ConCo: Optimizing Compilation of Concurrent Tensor Programs on Shared GPU

Jiamin Lu
University of Science and Technology of China
Hefei, China
lujiamin@mail.ustc.edu.cn

Jingwei Sun*
University of Science and Technology of China
Hefei, China
sunjw@ustc.edu.cn

Yunlong Xu
Independent Researcher
Beijing, China
xjtu.ylxu@gmail.com

Peng Sun
Independent Researcher
Beijing, China
peng.sunshine@gmail.com

Guangzhong Sun
University of Science and Technology of China
Hefei, China
gzsun@ustc.edu.cn

Abstract

Serving multiple inference tasks of deep neural networks (DNNs) concurrently on a shared GPU is an established method for maximizing hardware resource. Although DNN compilers effectively generate optimal kernel code for individual DNN inferences, they fall short in optimizing for concurrent tasks. This paper presents ConCo, a concurrency-aware compilation scheme designed to optimize the execution of concurrent DNN inference tasks on a shared GPU. ConCo dynamically generates multiple code variants, each tailored to different GPU resource constraints, and efficiently selects optimal variants at runtime according to concurrent workload characteristics. To mitigate the substantial overhead associated with multi-variant compilation, ConCo employs an optimal-code-sharing strategy, significantly accelerating compilation by leveraging commonalities across resource configurations. Evaluations demonstrate that ConCo improves inference throughput by up to 1.2× and reduces job completion time by up to 69.85% compared to existing solutions.

*Corresponding author

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730433

CCS Concepts

• **Computing methodologies** → **Concurrent computing methodologies**; **Neural networks**.

Keywords

Deep Neural Network, DNN Compiler, GPU Sharing

ACM Reference Format:
Jiamin Lu, Jingwei Sun, Yunlong Xu, Peng Sun, and Guangzhong Sun. 2025. ConCo: Optimizing Compilation of Concurrent Tensor Programs on Shared GPU. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3730433

1 Introduction

Deep Neural Networks (DNNs) have found extensive applications in domains such as computer vision, autonomous driving systems, and recommendation systems. To meet the demands for real-time and efficient inference, Graphics Processing Units (GPUs) have emerged as powerful parallel processing hardware extensively utilized to accelerate the DNN inference process. The shared utilization of GPU resources has further propelled the efficient use of resources and the pursuit of computational performance. By serving multiple DNN inference tasks simultaneously, shared GPUs not only enhance model utilization and improve the overall throughput of inference tasks but also reduce costs [20, 46]. Consequently, research and optimization of DNN inference on shared GPU resources have become crucial topics in the current landscape of DNN applications.

With the evolution of deep learning, numerous complex DNN models and diverse architectures of deep learning accelerators have emerged. Traditional deep learning frameworks typically employ DNN libraries [5, 6, 15, 31] to execute DNN

operators. Manually crafting optimal code implementation for each model and hardware type is an exceedingly laborious task. In response, developers try to seek a universal method to accommodate this diversity while fully leveraging the performance potential of the hardware. To address this challenge, various DNN compilers have been developed [3, 11, 40, 41]. These compilers accept models from deep learning frameworks and automatically generate low-level code targeting specific hardware.

Existing DNN compilers implicitly assume that the model can utilize the entire hardware resources, including computational resources, cache capacity, and memory bandwidth. However, this assumption fails in concurrent inference scenarios, where multiple tasks share GPU resources, leading to performance degradation. Each inference task only partially occupies GPU resources, making the original optimizations suboptimal due to increased contention for computation units and memory bandwidth. One possible approach to address varying resource availability is to prepare multiple code variants tailored to different GPU conditions. However, this dramatically increases compilation time, making it impractical for real-world deployment. Furthermore, even if multiple code variants have been generated, existing GPU resource allocation and scheduling strategies lack the ability to dynamically select proper variants in response to workload fluctuations, resulting in persistent inefficiencies when concurrency levels change. These challenges highlight the need for a new compilation approach that ensures high performance, manageable compilation overhead, and adaptability to dynamic execution environments.

In this paper, to address these challenges, we propose ConCo, a **Con**currency-aware **Co**mpilation scheme for optimizing concurrent DNN inference tasks on a shared GPU. ConCo consists of three key components: (a) **Concurrency-Aware Code Generation**: Instead of assuming full GPU utilization, ConCo pre-generates multiple optimized code variants tailored to different GPU resource demands, ensuring efficient execution under varying resource allocation conditions. (b) **Efficient Code Generation Strategy**: To mitigate the excessive compilation time of traditional DNN compilers, ConCo introduces an optimized code generation strategy, reducing compilation overhead by over 2×. (c) **Code Selection for Different Concurrency Scenarios**: Unlike existing schedulers [10, 18, 24, 29, 45] that rely on static code optimizations, ConCo pre-generates and selects optimized code offline for different concurrency scenarios, ensuring that the most suitable implementation is used for the corresponding workload condition.

In summary, this paper makes the following contributions:

- We proposed a compilation optimization scheme for shared GPU scenarios, introducing a selection strategy that identifies the optimal combination of candidate codes and determines the most efficient resource allocation plan.
- We proposed a compilation method based on optimal code sharing in concurrent scenarios with fallback codes. Compared to Ansor [48] compilation process, our approach can achieve more than a 2× increase in compilation speed.
- We evaluated ConCo on NVIDIA A100 and 2080 Ti GPUs using a diverse set of workloads. Compared to existing solutions that are not aware of the characteristics of concurrent inference scenarios, ConCo achieved a throughput improvement of up to 1.2× and a reduction in JCT of up to 69.85%.

2 Background & Motivation

2.1 GPU Concurrency Mechanisms

With the growing interest in concurrent inference using shared GPUs, GPU vendors, particularly NVIDIA, have proposed several methods for sharing GPU resources at the spatial level (Figure 1). Since the Fermi architecture, NVIDIA introduced a software programming model known as multi-stream [32], which allows developers to define multiple CUDA Streams within a single process. Kernels from different streams within the same process can be executed concurrently on the GPU, sharing its resources. Following the introduction of multi-streaming, NVIDIA further introduced Multi-Process Service (MPS) [33] to support flexible resource allocation, isolation, and virtualization. This approach allows kernels from different processes to share GPU resources. Although MPS increased the flexibility of resource sharing, it did not address issues such as memory capacity and bandwidth contention, nor did it provide error isolation mechanisms. To address these challenges, starting with the Ampere architecture, NVIDIA introduced a new feature called Multi-Instance GPU (MIG) [34]. MIG achieves complete isolation of computational units, memory, and errors by creating hardware isolation units called GPU Instances (GIs). The development of these GPU resource-sharing mechanisms provides technical support for concurrent DNN inference on shared GPUs.

2.2 DNN Compiler

DNN compilers have been proposed as a generalized approach to accommodate diverse DNN models and harness the full potential of the hardware performance. The objective of a DNN compiler is to enhance the execution efficiency of DNN models across various acceleration hardware, thereby maximizing hardware performance.

