{Top}_N \left(P'(r_1), P'(r_2), \ldots, P'(r_n) \right), \tag{3}$$

where N is dynamically determined based on a fraction of the total result set R, denoted by $N = \lceil \kappa \cdot |R| \rceil$, and $\kappa \in (0, 1]$ is a user-defined parameter controlling selection size.

5. Resource-constrained rules

Existing exploration-based methods face significant challenges in both performance and scalability, primarily due to two factors. First, although the design space is vast, it contains numerous inefficient kernels. For example, in the GEMM operation with dimensions $512 \times 768 \times 3072$ (used in GPT-1 on Tensor Core), the kernel space size reaches $O(10^{16})$, with over 90% of the kernels being inefficient [63,64].

Table 4
Resource-constrained rules and related conditions.

No.	Rule	Condition
R1	Multi-Level Tiling	HasDataReuse(R, i) & HasMultiLevelCache(R, i)
R2	Set Multi-Scope	HasDataReuse(R, i) & HasMultiScopeCache(R, i)
R3	Fuse Main Op	HasStagesFused(R)
R4	Fuse Output Op	HasStagesFused(R)
R5	AddMemLimit	HasDSM(R)[a]
...	Ansor Defined Rule[b]	...

[a] DSM: dedicated scratchpad memory.

[b] Ansor [25].

Second, current approaches are largely tailored to general-purpose processors and lack consideration for specific architectural constraints. This highlights the need to construct a high-quality kernel design space to effectively reduce inefficient exploration and improve overall performance.

To address these challenges, GTA's implementation of resource-constrained generation rules is based on existing open-source code for DLAs and general-purpose accelerators [22,25]. In particular, the DLA-specific rules are adapted to leverage hardware intrinsics and dedicated scratchpad memory (DSM) efficiently. From a programmer's perspective, DLAs, in contrast to general-purpose accelerators, feature coarse-grained hardware intrinsics (e.g., WMMA in Tensor Core) and user-programmable dedicated DSM (e.g., Unified Buffer in TPU). Based on these existing implementations, we made targeted modifications to better align the rules with the search strategies and optimization methods proposed in this work. Table 4 summarizes five key generation rules that GTA employs to optimize data movement, operation fusion, and memory management in DLAs. Each rule addresses specific challenges to enhance computational efficiency and resource utilization. The following is a detailed description of each rule:

Rule-R1 generates multiple nodes for data movement between different levels of on-chip DSMs. To apply this rule, GTA first checks for data reuse opportunities and verifies if the DLA has multiple DSM levels (e.g., Tensor Core provides two levels of DSMs for WMMA fragments and shared memory). If these conditions are met, GTA inserts $cache_{read}$ primitives for the node and its producers to facilitate data movement.

Rule-R2 marks the data storage scope for each operation within the DSM hierarchy. To apply this rule, GTA first checks for data reuse opportunities and verifies whether the DLA provides multiple DSM scopes for different data types. If these conditions are satisfied, GTA assigns $cache_{write}$ primitives to the node and $cache_{read}$ primitives to its producers, ensuring that data is efficiently stored and accessed within the appropriate DSM levels.

Rule-R3 enables the fusion of main operations within a subgraph by identifying opportunities to combine operations with shared data dependencies. This reduces data movement overhead and improves computational efficiency. When multiple stages are fused, GTA inserts the appropriate primitives to implement the fusion, streamlining the execution flow.

Rule-R4 focuses on fusing output operations within a computational graph. Similar to Rule-R3, it targets operations that can be combined to minimize data transfer costs and enhance throughput. By analyzing data flow between operations, GTA inserts necessary primitives to achieve output fusion, resulting in a more compact and efficient execution structure.

Rule-R5 constrains memory usage for operations that utilize dedicated scratchpad memory (DSM). By evaluating each operation and its memory requirements, GTA ensures memory limits are respected, preventing allocations from exceeding hardware capacity, which could lead to inefficient execution. This rule helps maintain an efficient memory allocation strategy, optimizing overall resource utilization.

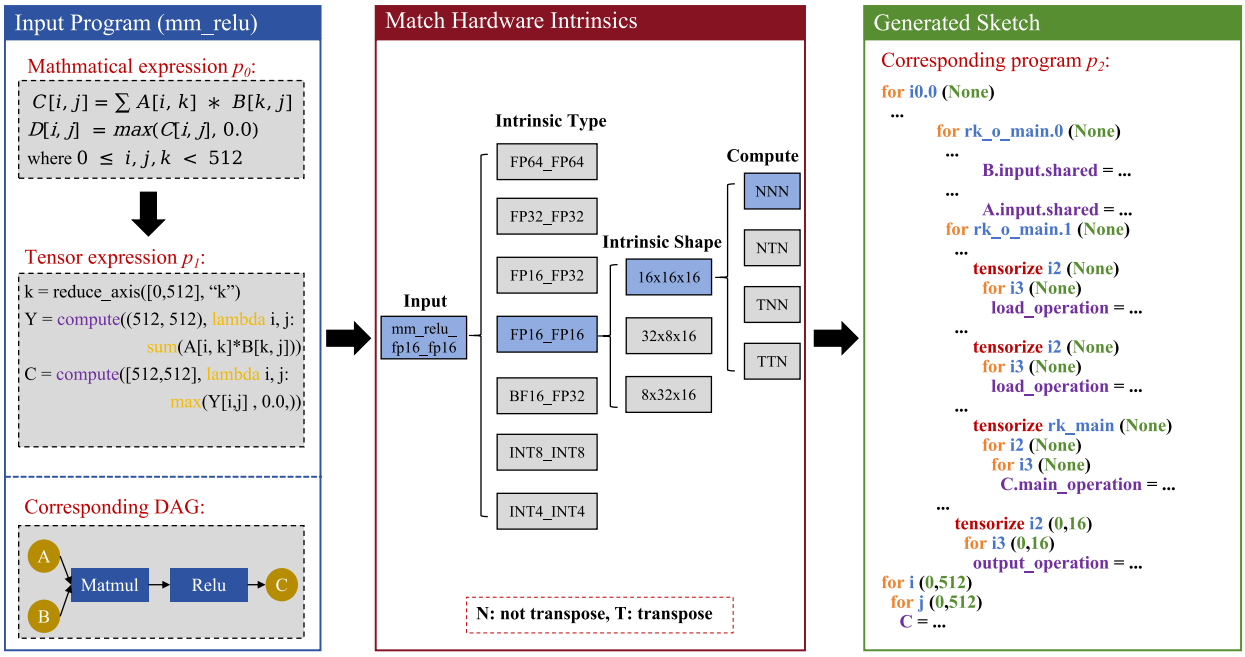

Fig. 3. An illustrative example of tensorized program generation for a GEMM-ReLU operator, demonstrating the transformation of the input program from a mathematical expression (p_0) to a tensor expression (p_1) written in a domain-specific language using TVM. The process further includes intrinsic matching based on the type and shape of the input operator to select and generate intrinsic mapping candidates, followed by the application of resource-constrained rules to guide the creation of a tensorized program sketch (p_2).