Taking Ansor [48] as a representative example, Ansor begins by partitioning the computation graph into subgraphs

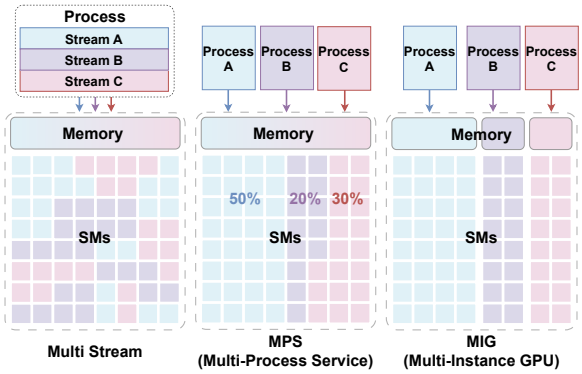


Figure 1: GPU Concurrency Mechanisms

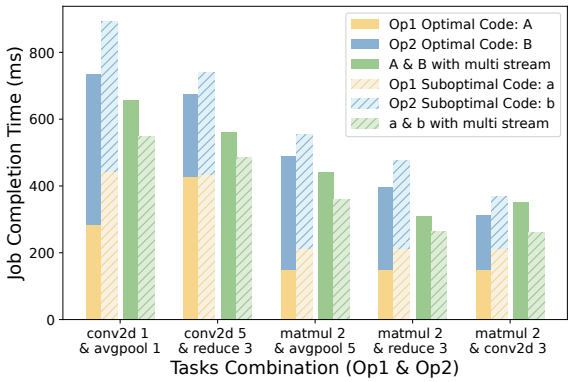


Figure 2: Completion Times of Exclusive and Concurrent Job Executions for Different Codes (The yellow and blue bars represent the completion times for 1,000 exclusive GPU runs of operators Op1 and Op2, respectively. The green bars denote the completion times for 1,000 concurrent executions of Op1 and Op2 using the multi-stream method.)

and optimizes them iteratively. In each tuning round, a set of random code variants is generated via sketch-based sampling and random annotations. These candidates, together with top-performing implementations from previous rounds, form the initial population for evolutionary search. The generated offspring are evaluated by an XGBoost-based cost model [2], and only the most promising candidates are selected for actual execution on hardware. The measured latencies then serve as feedback to refine the cost model and guide future search iterations.

2.3 Motivation

Existing compilers have consistently prioritized the generation of operators that perform optimally in exclusive scenarios. However, we have observed that certain operators, which are typically discarded by Ansor's traditional search process due to overlong exclusive latency, can exhibit unexpected performance advantages in concurrent environments.

We construct operator pairs (Op1 and Op2) comprising fundamental deep learning operation types, such as matrix multiplication (matmul), 2D convolution (conv2d), reduction (reduce), average pooling (avgpool). These operators are adopted from the benchmark collection in ROLLER [49], which provides 119 representative operators extracted from models such as ResNet-50 [12], LSTM [28], NASNet [50], and BERT [7] for compiler evaluation. In our study, we employed Ansor to measure operators Op1 and Op2, each conducted 1,000 times, thereby acquiring the exclusive optimal codes A and B, respectively. To attain codes without an exclusive optimal bias, we eliminated the predictive capabilities of the Ansor's learned Cost Model, which is based on XGBoost [2], resulting in a completely random generation of operators. Among the 8,000 codes generated through this random search process, we identified codes 'a' and 'b' for operators Op1 and Op2 that, despite underperforming compared to codes A and B when run exclusively on the GPU, achieved shorter concurrent completion times under concurrent execution conditions.

Operator	Configuration
matmul 2	M=128, K=4032, N=1000
conv2d 1	D=(16,128,58,58), K=(128,128,3,3), S=2
conv2d 3	D=(16,168,42,42), K=(168,168,1,1), S=1
conv2d 5	D=(16,512,16,16), K=(512,512,3,3), S=2
reduce 3	I=(128,4032,11,11), axis=[2,3]
avgpool 1	D=(128,168,83,83), K=1, S=2, VALID
avgpool 5	D=(128,336,42,42), K=3, S=2, SAME

Table 1: The configuration of operators used in Figure 2. (In the Roller benchmark, the conv2d operator was configured with a batch size of 128. However, given its substantially longer execution time compared to other operators (by an order of magnitude), we reduced the batch size to 16.)

Figure 2 shows the top 5 pairs with the most significant co-location speedups in our experiments, whose operator configurations are presented in Table 1. Our experiments demonstrated that meticulously selected suboptimal exclusive codes could reduce the concurrent completion time by as much as 26.2%. This finding has sparked our interest in investigating compilation optimizations that are concurrency-aware within shared GPU contexts.

2.4 Compilation Optimization for Concurrent DNN Inference

In the context of concurrent DNN inference tasks on shared CPUs, prior studies [26] proposed a compilation optimization and scheduling scheme tailored for multi-tenant environments. The core idea for achieving efficient concurrent inference in shared CPU environments is as follows:

(1) **Single-Pass Multi-Version Compilation**: By generating multiple code versions in a single search pass of Ansor, each version is optimized for different interference levels. These versions are selected from the Pareto frontier of the **parallelism-locality** trade-off space to adapt to varying interference pressures. Code implementations with small locality and high concurrency are more suitable for high-interference scenarios, while those with large locality and low concurrency perform better in low-interference scenarios.
(2) **Dynamic Interference Awareness**: By monitoring performance counters (such as L3 cache miss Rate, L3 access, instruction per cycle (IPC), etc.), the system dynamically predicts the interference level, selects the according code version to ensure quality of service (QoS) in multi-tenant environments.

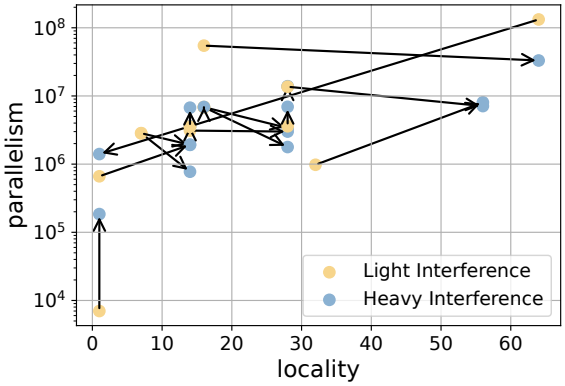


Figure 3: Parallelism and locality cannot effectively distinguish between high-interference and low-interference code versions on a shared GPU.

However, the above techniques cannot be directly applied to shared GPUs due to two factors: the difference between CPU and GPU architectures, and the lack of necessary performance data.

Difference between CPU and GPU: The architecture of GPUs is more complex than that of CPUs, and relying solely on the Pareto frontier of the **parallelism-locality** trade-off space is insufficient to distinguish code versions suitable for different interference levels. We attempted to replicate VELTAIR's single-pass multi-version compilation strategy on GPU architectures, where: **parallelism** is interpreted as the number of CUDA threads multiplied by the loop unrolling factor, and **locality** is interpreted as the tile size computed by each thread. Figure 3 illustrates the differences in parallelism and locality of the optimal implementations under varying interference pressures on GPUs, with arrows linking optimal points from low to high interference. VELTAIR's findings on CPUs indicate that these arrows predominantly point towards the upper left on CPUs, suggesting that code with higher concurrency but lower locality performs better in high-interference scenarios. However, our experiments on GPUs reveal no such consistent pattern, and optimal configurations appear more chaotic. This indicates that the parallelism-locality trade-off does not generalize well to GPU environments. This fundamental difference prevents the direct application of VELTAIR's method to shared GPU scenarios.