An example. Fig. 3 illustrates how resource-constrained rules are applied during tensorized program generation. Starting from the input program written as a mathematical expression (p_0), the process converts it into a tensor expression (p_1) using domain-specific language (DSL) in TVM. The intrinsic matching step leverages compute abstraction and memory abstraction, as proposed in AMOS [22], to complete the software-hardware mapping generation. This process selects and generates intrinsic mapping candidates by analyzing the operator's computation type, data type, and memory access patterns based on its shape and hardware-specific constraints. Subsequently, resource-constrained rules play a critical role in guiding the generation of the tensorized program sketch, ensuring efficient utilization of hardware intrinsic functions while respecting memory and architectural constraints. Specifically, the derivation for generated rules and the transformed program can be expressed as:

$$input\ p_1 \rightarrow M_{candi} \rightarrow o(S_0, i=3) \overset{R2}{\rightarrow} o(S_1, i=3) \overset{R1}{\rightarrow} o(S_2, i=2) \overset{R3}{\rightarrow} \ldots \overset{R5}{\rightarrow} output\ p_2 \tag{4}$$

We defined the state as $o = (S, i)$, where S represents the current partially generated sketch program for the DAG, and i denotes the index of the node currently being transformed. For each rule, if the application conditions are met, the rule is applied to $o = (S, i)$, resulting in a new state $o' = (S', i')$, where $i' \le i$. This ensures that the index i (indicating the transforming node) decreases monotonically. A state reaches a terminal condition when $i = 0$. During the enumeration process, multiple rules may be applicable to a single state, generating several succeeding states. Additionally, a single rule can produce multiple succeeding states under certain conditions.

6. Implementation

In this section, we delve into the technical details in our implementation. GTA extends TVM, a end-to-end deep learning compiler, to support loop scheduling and generate high-performance programs with intrinsic instructions.

Task Generation. To mitigate the issue of search space explosion, compilers typically divide the large computational graph of a DNN into smaller subgraphs. Notably, for some subgraphs, spending time on tuning may not significantly enhance the end-to-end performance of the DNN. In this work, we adopt TVM's subgraph partitioning strategy to divide the input DNN into multiple smaller subgraphs, referred to as main-tasks. A main-task is considered a process executed to generate high-performance programs for a subgraph. TVM categorizes operators into four types: injective (e.g., add operations), reduction (e.g., sum operations), complex-out-feasible (e.g., matrix multiplication, where element-wise mappings can fuse to the output), and opaque (e.g., sort operations, which cannot be fused). Subgraph fusion is then performed based on predefined generic rules.

Mapping Generation and Scheduling. At each iteration, based on the intrinsic mapping generation approach described in AMOS [22], main-tasks can be classified into intrinsic-disabled and intrinsic-enabled tasks. For intrinsic-disabled main-tasks, we adopt Ansor's [25] compilation optimization to generate programs. In contrast, for intrinsic-enabled main-tasks, GTA optimizes task scheduling based on gradients and probabilities. This algorithm prioritizes subgraphs with higher potential for performance improvement, allocating them more tuning opportunities while reducing efforts on less promising mapping candidates based on performance feedback. Slice the time and prioritize important subgraphs and intrinsic mapping candidates, meaning that not all main-tasks and sub-tasks will be executed. For example, an intrinsic-enabled $main\text{-}task_i$ may contain both retained mapping and discarded mapping candidates. The former will proceed to subsequent tensor program optimization and tuning, while the latter will not participate in further optimization unless they are selected in the next scheduling round.

Search Space Exploration. Subsequently, GTA applies resource-constrained rules and existing derivation rules (Table 4) to each subgraph under the guidance of a genetic algorithm [25]. During this process, tens of thousands of tensor programs are generated, and cost model is employed to filter out the most promising candidates with near-optimal performance. These selected candidates are then executed on the target hardware to identify the tensor program with the best performance.

7. Evaluation

7.1. Evaluation platforms

Our experiments were conducted on two distinct hardware platforms to evaluate the performance of the proposed GTA framework:

- **NVIDIA GPUs:** We performed experiments on two NVIDIA GPUs, specifically the RTX 3060 and A100, which are equipped with Tensor Cores optimized for deep learning tasks. The RTX 3060 represents a consumer-grade GPU, while the A100 is a data center-grade GPU designed for high-performance computing.
- **AMD CPU:** We evaluated the performance on an AMD Ryzen 7 7840H CPU,[2] which supports advanced SIMD (Single Instruction, Multiple Data) instructions, enabling efficient vectorized computations. This CPU platform provides a competitive environment for testing AVX512-like optimizations in general-purpose processors, allowing us to benchmark GTA's performance on non-GPU hardware.

7.2. Evaluated benchmarks

We evaluate the performance of GTA using both deep learning (DL) operators and complete neural network models.

- **Operator-Level Evaluation:** We select nine widely-used operators for this evaluation: General Matrix Multiplication (GEMM), 1D convolution (C1D), 2D convolution (C2D), 3D convolution (C3D), transposed 2D convolution (T2D), dilated convolution (DIL), batch matrix multiplication (BMM), General Matrix–Vector multiplication (GEMV), and scan (SCAN). For each operator, we test 6–10 different shape configurations and report the geometric mean of speedups normalized to GTA. The shape configurations are consistent with those used in Ansor and AMOS to ensure a fair comparison.
- **Network-Level Evaluation:** We benchmark six commonly-used neural network models: ResNet18 and ResNet50 [1], BERT (base configuration) [65], MI-LSTM [66], MobileNet-V1 [67], and ShuffleNet [11]. For each model, we evaluate the performance with batch sizes of 1 and 16.