Unavailable Performance Data: VELTAIR's scheduling strategy relies on CPU performance counters to assess interference pressure in real time. On GPUs, however, these counters cannot be accessed concurrently while executing DNN inference tasks. To obtain such performance data, profiling is required, which introduces thousands of times additional overhead, making it impractical for real-time interference pressure evaluation.

3 Design & Implementation

3.1 Problem Formulation

Recent research indicates that low GPU utilization is a prevalent issue in deep learning clusters [16, 23, 43, 44]. A promising approach to ameliorate this challenge involves concurrent inference by sharing a single GPU among multiple tasks. By executing multiple inference tasks on the same GPU in parallel, we can maximize the GPU's computational capacity and minimize idle time. This method not only enhances the efficiency of individual GPUs but also boosts the overall processing capability and responsiveness of the cluster without incurring additional hardware costs, thereby optimizing the system's overall performance.

However, optimizing the compilation of DNN models for concurrent execution on shared GPU remains an open problem due to the following challenges: (a) **Resource Contention**: When multiple DNN inference tasks share a GPU, they compete for computational resources (e.g., SMs, memory bandwidth), leading to performance degradation. (b) **Code Selection**: Traditional DNN compilers generate optimized code for single-task execution but do not consider the concurrent execution context, resulting in suboptimal performance in shared GPU scenarios. (c) **Concurrency-aware Scheduling**: The selection of compiled DNN kernels must account for both the available GPU resources and the concurrent inference workload to maximize throughput.

To address these challenges, we formulate the problem in two stages:

Stage 1: Multi-version Code Generation. Given a DNN model M and a target GPU architecture G, the goal is to generate a set of code variants $C = \{c_1, c_2, \ldots, c_k\}$, where each variant c_j is optimized for executing M with a specific GPU

resource fraction, particularly j/k of the available Streaming Multiprocessors (SMs).

Stage 2: GPU Resource Allocation and Code Selection. Given:

- A set of DNN models $M = \{M_1, M_2, \ldots, M_N\}$ to be executed concurrently.
- A target GPU architecture G.
- A set of candidate code versions $C_i = \{c_{i1}, c_{i2}, \ldots, c_{ik}\}$ for each model M_i, each with different performance trade-offs.
- A maximum concurrency level L, i.e., the number of models that can execute simultaneously.

Our objective is to find the optimal combination of model-code pairs (M_i, c_{ij}) and their resource allocation r_i such that the **overall throughput** (inference tasks per second) is maximized on GPU G:

$$\max_{\{c_{ij}, r_i\}} \sum_{i=1}^{N} \text{Throughput}(M_i, c_{ij}, r_i),$$

where $\text{Throughput}(M_i, c_{ij}, r_i)$ represents the throughput of model M_i when executed with code variant c_{ij} and allocated resource r_i.

Concurrency Constraint: At any time, the number of simultaneously executing models should not exceed L:

$$|\{i \mid M_i \text{ is running}\}| \leq L$$

This formulation defines the concurrent DNN inference compilation problem as a resource allocation and code selection optimization problem. Our proposed method addresses this by searching for the best combination of compiled code variants and resource allocation to maximize throughput while respecting GPU constraints.

To reduce the online optimization overhead, we leverage a key observation: in production environments, workload patterns are often periodic and predictable [13, 14, 22, 43]. This stability allows us to pre-search and cache optimal code selections and resource allocations for a manageable set of expected workload combinations.

3.2 System Overview

ConCo consists of two aspects: compilation and runtime scheduling, as illustrated in Figure 4. The ConCo compiler generates multiple candidate codes for the target model, each optimized for peak performance under different MPS GPU resource constraints. To simplify the problem, ConCo categorizes the resource constraints into multiple distinct levels ranging from 0 to 100%.

In practical production environments, there are numerous repetitive tasks [43]. For instance, a GPU cluster deployed to support product recommendation systems receives thousands of inference tasks from recommendation models every

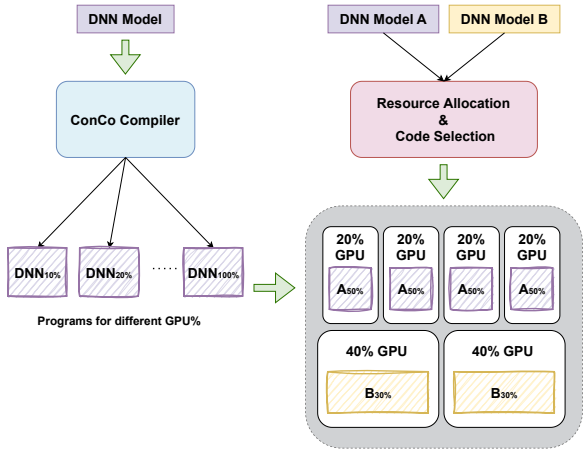


Figure 4: ConCo System overview

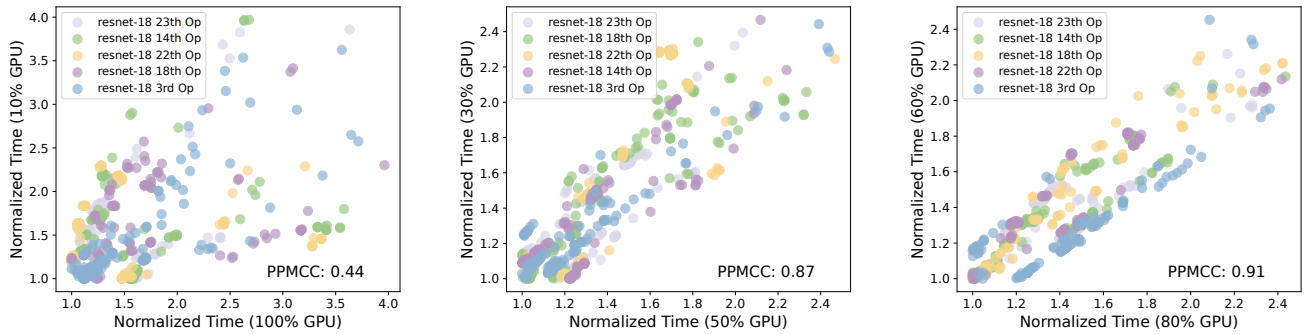
second. Based on this, we assume that the DNN inference tasks processed on the GPU and the arrival rates of each model task are known. Leveraging this assumption, we have designed our resource allocation and code selection scheme. This scheme selects the optimal MPS resource allocation for each DNN model task and picks the most suitable code from all candidate versions for the given concurrency scenario.

3.3 Concurrency-Aware Code Generation

To support efficient concurrent execution of DNN models under dynamic resource allocation, ConCo generates multiple code variants optimized for different GPU resource configurations. The key insight is that code performance is highly sensitive to the available Streaming Multiprocessors (SMs), which can vary significantly in multi-tenant environments.

We leverage NVIDIA's Multi-Process Service (MPS) to emulate different resource constraints by controlling the active thread percentage during the compilation process. Specifically, by limiting the available SMs through MPS, we guide the compiler to generate code variants that are optimized for specific resource allocation levels (e.g., 10%, 20%, ..., 100% of total SMs). This approach ensures that each variant is tailored to achieve optimal performance under its target resource constraint.

However, naively generating these variants would require independent tuning for each configuration, leading to prohibitive compilation overhead. To address this, we introduce a shared optimization space that exploits hardware commonalities across different SM allocations, as discussed in Section 3.4. This enables ConCo to efficiently produce a spectrum of high-performance code variants while significantly reducing the overall compilation cost.



(a) The normalized execution latency under 10% and 100% GPU resource constraints.

(b) The normalized execution latency under 30% and 50% GPU resource constraints.

(c) The normalized execution latency under 60% and 80% GPU resource constraints.

Figure 5: The correlation between normalized execution latencies of different kernel implementations of ResNet-18 operators under various GPU resource constraints on the 2080 Ti.

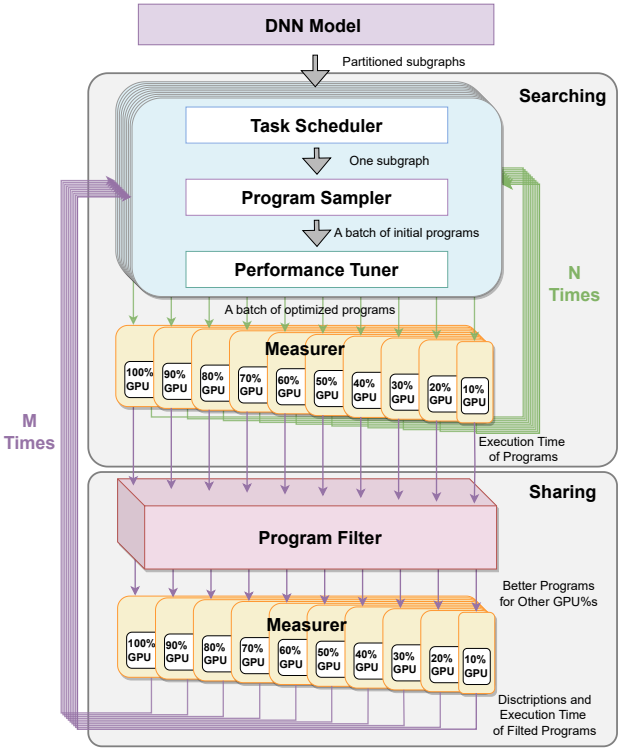


Figure 6: Compilation workflow of ConCo

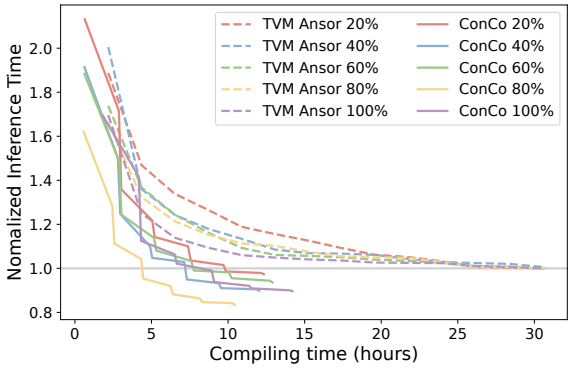
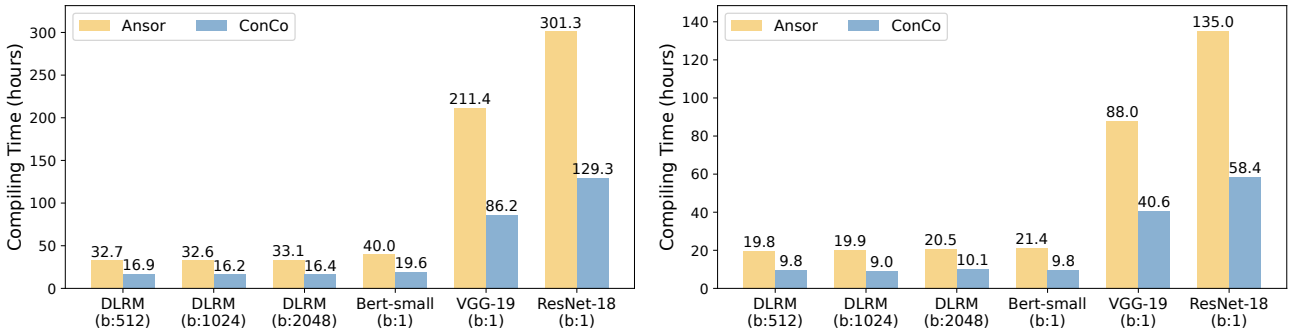


Figure 7: Change of inference time for Resnet-18 by compilation time on RTX 2080 Ti

3.4 Efficient Code Generation Strategy

Generating performance-optimal code for a target model under varying MPS GPU resource constraints can be achieved by adjusting the active thread percentage in MPS to limit the available GPU SMs during the traditional Ansor measurement process. However, producing multiple code versions for different SM allocations requires repeating this process for each configuration. Given that compiling a single model with Ansor typically takes several hours to days, generating 10 versions could extend the total compilation time to several days or even weeks, making this approach highly time-consuming and impractical.

Figure 5 illustrates the normalized execution latency of ResNet-18 operators under various resource constraints on a 2080 Ti GPU (i.e., the ratio of each operator's execution latency using different implementations relative to the optimal implementation). Each point corresponds to a distinct code implementation, with the x-axis and y-axis representing the normalized execution times under 10% GPU and 100% GPU resource allocations, respectively. As the resource allocation percentages become more similar, the PPMCC (Pearson product-moment correlation coefficient) increases, reflecting a stronger correlation in execution times across different



(a) RTX 2080 Ti (b) A100 80 GB

Figure 8: Comparison of compilation times

resource allocations. The PPMCC is defined as follows,

$$PPMCC(X,Y) = \frac{\sum_{i=1}^{n}[(X_i - \overline{X})(Y_i - \overline{Y})]}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}} \quad (1)$$

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i,\ \overline{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i \quad (2)$$

where, n denotes the number of points, and X_i and Y_i represent the coordinates of the data points on the x-axis and y-axis, respectively.

It was observed that code that performs well under one resource constraint does not significantly underperform under other resource constraints. Additionally, we posit that code utilizing different SM proportions on the same GPU may share certain hardware commonalities such as the number of compute cores per SM, L2 Cache size, and bandwidth. Hence, the tuning process for generating code with differing SM GPU resource proportions should not be entirely independent. Based on these observations and considerations, we designed a method of optimal code configuration sharing that accelerates the convergence speed of code generation across diverse SM resource constraints.