7.3. Comparison baselines

Our evaluation compares GTA against three state-of-the-art automatic generation methods (AutoTVM [49], Ansor [25] (v0.8), and AMOS [22] (commit: 0f39742)) as well as two vendor-optimized, hand-tuned libraries (cuDNN (v11.6) and PyTorch (v1.13.1, v2.0.1)):

- **AutoTVM:** This method uses hand-written templates to support all three selected platforms, demonstrating high performance across a range of baseline operators.
- **AMOS:** AMOS systematically explores various mappings of loop iterations to DLAs, representing the state-of-the-art for operators with multiple feasible mappings, such as C1D and C2D.
- **Ansor:** As a leading method for GPU CUDA Core and CPU code generation, Ansor does not support DLAs like Tensor Core due to architectural limitations. However, comparing GTA with Ansor highlights the benefits of leveraging DLA-specific features in tensor program generation.
- **PyTorch:** PyTorch, a widely-used deep learning framework, serves as a strong baseline for evaluating GTA's ability to outperform standard hand-tuned implementations in practical deep learning applications. Our experiments include both PyTorch 1.13, which relies heavily on vendor-optimized libraries such as cuDNN and cuBLAS for high-performance computations, and PyTorch 2.0, which introduces the TorchInductor compiler.

For a fair comparison, we evaluate AutoTVM, Ansor, AMOS, and GTA with up to 200 measurement trials per test case and report the best performance achieved. For the vendor-optimized libraries on Tensor Core, we use PyTorch, which relies on hand-optimized libraries such as cuDNN to support various types of operators. These optimized libraries serve as strong baseline references for evaluating the performance of GTA.

7.4. Experimental results

We evaluate the performance of GTA on both operators and neural networks, comparing it against several baselines on two DLAs: GPU Tensor Cores and CPU AVX512. To further demonstrate the effectiveness of GTA, we analyze the quality of the generated search spaces and the efficiency of the exploration process. Finally, we highlight how the dual-task scheduling strategy significantly reduces compilation time by dynamically prioritizing subgraphs and mapping candidates, effectively cutting down unnecessary search efforts.

7.5. Operator performance

Tensor Core. First, we compare GTA with PyTorch, which relies on hand-optimized libraries such as cuDNN to support various operators. Fig. 4 shows the results for all operators with batch size 1 on the NVIDIA RTX 3060. GTA consistently outperforms PyTorch across all operators, achieving an average 2.44× geometric mean speedup. The speedup is attributed to GTA's comprehensive software-hardware mapping exploration, which contrasts with PyTorch's use of fixed mappings from hand-optimized libraries, often leading to suboptimal performance.

Next, we evaluate the performance on the NVIDIA A100 GPU for various operator. As shown in Fig. 9, GTA achieves 1.26×, 5.24×, and 1.93× geometric mean speedup over Ansor, PyTorch, and AMOS, respectively. The significant improvement is due to GTA's ability to effectively utilize the high-performance Tensor Core units through enhanced mapping and scheduling strategies.

We also compare GTA with state-of-the-art compilers on RTX 3060 using the C2D in NCHW layout. We test all convolution layers from ResNet18 (a total of 12 configurations, labeled as C0–C11). These configurations are standard benchmarks from well-known networks. The results are shown in Figs. 4, 5, and 6. GTA achieves speedups of 1.85×, 1.76×, and 2.10× over Ansor, AMOS, and hand-tuned PyTorch, respectively. Compared to Ansor, GTA leverages high-performance Tensor Core units alongside efficient auto-scheduling strategies, resulting in better optimization. In contrast to AMOS, GTA employs DTS to efficiently explore the scheduling space, reducing search time while enhancing program performance. Moreover, AMOS cannot utilize resource-constrained rules for shared memory allocation, leading to the generation of some tensor programs that exceed hardware resource limits. This limitation reduces AMOS's capability to achieve higher-performing programs.

AVX512. On the AMD CPU platform, we utilize hardware abstraction for AVX512 intrinsics (specifically for matrix–vector multiplication) and apply GTA to generate code for C2D. As shown in Fig. 7, GTA achieves 1.49× and 2.76× performance improvements over Ansor and PyTorch, respectively. GTA's advantage stems from combining high-performance AVX512 intrinsics with efficient auto-scheduling strategies, leading to superior program optimization compared to baseline methods.

[2] Intel CPUs also support AVX512 instructions and could be used for similar experiments.

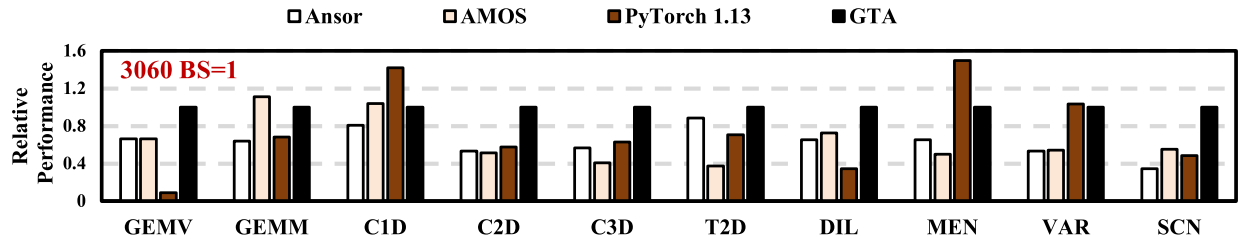

Fig. 4. Single operator performances comparison on NVIDIA RTX 3060.

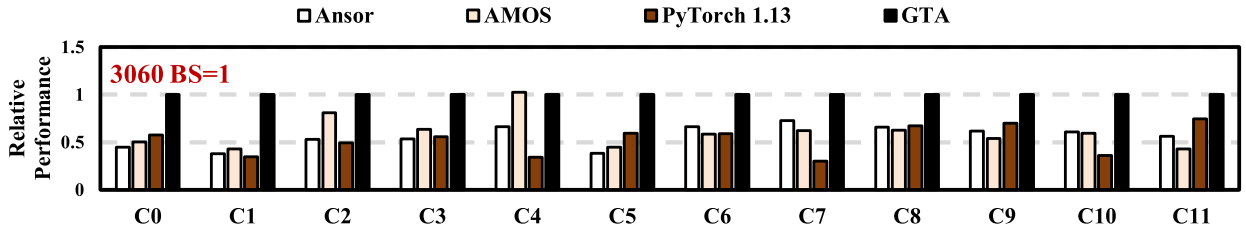

Fig. 5. Performance comparison of C2D on NVIDIA RTX 3060 with batch size = 1, using all convolution layers from ResNet18 (12 configurations, labeled as C0–C11).

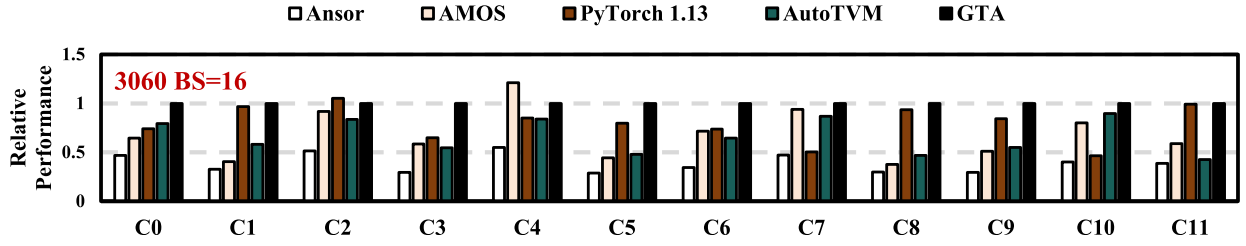

Fig. 6. Performances comparison for C2D on NVIDIA RTX 3060 with batch size = 16.