As depicted in Figure 6, the execution flow chart for the Compiling component of ConCo entails multiple rounds of optimal code sharing. During the compiling process, ConCo engages in M rounds of optimal code sharing. Before each sharing round, TVM Ansor is utilized for N rounds of operator tuning search for each resource constraint. The Program Filter screens out the top $\alpha\%$ performing code from the current tuning round based on measure performance, and collates code filtered from the remaining GPU percentages for performance measurement on the current GPU percentage hardware. For instance, if we were to split the search space for GPU resource constraints into 10 levels spread out from 10% to 100% uniformly, for one of the rounds of compiling code under the 10% resource constraint, after ConCo executes N * batch_size operator tuning searches, it gathers a total of $9\alpha\%N \cdot batch\_size$ code pieces optimized under the remaining resource constraints (i.e., 20%, 30%, ..., 100%) and measures their performance under the 10% constraint. These filtered codes and their execution times across various resource constraints are then provided to the subsequent round's corresponding GPU percentage constraint Ansor search to train the Cost Model. If these shared codes perform exceptionally, they can also serve as the initial population for the evolutionary search.

3.5 GPU Resource Allocation and Code Selection

In our experiments, we discovered that optimal GPU resource utilization often necessitates over-provisioning concurrently, meaning that the total allocation of resources to all concurrent processes exceeds 100%. To achieve higher resource utilization rates, we implemented over-provisioning, thereby setting tasks to occupy a designated percentage of GPU resources, which typically results in these tasks contending for resources with others. Consequently, despite generating ten distinct versions of code for different GPU resource percentages, the optimal execution strategy is not to run the version generated under a specific GPU percentage constraint within the same percentage-limited resource framework.

For a task that requires running n different models, we need to determine the optimal code version $C_i\%$ and runtime resource constraint $R_i\%$ for each model $i \in \{1, \ldots, n\}$ to achieve the highest overall throughput. Here, the code version $C_i\%$ represents the code compiled for model i on a GPU with a resource constraint of $C_i\%$. To simplify the problem, we restrict the values of $C_i\%$ and $R_i\%$ to discrete values with a 10% interval. For this combinatorial optimization problem,

with a search space of 10^{2n}, we use simulated annealing to search for the optimal configuration.

4 Evaluation

4.1 Setup

We have selected a series of classic models for evaluation, encompassing CNN-based models (such as ResNet-18 [12] and VGG-19 [38]), Transformer-based models (such as BERT-Small [7]), as well as recommendation systems (such as Facebook DLRM [30]). The testing was conducted utilizing two types of GPUs: Nvidia RTX 2080 Ti and Nvidia A100 80 GB. Specifically, the Nvidia RTX 2080 Ti was paired with an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, while the Nvidia A100 80 GB was paired with an Intel(R) Xeon(R) w5-2455X CPU.

As elucidated in Section 4.2, the code obtained via ConCo surpasses that generated by traditional Ansor. Code generated by ConCo for 100% GPU resource allocation consistently outperforms that produced by Ansor, rendering Ansor-generated code as a baseline somewhat unfair. Consequently, we have opted to regard the code, generated by ConCo for 100% resource allocation, as the optimal benchmark for exclusive use of all GPU resources.

Our benchmarking encompasses two principal scenarios, both employing code generated by ConCo for 100% resource allocation. The distinction between these scenarios lies in their execution environments: **Baseline1** operates under the default MPS mode (i.e., without imposing a limit on the maximum usable resources per process), whereas **Baseline2** traverses the entire resource limitation space to identify the allocation scheme that maximizes overall throughput, subsequently running concurrently under MPS with the optimal resource allocation scheme.

4.2 Cost of Compiling

Figure 8 presents the compilation times for generating all 10 candidate codes on the 2080 Ti and A100 GPUs, respectively. The findings illustrate that the compilation time of our method is approximately half that of the conventional Ansor approach. Specifically, Figure 7 delineates the curve of normalized inference time for ResNet-18 as a function of compilation time under varying GPU resource constraints. Here, the number of measurement trials for Ansor is configured according to the TVM documentation recommendation of $900 \times len(tasks)$ [47], where $len(tasks)$ denotes the number of sub-tasks (i.e., subgraphs) partitioned by Ansor. The normalized inference time refers to the ratio of the inference latency of the current code to the inference latency of the optimal implementation generated by Ansor at the final moment under the corresponding resource proportion.

The experimental results demonstrate that ConCo can significantly enhance the average convergence speed for the generated 10 codes. To achieve the compilation effectiveness of traditional Ansor, ConCo requires only 14.3%-32.3% of the time needed by conventional Ansor. After evaluating the trade-off between the effectiveness of generated codes and the compilation duration, we set the compilation hyperparameters to $measuring\_batch\_size = 64$, $N = len(tasks)$, $M = 5$, and $\alpha\% = 2.5\%$. Under this hyperparameter configuration, we can obtain codes with superior performance to those generated by traditional Ansor in less than half the time.

4.3 Results for Compiled Code

Tables 3 and 4 elucidate the inference latency of codes generated by ConCo under various GPU resource constraints. Upon analysis, it becomes evident that the minimal delay in each column is either located on the diagonal of the table or exhibits negligible deviation from the diagonal values. This indicates that when a model operates under a resource constraint R, the code compiled with constraint C invariably exhibits the lowest latency compared to all other codes. Consequently, this experimental observation substantiates that our compilation method indeed guarantees the optimality of the code with respect to the corresponding resource configuration.

4.4 Performance of Operators

We performed experiments on processes that execute a single type of operator only to validate the efficacy of our compilation optimization strategy, ConCo. Given that the execution time of a single operator is typically very short, often in the microsecond range, scheduling on the CPU would introduce unacceptable overhead. Consequently, in this segment of the evaluation, each process repetitively executed the same operator 500 times. Subsequently, we measured the Job Completion Time (JCT) for the concurrent execution of multiple such processes. This JCT serves as the optimization target for ConCo and the selection criterion for resource allocation constraints of Baseline 2.

From ResNet-18, Bert-small, VGG-19, and DLRM, we selected operators with relatively long execution times, specifically those ranging from tens to hundreds of microseconds when executed exclusively on a GPU. Table 2 compares the JCT reductions achieved by our ConCo relative to the baseline approaches:

$$Reduce\ k = \frac{JCT_{Baseline\ k} - JCT_{ConCo}}{JCT_{Baseline\ k}} \quad k = 1, 2$$

Empirical results indicate that ConCo can reduce the JCT for concurrent execution of single operator processes by up to 69.85% compared to Baseline 1 and by up to 22.44%