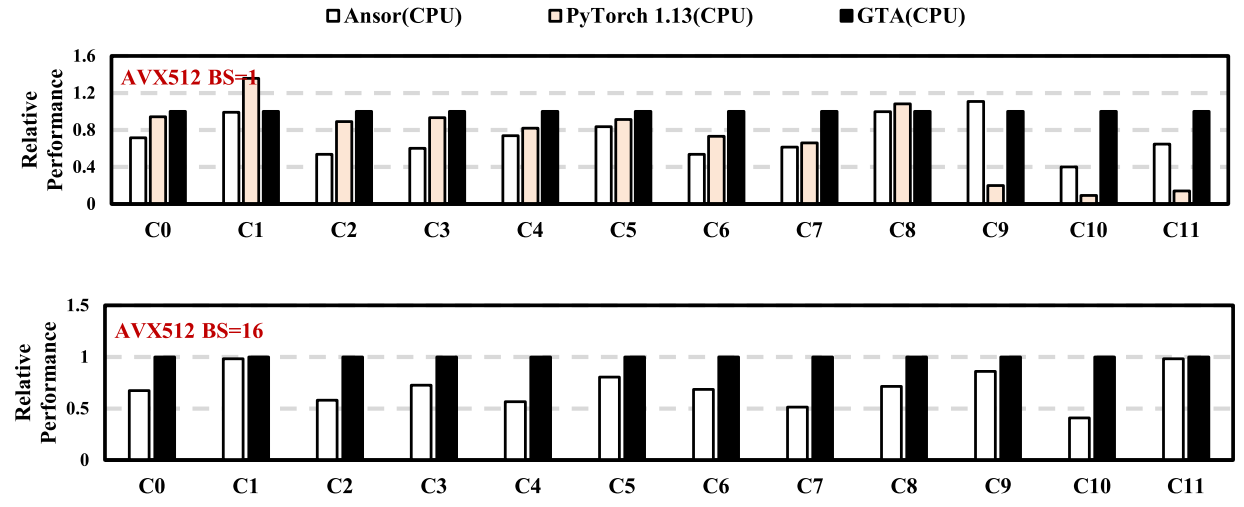

Fig. 7. Performance on AMD Ryzen 7 7840H CPU relative to Ansor and PyTorch.

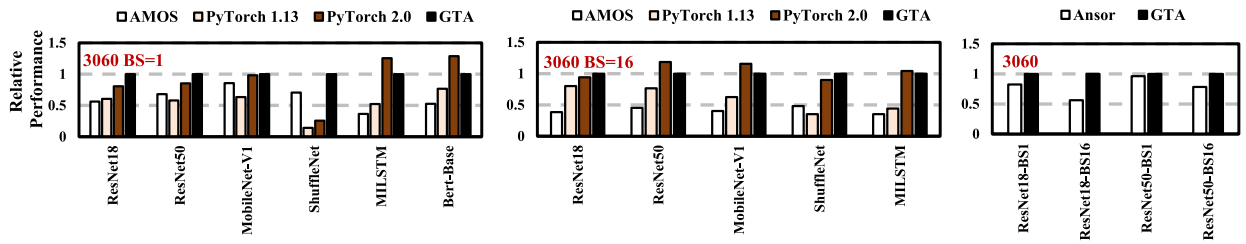

Fig. 8. Performance of different networks relative to GTA on Tensor Core.

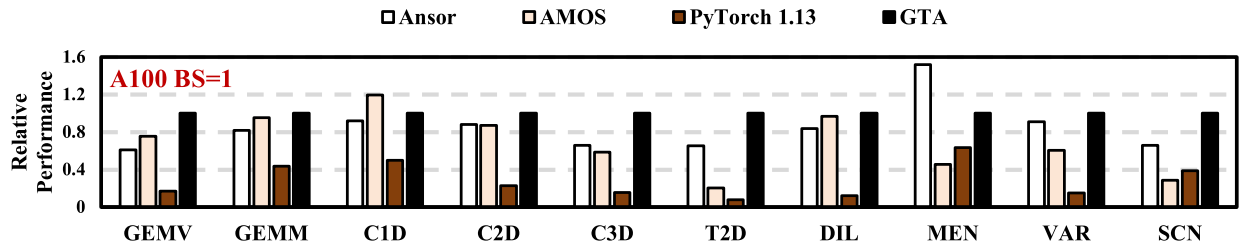

Fig. 9. Performance comparison of GTA across multiple individual operators on the NVIDIA A100 GPU, compared with baseline methods.

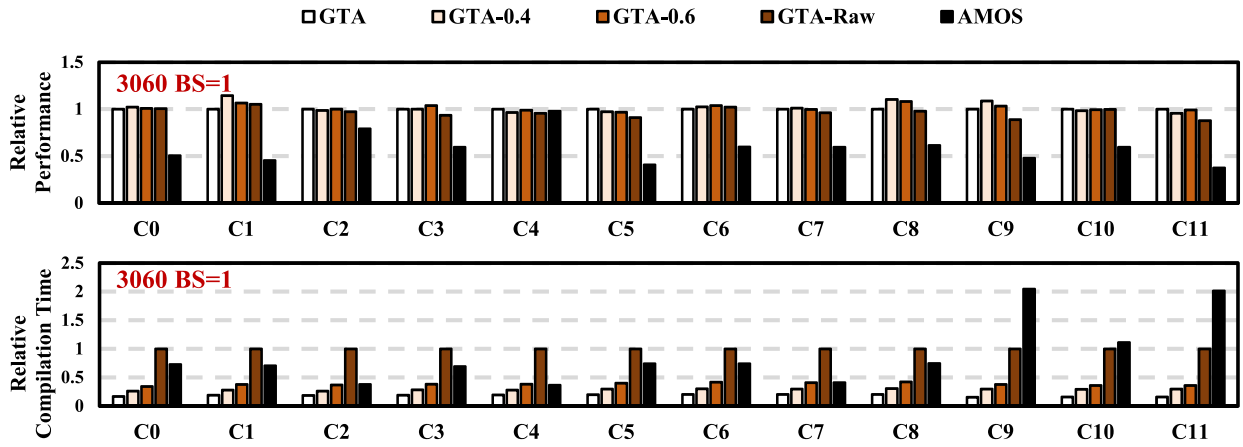

Fig. 10. Compilation time overhead and corresponding performance variations under different sampling rates.

7.6. Network performance

Fig. 8 illustrates the performance of GTA on six evaluated networks. On average, GTA achieves 1.75×, 1.42× and 1.29× speedups over AMOS, PyTorch 1.13 and PyTorch 2.0 with TorchInductor, respectively. For ResNet18 and ResNet50, GTA finds better mappings for operators, enabling more extensive utilization of Tensor Cores compared to hand-tuned libraries and AMOS's optimized templates. GTA overcomes the limitations of these baselines by generating accurate search spaces that encompass most high-performance programs, along with an efficient search algorithm for finding optimal or near-optimal solutions. The results demonstrate GTA's capability to handle complex operators and effectively leverage Tensor Cores for high performance.

7.7. Compilation time

The search time overhead is a critical factor for practical deployment in deep learning frameworks, as reducing it can significantly enhance usability. To evaluate the efficiency of our dual-task scheduling strategy, we analyze the search time and corresponding performance variations under different sampling rates, specifically comparing GTA at sampling rates of 40% (GTA-0.4), 60% (GTA-0.6), and 100% (GTA-Raw). In this experiment, GTA operates at a sampling rate of 20% (GTA-0.2), representing a highly efficient configuration with minimal search overhead. The results, as shown in Fig. 10, demonstrate that as the sampling rate decreases, the search time is significantly reduced while maintaining less than a 5% performance degradation on average, thereby achieving an excellent balance between search efficiency and performance.

Additionally, we compare GTA's search time overhead and performance with AMOS, a state-of-the-art compiler designed for DLAs. Our findings reveal that GTA achieves an average performance improvement of 1.88× over AMOS while maintaining significantly lower search time. Specifically, AMOS's average compilation time is approximately five times that of GTA. This substantial reduction in search time underscores the effectiveness of GTA's dual-task scheduling strategy, which optimizes resource allocation during the search process and enables the rapid identification of high-performance tensor programs.

Unlike traditional methods that exhaustively explore all mapping candidates, GTA employs a dynamic prioritization strategy that adaptively allocates tuning resources based on performance feedback. This strategy ensures that the most promising subgraphs and intrinsic mapping candidates are prioritized, while less promising candidates receive fewer tuning opportunities. By combining this with a sampling-based approach, GTA minimizes unnecessary exploration while maintaining high-quality tensor programs. These results underscore GTA's suitability for real-world deployment scenarios, where both rapid code generation and performance optimization are critical. Furthermore, the ability to adjust sampling rates offers flexibility in balancing search time and performance, making GTA a robust solution for optimizing tensor programs across diverse workloads.

8. Related work

In addition to reviewing DLAs, we summarize related work on numeric precision and dynamic shape optimization for deep learning.

Deep learning accelerators. DLAs offer several significant advantages, making them essential for advancing DNN research and deployment. First, DLAs feature large memory capacities, which accommodate the rapidly growing number of parameters in modern models and facilitate efficient training processes. Second, they provide model-specific optimizations while maintaining a degree of flexibility, enabling tailored performance improvements for various architectures. Additionally, DLAs support a broader range of data formats, such as FP16, BF16, and INT8, which enhance computational efficiency and reduce memory usage. Third, DLAs are equipped with a high number of computing units, enabling extensive parallelism to handle the computational demands of DNNs effectively. These characteristics position DLAs as a cornerstone technology for accelerating the training and inference of deep learning models. Following this trend, many emerging accelerators have been proposed, targeting specific algorithms or utilizing new technologies. In academia, the DianNao

family [68–71] significantly improves DL computation performance by leveraging specialized functional units, memory hierarchy, and interconnects. Meanwhile, the expansion of DL applications in industry has led hardware vendors (e.g., NVIDIA Tensor Core [17–19] and Intel NNP [72]), internet giants (e.g., Tesla Dojo [73], Huawei Ascend [74], Google TPU [10] and Apple M4 [75,76]), and startups (e.g., Cambricon MLU [77] and Graphcore IPU [78]) to develop various DLAs. Both academic and industry DLAs are fundamentally domain-specific, rather than general-purpose accelerators, inevitably leading to complex and diverse architectural constraints.

Numeric precision optimization. Quantization [79,80], a pivotal technique in deep learning, reduces the numeric precision of weights and activations to enhance computational efficiency and lower resource requirements. By transitioning from high-precision formats such as FP32 to lower-precision formats like FP16, INT8, or even single-bit representations [81,82], quantization enables significant reductions in memory usage and power consumption [83,84]. The progression of hardware architectures aligns with the increasing demands for low-precision computations. For instance, NVIDIA's recent developments, such as the Turing and Ampere architectures, incorporated INT8 and INT4 tensor cores to enhance efficiency. Meanwhile, the latest Hopper architecture has shifted focus by replacing INT4 support with FP8 tensor cores, prioritizing improved numerical precision. These advancements allow large-scale models, including Large Language Models (LLMs) [85], to be deployed on resource-constrained devices like edge devices and DLAs without sacrificing performance. Compilers play a critical role in making quantization effective. Tools like AMOS [22], PreTuner [86] and LADDER [39] introduce advanced optimizations for low-precision data types, including hardware-aware scheduling, loop tiling, and fine-grained scaling strategies. Expanding on existing techniques, an automated approach [87] integrates bit-slicing into the scheduling phase, treating quantization as part of the schedule space. Coupled with program synthesis, this method efficiently generates hardware-specific kernels, supporting diverse quantization configurations and ensuring seamless adaptation to new hardware architectures.

Dynamic shape optimization. Dynamic-shape workloads are characteristic of DNN models where tensor shapes vary at runtime based on input data, such as the sequence length in Transformer models. These workloads pose substantial challenges for existing autotuning frameworks like TVM, which primarily rely on static input shapes to construct search spaces and cost models. For instance, TVM's second-generation IR, Relay [35], lacks the capability to represent dynamic tensors. While its third-generation IR, Relax [88], introduces symbolic shapes to support dynamic workloads, Relax still depends on hand-written templates for tensor program generation and lacks automatic tuning support. To address these limitations, recent works such as Nimble [89], DietCode [90], FTuner [91], and MIKPOLY [92] have introduced innovative techniques. These approaches construct shape-agnostic search spaces and cost models to optimize dynamic-shape workloads. For example, DietCode effectively groups kernels with varying shapes into unified workloads, enabling efficient tuning as a single entity and significantly reducing overall tuning time. FTuner introduces a uKernel-based approach for dynamic tensors, leveraging hardware-aware constraints to generate high-performance kernel programs and combining uKernels during runtime to optimize padding and execution efficiency. While these advancements mark significant progress, further research is needed to fully exploit the potential of dynamic-shape DNNs on modern hardware accelerators.