GPU	Operate 1	Operate 2	batch size	concurrency	Reduce 1	Reduce 2
2080 Ti	2^{nd} $OP_{resnet-18}$	6^{th} $OP_{bert-small}$	1, 1	20, 20	27.22%	21.85%
	2^{nd} $OP_{resnet-18}$	4^{th} $OP_{bert-small}$	1, 1	20, 20	17.24%	16.55%
	2^{nd} $OP_{resnet-18}$	9^{th} OP_{vgg-19}	1, 1	1, 1	16.49%	2.58%
	14^{th} $OP_{resnet-18}$	4^{th} $OP_{bert-small}$	1, 1	5, 5	16.22%	3.83%
	22^{th} $OP_{resnet-18}$	4^{th} $OP_{bert-small}$	1, 1	5, 5	14.70%	4.81%
	5^{th} OP_{dlrm}	3^{rd} $OP_{bert-small}$	1024, 1	1, 5	13.89%	9.05%
	2^{nd} $OP_{resnet-18}$	4^{th} $OP_{bert-small}$	1, 1	5, 5	13.09%	11.72%
	2^{nd} OP_{dlrm}	3^{rd} $OP_{bert-small}$	1024, 1	1, 5	11.49%	9.56%
	14^{th} $OP_{resnet-18}$	1^{st} OP_{vgg-19}	1, 1	10, 10	11.14%	2.09%
	1^{st} OP_{vgg-19}	3^{rd} $OP_{bert-small}$	1, 1	5, 5	10.16%	4.68%
A100 80 GB	22^{th} $OP_{resnet-18}$	6^{th} $OP_{bert-small}$	1, 1	10, 10	69.85%	11.91%
	9^{th} OP_{vgg-19}	6^{th} $OP_{bert-small}$	1, 1	5, 5	52.18%	11.38%
	15^{th} OP_{vgg-19}	3^{rd} $OP_{bert-small}$	1, 1	5, 5	33.08%	8.79%
	5^{th} OP_{dlrm}	3^{rd} $OP_{bert-small}$	1024, 1	1, 5	32.05%	14.68%
	15^{th} OP_{vgg-19}	6^{th} $OP_{bert-small}$	1, 1	5, 5	31.04%	22.44%
	2^{nd} OP_{dlrm}	3^{rd} $OP_{bert-small}$	1024, 1	1, 5	30.23%	9.36%
	5^{th} OP_{dlrm}	4^{th} $OP_{bert-small}$	1024, 1	1, 5	26.86%	3.64%
	14^{th} $OP_{resnet-18}$	6^{th} $OP_{bert-small}$	1, 1	10, 10	22.35%	6.44%
	2^{nd} OP_{dlrm}	6^{th} $OP_{bert-small}$	1024, 1	1, 5	19.99%	19.61%
	22^{th} $OP_{resnet-18}$	6^{th} $OP_{bert-small}$	1, 1	5, 5	18.60%	8.12%
	22^{th} $OP_{resnet-18}$	9^{th} OP_{vgg-19}	1, 1	10, 10	17.11%	5.25%

Table 2: Comparison of JCT for Concurrent Execution of Single Operator Processes

C% \ R%	10%	30%	50%	70%	90%	100%
10%	**4.92**	1.94	1.57	1.5	1.44	1.42
30%	5.11	**1.90**	1.54	1.44	1.34	1.30
50%	5.85	2.10	**1.37**	1.23	1.16	1.07
70%	5.89	2.11	1.39	**1.18**	1.15	1.08
90%	6.31	2.19	1.44	1.27	**1.12**	1.06
100%	6.44	2.23	1.45	1.27	1.14	**1.02**

Table 3: Inference latency (ms) of ConCo-generated codes for ResNet-18 under different GPU resource constraints on RTX 2080 Ti

C% \ R%	10%	30%	50%	70%	90%	100%
10%	**10.3**	5.46	3.75	3.7	3.62	3.61
30%	10.9	**3.78**	3.13	3.07	2.15	2.14
50%	11.4	4.04	**2.63**	2.51	2.25	2.26
70%	12.5	4.34	2.71	**2.35**	2.18	2.16
90%	12.7	4.27	2.90	2.54	**1.76**	**1.75**
100%	12.6	4.25	2.89	2.56	1.80	1.76

Table 4: Inference latency (ms) of ConCo-generated codes for bert-small under different GPU resource constraints on RTX 2080 Ti

compared to Baseline 2. This represents a substantial performance enhancement.

4.5 End-to-end Performance

Given that the arrival proportions of the various models are known, to ensure fairness among models, we employed a global queue scheduling model. In this model, different models are enqueued at proportionate, equal time intervals, with tasks being dequeued in a First-In-First-Out (FIFO) manner. This operational mode ensures comparable throughput for different models within the same concurrent testing environment, thereby allowing us to evaluate concurrent performance based on overall throughput.

Table 5 enumerates the concurrent tasks utilized in the testing, where the arrival rates of each model in the multi-model concurrent task experiments were set to identical values. The experimental results are depicted in Figures 9 and 10. The experiments demonstrate that on both 2080 Ti and A100, ConCo achieves up to a 1.16× and 1.12× increase in throughput, respectively, compared to Baseline2. This improvement indicates that although code compiled with 100% resource utilization performs optimally when the entire GPU is exclusively used, in concurrent scenarios, such code is outperformed by code compiled with restrained GPU resource utilization.

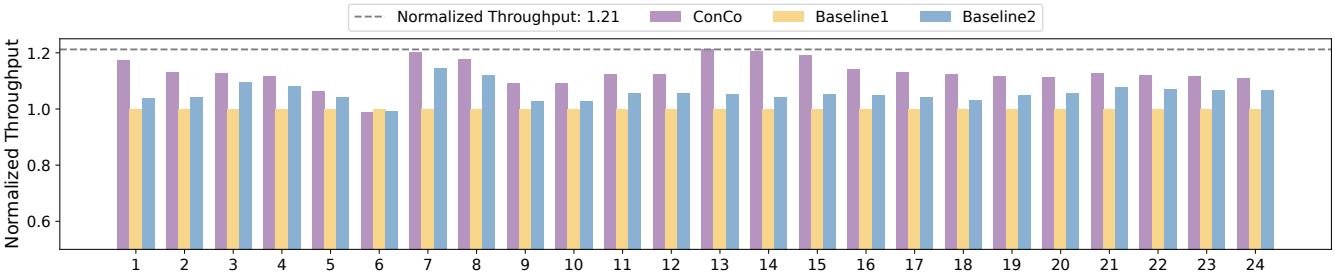


Figure 9: Normalized Throughput on RTX 2080 Ti

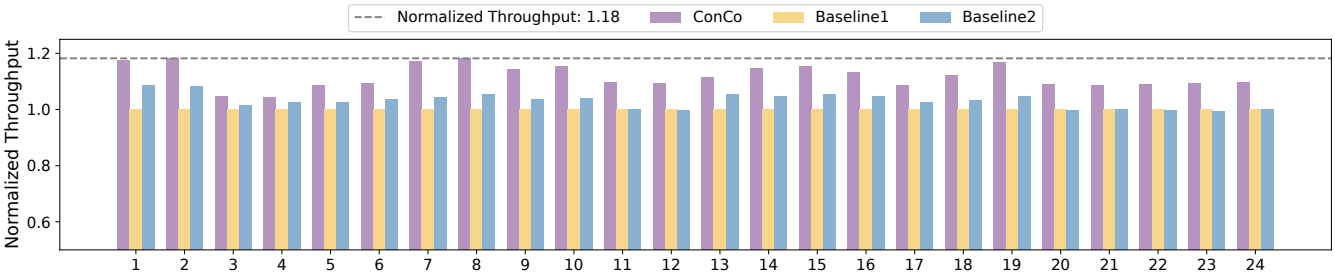


Figure 10: Normalized Throughput on A100 80 GB

Experimental results indicate that the performance improvements demonstrated by ConCo in end-to-end concurrent DNN model tests are inferior to those observed in experiments where only a single operator is executed concurrently. This phenomenon can be attributed to the intrinsic complexity of DNN models, which comprise multiple operators (or sub-tasks). For instance, the ResNet-18 model includes 24 distinct operators, whereas the Bert-small model comprises 9 distinct operators. Theoretically, to achieve optimal performance, the selection of candidate code and resource allocation strategy should differ when a specific operator from ResNet-18 runs concurrently with a different operator from Bert-small.