9. Conclusion

We propose GTA, a novel compilation framework for high-performance tensor program generation on DLAs. GTA expands the search space by coordinating intrinsic-based automatic mapping abstraction with rule-based tensor program generation strategy and applies pruning rules to eliminate ineffective program candidates. Additionally, GTA employs dual-task scheduling strategy for tensorized programs, effectively reducing tuning efforts while enhancing performance. Experimental results on three DLAs show that GTA outperforms state-of-the-art automatic generation approaches and vendor-provided hand-tuned libraries by 1.88× and 2.29×, respectively.

CRediT authorship contribution statement

Anxing Xie: Writing – original draft, Software, Resources, Project administration, Methodology, Investigation, Data curation. **Yonghua Hu:** Writing – review & editing, Supervision, Investigation, Funding acquisition. **Yaohua Wang:** Writing – review & editing, Supervision, Methodology, Investigation, Funding acquisition, Formal analysis. **Zhe Li:** Writing – review & editing, Supervision, Investigation, Formal analysis. **Yuxiang Gao:** Investigation. **Zenghua Cheng:** Investigation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable suggestions. This work is supported by the National Key R&D Program of China (No. 2022ZD0119003), Hunan Provincial Natural Science Foundation (No. 2023JJ50019), the Postgraduate Scientific Research Innovation Project of Hunan Province (No. CX20231019) and the National Natural Science Foundation of China (No. 62272477).

Data availability

Data will be made available on request.

References

[1] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[2] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al., Evolving deep neural networks, in: Artificial Intelligence in the Age of Neural Networks and Brain Computing, Elsevier, 2024, pp. 269–287.
[3] C.-Y. Wang, I.-H. Yeh, H.-Y. Mark Liao, Yolov9: Learning what you want to learn using programmable gradient information, in: European Conference on Computer Vision, Springer, 2025, pp. 1–21.
[4] A. Vaswani, Attention is all you need, Adv. Neural Inf. Process. Syst. (2017).
[5] P.P. Ray, ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope, Internet Things Cyber- Phys. Syst. 3 (2023) 121–154.
[6] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al., The llama 3 herd of models, 2024, arXiv preprint arXiv:2407.21783.
[7] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, B. Schiele, The cityscapes dataset for semantic urban scene understanding, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 3213–3223.
[8] D. Fu, X. Li, L. Wen, M. Dou, P. Cai, B. Shi, Y. Qiao, Drive like a human: Rethinking autonomous driving with large language models, in: Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision, 2024, pp. 910–919.
[9] C. Cui, Y. Ma, X. Cao, W. Ye, Y. Zhou, K. Liang, J. Chen, J. Lu, Z. Yang, K.-D. Liao, et al., A survey on multimodal large language models for autonomous driving, in: Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision, 2024, pp. 958–979.

[10] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., In-datacenter performance analysis of a tensor processing unit, in: Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017, pp. 1–12.

[11] X. Zhang, X. Zhou, M. Lin, J. Sun, Shufflenet: An extremely efficient convolutional neural network for mobile devices, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 6848–6856.

[12] C.-Y. Wang, A. Bochkovskiy, H.-Y.M. Liao, YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 7464–7475.

[13] Z. Xu, W. Wang, H. Dai, Y. Xu, XFC: Enabling automatic and fast operator synthesis for mobile deep learning compilation, J. Syst. Archit. 142 (2023) 102921.

[14] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, D. Chen, FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge, in: Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[15] W. Jiang, L. Yang, E.H.-M. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, J. Hu, Hardware/software co-exploration of neural architectures, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (12) (2020) 4805–4815.

[16] Z. Xie, M. Emani, X. Yu, D. Tao, X. He, P. Su, K. Zhou, V. Vishwanath, Centimani: Enabling fast {AI} accelerator selection for {dNN} training with a novel performance predictor, in: 2024 USENIX Annual Technical Conference, USENIX ATC 24, 2024, pp. 1203–1221.

[17] Nvidia, Ampere architecture white paper, 2022, URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf Online (Accessed 13 November 2024).

[18] Nvidia, Turing architecture white paper, 2022, URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf Online (Accessed 13 November 2024).

[19] Nvidia, Volta architecture white paper, 2022, URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf Online (Accessed 13 November 2024).

[20] K. Troester, R. Bhargava, AMD next generation "Zen 4" core and 4th Gen AMD EPYC™ 9004 server CPU, in: 2023 IEEE Hot Chips 35 Symposium, HCS, IEEE Computer Society, 2023, pp. 1–25.

[21] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al., {TVM}: An automated {end-to-end} optimizing compiler for deep learning, in: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 18, 2018, pp. 578–594.

[22] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, Y. Liang, AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction, in: Proceedings of the 49th Annual International Symposium on Computer Architecture, 2022, pp. 874–887.

[23] S. Feng, B. Hou, H. Jin, W. Lin, J. Shao, R. Lai, Z. Ye, L. Zheng, C.H. Yu, Y. Yu, et al., Tensorir: An abstraction for automatic tensorized program optimization, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2023, pp. 804–817.

[24] J. Bi, Q. Guo, X. Li, Y. Zhao, Y. Wen, Y. Guo, E. Zhou, X. Hu, Z. Du, L. Li, et al., Heron: Automatically constrained high-performance library generation for deep learning accelerators, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2023, pp. 314–328.

[25] L. Zheng, C. Jia, M. Sun, Z. Wu, C.H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al., Ansor: Generating {high-performance} tensor programs for deep learning, in: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 20, 2020, pp. 863–879.

[26] S. Zheng, Y. Liang, S. Wang, R. Chen, K. Sheng, Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 859–873.

[27] A. Sabne, Xla: Compiling machine learning for peak performance, Google Res (2020).

[28] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W.S. Moses, S. Verdoolaege, A. Adams, A. Cohen, Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018, arXiv preprint arXiv:1802.04730.

[29] P. Tillet, H.-T. Kung, D. Cox, 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, 2019, pp. 10–19.

[30] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, O. Zinenko, MLIR: A compiler infrastructure for the end of Moore's law, 2020, arXiv preprint arXiv:2002.11054.

[31] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, L. Zhou, Rammer: Enabling holistic deep learning compiler optimizations with {rtasks}, in: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 20, 2020, pp. 881–897.

[32] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, et al., AKG: automatic kernel generation for neural processing units using polyhedral transformations, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 1233–1248.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, Adv. Neural Inf. Process. Syst. 32 (2019).

[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., {TensorFlow}: a system for {large-scale} machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 16, 2016, pp. 265–283.

[35] J. Roesch, S. Lyubomirsky, M. Kirisame, L. Weber, J. Pollock, L. Vega, Z. Jiang, T. Chen, T. Moreau, Z. Tatlock, Relay: A high-level compiler for deep learning, 2019, arXiv preprint arXiv:1904.08368.

[36] J. Zhao, X. Gao, R. Xia, Z. Zhang, D. Chen, L. Chen, R. Zhang, Z. Geng, B. Cheng, X. Jin, Apollo: Automatic partition-based operator fusion through layer by layer optimization., in: MLSys, 2022.

[37] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, L. Zhou, Welder: Scheduling deep learning memory access via tile-graph, in: 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 23, 2023, pp. 701–718.

[38] C. Xia, J. Zhao, Q. Sun, Z. Wang, Y. Wen, T. Yu, X. Feng, H. Cui, Optimizing deep learning inference via global analysis and tensor expressions, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, 2024, pp. 286–301.

[39] L. Wang, L. Ma, S. Cao, Q. Zhang, J. Xue, Y. Shi, N. Zheng, Z. Miao, F. Yang, T. Cao, et al., Ladder: Enabling efficient {low-precision} deep learning computing through hardware-aware tensor transformation, in: 18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 24, 2024, pp. 307–323.

[40] F. Wang, M. Shen, Y. Lu, N. Xiao, TensorMap: A deep RL-based tensor mapping framework for spatial accelerators, IEEE Trans. Comput. (2024).

[41] Y. Zhao, H. Sharif, V. Adve, S. Misailovic, Felix: Optimizing tensor programs with gradient descent, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2024, pp. 367–381.

[42] Q. Zhao, R. Wang, Y. Liu, H. Yang, Z. Luan, D. Qian, Sifter: An efficient operator auto-tuner with speculative design space exploration for deep learning compiler, IEEE Trans. Comput. (2024).

[43] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, Acm Sigplan Not. 48 (6) (2013) 519–530.

[44] Y. Bai, X. Yao, Q. Sun, W. Zhao, S. Chen, Z. Wang, B. Yu, Gtco: Graph and tensor co-design for transformer-based image recognition on tensor cores, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2023).

[45] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, A. Parashar, Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings, IEEE Micro 40 (3) (2020) 20–29.

[46] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, Y. Liang, Tenet: A framework for modeling tensor dataflow based on relation-centric notation, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, ISCA, IEEE, 2021, pp. 720–733.

[47] A. Parashar, P. Raina, Y.S. Shao, Y.-H. Chen, V.A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S.W. Keckler, J. Emer, Timeloop: A systematic approach to dnn accelerator evaluation, in: 2019 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2019, pp. 304–315.

[48] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, et al., Interstellar: Using halide's scheduling language to analyze dnn accelerators, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 369–383.

[49] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, A. Krishnamurthy, Learning to optimize tensor programs, Adv. Neural Inf. Process. Syst. 31 (2018).

[50] J. Appleyard, S. Yokim, NVIDIA developer technical blog, 2017, URL:https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9 Online (Accessed 13 November 2024).

[51] NVIDIA, Basic linear algebra on NVIDIA GPUs, 2024, URL: https://developer.nvidia.com/cublas Online (Accessed 13 November 2024) n.d.