Due to the proprietary nature of Nvidia's GPU architecture, we lack control over the scheduling of operators on the GPU. On a shared GPU, the operators from different models that end up running concurrently are determined randomly. Consequently, ConCo's search strategy, which aims to maximize concurrent throughput, must balance the considerations for all possible pairs of concurrent operators. This results in code combinations and resource allocation strategies that are suboptimal compared to those designed for single-operator processes. Therefore, the performance gains achievable by ConCo in end-to-end concurrent DNN model execution are less significant than those in single-operator concurrent execution scenarios.

4.6 Features Analysis

To investigate why ConCo outperforms Ansor in concurrent scenarios, we train a random forest model to identify the features that most significantly influence performance. First, we extract 164 features, consistent with those used in the cost model of Ansor. These features encompass aspects related to computation, buffer access patterns, arithmetic intensity, memory allocation, and other relevant factors. In the end-to-end experiments described in Section 4.5, we evaluate both the ConCo-generated code and the baseline code (i.e., the Ansor implementation) for each subtask within the same concurrent scenario. For each pair of implementations, we randomly assign one as Code A and the other as Code B. After extracting the 164 features from both Code A and Code B, we define a label, `A_is_ConCo`, to distinguish the two. The label `A_is_ConCo` takes a value of 1 if Code A corresponds to the ConCo implementation, and 0 otherwise. To enhance the analysis, we introduce additional features to capture the differences between Code A and Code B, focusing on their relative performance characteristics.

$$\Delta(feature_i) = \frac{A.feature_i - B.feature_i}{A.feature_i + 1}, \quad i = 1, 2, \ldots, 164 \tag{3}$$

We partitioned the dataset into training and testing sets with a 7:3 ratio and trained a random forest model to predict the label `A_is_ConCo`. The results demonstrate that the model achieves an accuracy of 92.11% on the testing set,

Idx	Model	p_{2080Ti}	p_{A100}	Batch_size
1	Bert-small	20	20	1
2	Bert-small	40	40	1
3	DLRM	20	20	512
4	DLRM	40	40	512
5	DLRM	20	20	1024
6	DLRM	40	40	1024
7	DLRM	20	20	2048
8	DLRM	40	40	2048
9	ResNet-18	20	20	1
10	ResNet-18	40	40	1
11	VGG-19	10	20	1
12	VGG-19	14	30	1
13	Bert-small, DLRM	20	20	1, 512
14	Bert-small, DLRM	20	20	1, 1024
15	Bert-small, DLRM	20	20	1, 2048
16	ResNet-18, Bert-small	20	20	1, 1
17	ResNet-18, DLRM	20	20	1, 512
18	ResNet-18, DLRM	20	20	1, 1024
19	ResNet-18, DLRM	20	20	1, 2048
20	ResNet-18, VGG-19	10	15	1, 1
21	VGG-19, Bert-small	10	15	1, 1
22	VGG-19, DLRM	10	15	1, 512
23	VGG-19, DLRM	10	15	1, 1024
24	VGG-19, DLRM	10	15	1, 2048

Table 5: Tasks setup. p_{2080Ti} and p_{A100} represent the maximum concurrency of experiments on RTX 2080 Ti and A100 80 GB, respectively.

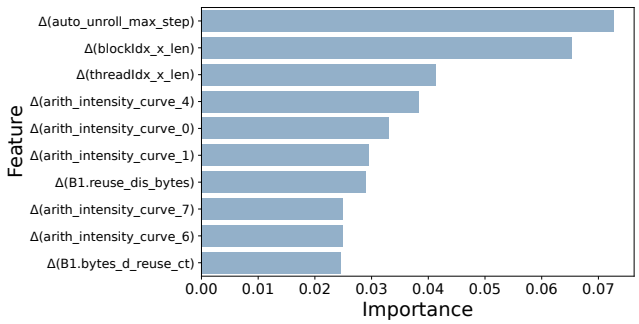


Figure 11: The top-10 feature importance of random-forest-based predictor

indicating its effectiveness in distinguishing between the generated codes of ConCo and Ansor. Figure 11 presents the top 10 most important features identified by the random forest model. Among these, the features with the highest influence on the prediction include `auto_unroll_max_step`, `blockIdx_x_len`, `threadIdx_x_len`, etc.

As shown in Figure 12, ConCo tends to generate code with more aggressive loop unrolling, fewer blocks, fewer threads

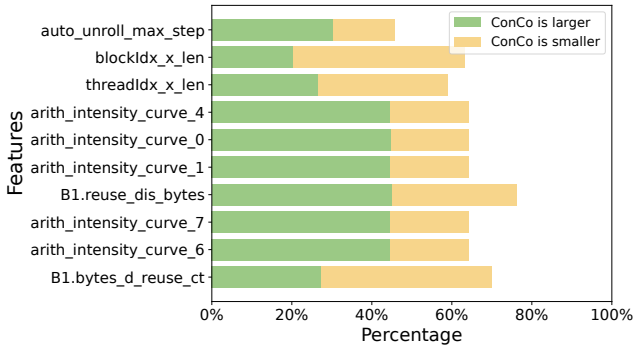


Figure 12: Percentage of samples where ConCo feature values are larger or smaller than Ansor feature values

per block, and higher computational intensity compared to Ansor. Increased loop unrolling allows the GPU to issue more independent instructions per cycle, reducing control flow overhead and enhancing pipeline utilization. This helps mask instruction latency, particularly for memory-bound operations, by keeping execution units active. Additionally, the use of fewer thread blocks and fewer threads per block increases the workload assigned to each thread. This strategy reduces scheduling overhead and improves register and shared memory reuse, as threads retain more data in fast on-chip memory rather than frequently accessing global memory. Furthermore, ConCo-generated code tends to leverage higher computational intensity, meaning a greater ratio of arithmetic operations to memory accesses. This improves compute unit utilization while reducing pressure on global memory bandwidth, a critical bottleneck in high-concurrency scenarios. By minimizing memory transactions and improving data locality, ConCo effectively alleviates contention for memory bandwidth when multiple tasks execute concurrently.

In summary, ConCo's optimizations—including aggressive loop unrolling, efficient workload distribution, computationally intensive operator selection, and improved memory resource utilization—enable it to achieve superior throughput in concurrent execution environments. These advantages become particularly pronounced under high contention for GPU resources, allowing ConCo to consistently outperform Ansor in such conditions.

5 Related Work

5.1 DNN Compilers

DNN compilers typically optimize code generation through two complementary levels: computational graph-level and operator-level. At the graph level, techniques like layout optimization [25], operator fusion [3], constant folding [36], automatic batching, and subgraph generation and replacement [17] are widely used. Frameworks such as TensorFlow XLA

[40] and DLVM [42] serve as computational graph-level optimization tools tailored for deep learning. Specifically, TASO [17] is noted as the first deep neural network (DNN) optimizer that incorporates automatic subgraph replacements.