[52] A. Kerr, H. Wu, M. Gupta, D. Blasig, P. Ramini, D. Merrill, A. Shivam, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, M. Nicely, CUTLASS, 2022, URL: https://github.com/NVIDIA/cutlass Online (Accessed 13 November 2024).
[53] T. Zerrell, J. Bruestle, Stripe: Tensor compilation via the nested polyhedral model, 2019, arXiv preprint arXiv:1903.06498.
[54] R. Baghdadi, J. Ray, M.B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, S. Amarasinghe, Tiramisu: A polyhedral compiler for expressing fast and portable code, in: 2019 IEEE/ACM International Symposium on Code Generation and Optimization, CGO, IEEE, 2019, pp. 193–205.
[55] S. Tavarageri, A. Heinecke, S. Avancha, B. Kaul, G. Goyal, R. Upadrasta, Polydl: Polyhedral optimizations for creation of high-performance dl primitives, ACM Trans. Archit. Code Optim. (TACO) 18 (1) (2021) 1–27.
[56] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, Y.S. Shao, Cosa: Scheduling by constrained optimization for spatial accelerators, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, ISCA, IEEE, 2021, pp. 554–566.
[57] M. Sotoudeh, A. Venkat, M. Anderson, E. Georganas, A. Heinecke, J. Knight, ISA mapper: a compute and hardware agnostic deep learning compiler, in: Proceedings of the 16th ACM International Conference on Computing Frontiers, 2019, pp. 164–173.
[58] J. Weng, A. Jain, J. Wang, L. Wang, Y. Wang, T. Nowatzki, UNIT: Unifying tensorized instruction compilation, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization, CGO, IEEE, 2021, pp. 77–89.
[59] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui, et al., {RollER}: Fast and efficient tensor compilation for deep learning, in: 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 22, 2022, pp. 233–248.
[60] Y. Ding, C.H. Yu, B. Zheng, Y. Liu, Y. Wang, G. Pekhimenko, Hidet: Task-mapping programming paradigm for deep learning tensor programs, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2023, pp. 370–384.
[61] L. Zheng, H. Wang, J. Zhai, M. Hu, Z. Ma, T. Wang, S. Huang, X. Miao, S. Tang, K. Huang, et al., {EINNET}: Optimizing tensor programs with {derivation-based} transformations, in: 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 23, 2023, pp. 739–755.
[62] Y. Zhai, S. Yang, K. Pan, R. Zhang, S. Liu, C. Liu, Z. Ye, J. Ji, J. Zhao, Y. Zhang, et al., Enabling tensor language model to assist in generating {High-Performance} tensor programs for deep learning, in: 18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 24, 2024, pp. 289–305.
[63] F. Wang, M. Shen, Y. Ding, N. Xiao, Soter: Analytical tensor-architecture modeling and automatic tensor program tuning for spatial accelerators, in: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture, ISCA, IEEE, 2024, pp. 991–1004.
[64] F. Wang, M. Shen, Automatic kernel generation for large language models on deep learning accelerators, in: 2023 IEEE/ACM International Conference on Computer Aided Design, ICCAD, IEEE, 2023, pp. 1–9.
[65] J. Devlin, Bert: Pre-training of deep bidirectional transformers for language understanding, 2018, arXiv preprint arXiv:1810.04805.
[66] Y. Wu, S. Zhang, Y. Zhang, Y. Bengio, R.R. Salakhutdinov, On multiplicative integration with recurrent neural networks, Adv. Neural Inf. Process. Syst. 29 (2016).
[67] A.G. Howard, Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017, arXiv preprint arXiv:1704.04861.
[68] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning, ACM SIGARCH Comput. Archit. News 42 (1) (2014) 269–284.
[69] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al., Dadiannao: A machine-learning supercomputer, in: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE, 2014, pp. 609–622.
[70] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, Y. Chen, Pudiannao: A polyvalent machine learning accelerator, ACM SIGARCH Comput. Archit. News 43 (1) (2015) 369–381.
[71] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, O. Temam, ShiDianNao: Shifting vision processing closer to the sensor, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015, pp. 92–104.
[72] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, S. Avancha, Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product, in: 2020 IEEE 27th Symposium on Computer Arithmetic, ARITH, IEEE, 2020, pp. 133–136.
[73] E. Talpes, D. Williams, D.D. Sarma, Dojo: The microarchitecture of tesla's exascale computer, in: 2022 IEEE Hot Chips 34 Symposium, HCS, IEEE Computer Society, 2022, pp. 1–28.
[74] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, Y. Hu, Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper, in: 2021 IEEE International Symposium on High-Performance Computer Architecture, HPCA, IEEE, 2021, pp. 789–801.
[75] Apple, Apple introduces M4 chip, 2024, URL: https://www.apple.com/sg/newsroom/2024/05/apple-introduces-m4-chip/ Online (Accessed 13 November 2024).
[76] Apple, Apple introduces M4 pro and M4 max, 2024, URL: https://www.apple.com/sg/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/ Online (Accessed 13 November 2024).
[77] Cambricon, Cambricon MLU, 2024, URL: https://www.cambricon.com/ Online (Accessed 13 November 2024) n.d..
[78] Z. Jia, B. Tillman, M. Maggioni, D.P. Scarpazza, Dissecting the graphcore ipu architecture via microbenchmarking, 2019, arXiv preprint arXiv:1912.03413.
[79] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Quantized neural networks: Training neural networks with low precision weights and activations, J. Mach. Learn. Res. 18 (187) (2018) 1–30.
[80] T. Liang, J. Glossner, L. Wang, S. Shi, X. Zhang, Pruning and quantization for deep neural network acceleration: A survey, Neurocomput. 461 (2021) 370–403.
[81] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1, 2016, arXiv preprint arXiv:1602.02830.
[82] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, Xnor-net: Imagenet classification using binary convolutional neural networks, in: European Conference on Computer Vision, Springer, 2016, pp. 525–542.
[83] C.-C. Yang, Y.-R. Chen, H.-H. Liao, Y.-M. Chang, J.-K. Lee, Auto-tuning fixed-point precision with TVM on RISC-v packed SIMD extension, ACM Trans. Des. Autom. Electron. Syst. 28 (3) (2023) 1–21.
[84] D. Diamantopoulos, B. Ringlein, M. Purandare, G. Singh, C. Hagleitner, Agile autotuning of a transprecision tensor accelerator overlay for TVM compiler stack, in: 2020 30th International Conference on Field-Programmable Logic and Applications, FPL, IEEE, 2020, pp. 310–316.
[85] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, Z. Jia, Towards efficient generative large language model serving: A survey from algorithms to systems, 2023, arXiv preprint ArXiv:2312.15234.
[86] J. Xu, G. Song, B. Zhou, F. Li, J. Hao, J. Zhao, A holistic approach to automatic mixed-precision code generation and tuning for affine programs, in: Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2024, pp. 55–67.
[87] M. Cowan, T. Moreau, T. Chen, J. Bornholt, L. Ceze, Automatic generation of high-performance quantized machine learning kernels, in: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, 2020, pp. 305–316.
[88] R. Lai, J. Shao, S. Feng, S.S. Lyubomirsky, B. Hou, W. Lin, Z. Ye, H. Jin, Y. Jin, J. Liu, et al., Relax: Composable abstractions for end-to-end dynamic machine learning, 2023, arXiv preprint arXiv:2311.02103.
[89] H. Shen, J. Roesch, Z. Chen, W. Chen, Y. Wu, M. Li, V. Sharma, Z. Tatlock, Y. Wang, Nimble: Efficiently compiling dynamic neural networks for model inference, Proc. Mach. Learn. Syst. 3 (2021) 208–222.
[90] B. Zheng, Z. Jiang, C.H. Yu, H. Shen, J. Fromm, Y. Liu, Y. Wang, L. Ceze, T. Chen, G. Pekhimenko, DietCode: Automatic optimization for dynamic tensor programs, Proc. Mach. Learn. Syst. 4 (2022) 848–863.
[91] P. Mu, L. Wei, Y. Liu, R. Wang, FTuner: A fast dynamic shape tensors program auto-tuner for deep learning compilers, 2024, arXiv preprint arXiv:2407.21418.
[92] F. Yu, G. Li, J. Zhao, H. Cui, X. Feng, J. Xue, Optimizing dynamic-shape neural networks on accelerators via on-the-fly micro-kernel polymerization, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2024, pp. 797–812.

Anxing Xie is currently working toward a Ph.D. degree in the School of Computer Science and Engineering, Hunan University of Science and Technology, China. He is currently working on deep learning automatic compilation optimization and high-performance computation. His research interests include compiler optimization, and parallel computing.

Yonghua Hu is a professor in School of Computer Science and Engineering, Hunan University of Science and Technology, China. He received the Ph.D degree in Computer Application Technology from Hunan University, in 2008. He went to University at Buffalo SUNY as a visiting scholar in 2019. His research interests include compilation optimization, artificial intelligence and parallel computing.

Yaohua Wang is currently a professor with the College of Computer Science, National University of Defense Technology. His research interest is in computer architecture, machine learning and security. His work spans and stretches the boundaries of computer architecture. He is especially excited about novel, fundamentally-efficient computation, and memory/storage paradigms, applied to emerging machine learning applications.

Zhe Li received the Ph.D. degree in Computer Science from Jilin University in 2022. He is currently working at Tianjin Advanced Technology Institute. His research interests include deep learning compilation and combinatorial optimization.

Yuxiang Gao is currently working toward an M.S. degree in the School of Computer Science and Engineering, Hunan University of Science and Technology, China. He is currently working on code optimization and compilation technology. His research interests include automatic compilation optimization and code generation.

Zenghua Cheng is currently working toward an M.S. degree in the School of Computer Science and Engineering, Hunan University of Science and Technology, China. He is currently working on code optimization and compilation technology. His research interests include automatic compilation optimization and Web security.