Operator-level optimizations include loop unrolling, data tiling, memory management, and loop reordering. Roller [49], a DNN compiler targeting operator-level optimizations, enhances operator performance by exploiting saturated data processing pipelines, thereby boosting the overall performance of DNN models.

Among contemporary DNN compilers, TVM [3] is one of the most widely adopted deep learning compilers. Its intermediate representation, Relay [36], offers rich expressiveness for DNN models, coupled with optimization and transformation ease. TVM leverages and extends the scheduling primitives of Halide [35] to optimize the performance of target hardware. To perform kernel-level optimization, TVM searches for optimal configurations within a defined search space, utilizing strategies such as AutoTVM [4] and Ansor [48]. While AutoTVM necessitates predefined operator templates, Ansor expands the search space and removes the need for manual template design.

Besides the aforementioned DNN compilers, other notable compilers include Microsoft's Rammer [27], Google's Multi-Level Intermediate Representation (MLIR) [19], Facebook's Glow [37], and Tensor Comprehensions [41], introduced by Vasilache et al..

Despite the availability of a diverse array of DNN compilers, none explicitly generate optimized low-level DNN code specialized for concurrent execution scenarios on shared GPUs. SLICE-TUNE [9], proposed by Dhakal et al., attempts to identify optimal code at performance inflection points by combining multiple DNN compilers (e.g., TVM, Ansor, and Chameleon [1]). However, real-world concurrent DNN workloads rarely match these exact inflection points, limiting the practical benefits of this approach.

VELTAIR [26] improves concurrent DNN inference on CPUs by selecting implementations from a Pareto frontier of parallelism-locality trade-offs. Its runtime interference models leverage CPU performance counters. However, due to fundamental differences between CPU and GPU architectures, VELTAIR's methods, such as core-affinity binding and interference assessment, are incompatible with GPU architectures.

5.2 Resource management for shared GPUs

Efficient resource management is critical for concurrent DNN inference on shared GPUs. Spatial partitioning of GPU resources allows for the allocation of isolated resources (e.g., the number of Streaming Multiprocessors (SMs) and memory bandwidth) to different DNN workloads. This approach mitigates inter-task interference at the hardware resource level. Current resource partitioning mechanisms predominantly rely on resource-sharing strategies provided by GPU vendors, such as the Multi-Process Service (MPS) and Multi-Instance GPU (MIG) architectures.

GSLICE[8] is a reconfiguration scheme based on MPS that utilizes spare processes to reduce reconfiguration downtime. MIG-serving[39] explores the optimal partitioning schemes for MIG. MISO[21] proposes a machine learning-based methodology, leveraging runtime predictions from MPS schemes to infer the efficacy of MIG schemes and consequently determine the optimal MIG partitioning strategy.

Despite the notable efficiency in GPU resource utilization achieved by existing resource management and allocation schemes, there remains room for improvement. This is because the underlying code of DNN models is not deliberately tailored to the specific resource scenarios encountered in practical deployments. Consequently, further optimization is both necessary and possible.

6 Conclusion

This paper introduced ConCo, a concurrency-aware compilation framework for concurrent DNN inference tasks on shared GPUs. ConCo dynamically generates and selects optimal code variations based on available GPU resources, addressing the limitations of existing DNN compilers in concurrent scenarios. It has demonstrated promising improvements, increasing compilation speed by more than 2x, enhancing throughput by up to 1.2x for concurrent DNN inference tasks, and reducing JCT by up to 69.85% for concurrent single-operator processes compared to baselines.

ConCo's resource allocation and code selection strategy efficiently manage GPU resources, which is crucial for optimal performance in concurrent inference environments. Future work will focus on extending ConCo's capabilities to more hardware architectures and integrating with real-time systems requiring dynamic resource allocation. Enhancing adaptability to evolving DNN models and computational frameworks will also be a priority.

References

[1] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743* (2020).

[2] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI 18). 578–594.

[4] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[6] OpenBLAS contributors. 2024. OpenBLAS. https://github.com/OpenMathLib/OpenBLAS. Accessed: 2024-06.

[7] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[9] Aditya Dhakal, KK Ramakrishnan, Sameer G Kulkarni, Puneet Sharma, and Junguk Cho. 2022. Slice-tune: A system for high performance dnn autotuning. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. 228–240.

[10] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems* 3 (2021), 167–180.

[11] Google. 2024. IREE: Intermediate Representation Execution Environment. https://github.com/iree-org/iree. Accessed: 2024-06.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[13] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[14] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 457–472.

[15] Intel. 2024. oneAPI. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html. Accessed: 2024-06.

[16] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

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

[18] Turja Kundu and Tong Shu. 2023. HIOS: Hierarchical inter-operator scheduler for real-time inference of DAG-structured deep learning models on multiple GPUs. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 95–106.

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[20] Matthew LeMay, Shijian Li, and Tian Guo. 2020. Perseus: Characterizing performance and cost of multi-tenant serving for cnn models. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 66–72.

[21] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*. 173–189.

[22] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 835–850.

[23] Zhongjin Li, Victor Chang, Haiyang Hu, Maozhong Fu, Jidong Ge, and Francesco Piccialli. 2021. Optimizing makespan and resource utilization for multi-DNN training in GPU cluster. *Future Generation Computer Systems* 125 (2021), 206–220.

[24] Changyao Lin and Jie Liu. 2024. COS: Cross-Processor Operator Scheduling for Multi-Tenant Deep Learning Inference. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

[25] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1025–1040.

[26] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 388–401.

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

[28] Long Short-Term Memory. 1997. Sepp hochreiter and jürgen schmidhuber. *Neural Computation* 9, 8 (1997), 1735.

[29] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Interference-aware scheduling for inference serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 80–88.

[30] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[31] Nvidia. 2024. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: 2024-06.

[32] Nvidia. 2024. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency. Accessed: 2024-06.

[33] Nvidia. 2024. MULTI-PROCESS SERVICE. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 2024-06.

[34] Nvidia. 2024. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf. Accessed: 2024-06.

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

[36] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A high-level compiler for deep learning. *arXiv preprint arXiv:1904.08368* (2019).

[37] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele,

Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[38] Karen Simonyan. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[39] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).

[40] TensorFlow. 2024. TensorFlow XLA. https://www.tensorflow.org/xla. Accessed: 2024-06.

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

[42] Richard Wei, Lane Schwartz, and Vikram Adve. 2017. DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016* (2017).

[43] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.

[44] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[45] Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. 2021. Automated runtime-aware scheduling for multi-tenant DNN inference on GPU. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[46] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. 2022. A survey of multi-tenant deep learning inference on gpu. *arXiv preprint arXiv:2203.09040* (2022).

[47] Lianmin Zheng. 2024. Auto-scheduling a Neural Network for NVIDIA GPU. https://tvm.apache.org/docs/how_to/tune_with_autoscheduler/tune_network_cuda.html. Accessed: 2024-06.

[48] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[49] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.

[50] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.