RESEARCH-ARTICLE

PPD: A Portable and Highly Parallel Dispatching System for Deep Learning

WENDONG XU, The University of Hong Kong, Hong Kong, Hong Kong

YUHAO JI, Chinese University of Hong Kong, Hong Kong, Hong Kong

YANG BAI, Chinese University of Hong Kong, Hong Kong, Hong Kong

YUETING LI, Beihang University, Beijing, China

YUXUAN ZHAO, Chinese University of Hong Kong, Hong Kong, Hong Kong

ZHENGWU LIU, The University of Hong Kong, Hong Kong, Hong Kong

View all

Open Access Support provided by:

Beihang University

The University of Hong Kong

Chinese University of Hong Kong

Published: 17 December 2025
Online AM: 27 October 2025
Accepted: 15 October 2025
Revised: 11 October 2025
Received: 03 March 2025

PPD: A Portable and Highly Parallel Dispatching System for Deep Learning

WENDONG XU, Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong
YUHAO JI, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
YANG BAI, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
YUETING LI, Beihang University, Beijing, China
YUXUAN ZHAO, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
ZHENGWU LIU, Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong
BEI YU, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
NGAI WONG, Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong

The acceleration of inference process for deep learning models is closely tied with the parallelization capability of computational graph operators and the parallel scheduling strategies. Most existing deep learning compilers focus on optimizing intra-operator parallelism, while neglecting inter-operator parallelism. Furthermore, most industrial inference engines, such as PyTorch and TensorFlow, utilize a dataflow-based model to describe tasks and schedule operators. They are computationally expensive and operate in a topological order and are parallelized to run within a single CUDA stream. However, they fail to fully exploit the parallelism capabilities of multiple CUDA streams. In this article, we propose PPD, a portable, highly parallel dispatching system. It boosts the inference performance by dividing the computational graph into multiple taskflow-based subgraphs. Additionally, PPD entails a dispatching algorithm on a single GPU with multiple CUDA streams to enhance

This work was supported in part by the Theme-based Research Scheme (TRS) project T45-701/22-R, the AVNET-HKU Emerging Microelectronics and Ubiquitous Systems (EMUS) Lab.
Authors' Contact Information: Wendong Xu (corresponding author), Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: wdxu@connect.hku.hk; Yuhao Ji, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: yhji24@cse.cuhk.edu.hk; Yang Bai, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: ybai62868@gmail.com; Yueting Li, Beihang University, Beijing, Beijing, China; e-mail: yuetingli@buaa.edu.cn; Yuxuan Zhao, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: yxzhao21@cse.cuhk.edu.hk; Zhengwu Liu, Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong; e-mail: zwliu@eee.hku.hk; Bei Yu, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: byu@cse.cuhk.edu.hk; Ngai Wong, Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: nwong@eee.hku.hk.

ACM 1084-4309/2025/12-ART20
https://doi.org/10.1145/3773039

the parallelism and performance of model inference. PPD offers users a lightweight model definition and an inference C++ interface, allowing for seamless integration into any context. We also verify the feasibility of PPD on AMD and other graphics cards. We validate our approach on widely adopted neural network models with varying degrees of parallelism, and compare it with industrial inference engines. Experiments demonstrate that PPD outperforms SOTA methods by up to **2.28**×.

CCS Concepts: • **General and reference**; • **Computer systems organization** → **Real-time systems**; **Parallel architectures**; • **Software and its engineering**;

Additional Key Words and Phrases: Model inference, parallel computing, code generation

ACM Reference Format:
Wendong Xu, Yuhao Ji, Yang Bai, Yueting Li, Yuxuan Zhao, Zhengwu Liu, Bei Yu, and Ngai Wong. 2025. PPD: A Portable and Highly Parallel Dispatching System for Deep Learning. *ACM Trans. Des. Autom. Electron. Syst.* 31, 2, Article 20 (December 2025), 24 pages. https://doi.org/10.1145/3773039

1 Introduction

In recent years, there has been a rapid advancement in the field of **deep learning** (**DL**), with deep neural networks being widely utilized across various domains. As the demand for model functionality and the volume of data continues to grow, there is an increasing expectation for improved predictive performance of deep neural network models. Currently, TensorFlow [1], PyTorch [2], MXNet [3], and other analogous DL frameworks are extensively employed in both research and industry. These frameworks facilitate researchers in defining model structures, as well as in training and inference, without the need to deal with the intricate connections between individual operators (atomic computational units such as convolution, matrix-multiplication, or activation) and the underlying framework logic. Therefore, these engines compromise performance in favor of ease of use, which may hinder their adaptability to high-performance scenarios.

The primary application in the industry is model inference. Numerous efforts have been made to achieve acceleration of neural network inference, which can be broadly categorized into compile-time and runtime optimization [4]. Compile-time optimization primarily focuses on high-performance implementations of network operators using DL compilers. On the other hand, runtime optimization mainly involves various inference execution engines, each with different optimizations.

TVM [5] is one of the first DL compilers designed to bridge productivity-focused frameworks and performance-oriented hardware backends, proposing compilation optimization techniques for neural network acceleration.

DL compiler operators that fuse operations focus on code generation and do not take into account the higher parallelism capability of devices [4]. The optimized fused operations are designed to execute as quickly as possible when running in sequence without considering the ability to achieve parallelism (for example, using multiple CUDA streams). This is due to the fact that most runtime executors of DL frameworks [1, 2, 5] take a dataflow-based organization and operate in a serial manner based on the topological order. It is hard for frameworks to determine which operators are suitable to run in parallel because of lacking profilings for these operators. As a result, many complex models, including those with parallel modules that are not data-dependent [6–9], are unable to benefit from this approach. That is, most DL frameworks have a shallow runtime scheduling strategy.

To address these limitations, we present PPD, a portable and highly parallel dispatching system for DL. PPD divides the computational graph—a **directed acyclic graph** (**DAG**) whose nodes are operators—into balanced subgraphs that minimize data dependencies. Each subgraph is treated as a single scheduling unit and assigned to dedicated CUDA streams for concurrent execution. Our

approach uses mini-benchmarking to equalize subgraph execution times and employs a **breadth-first order** (**BFO**) dispatching algorithm that pre-assigns CUDA stream IDs for runtime execution, effectively enhancing GPU utilization through multi-stream parallelism.

In response to the constraint that DL compilers do not fully account for device parallelism during operator fusion, we have designed a subgraph partitioning algorithm based on mini-benchmarking results. The partitioning algorithm considers the individual execution time of each operator to equalize subgraph execution times while reducing data dependencies between subgraphs by minimizing inter-subgraph connections.

PPD includes a code generation tool that produces taskflow-based computational graph code, resulting in portable C++ implementations for inference. The generated code only includes CUDA kernel definitions and taskflow-based structure definitions, enabling easy deployment to any system without heavy dependencies. Our contributions are summarized as follows:

— We propose a taskflow-based dispatching system to enhance model inference parallelism and improve performance.
— We present a subgraph partitioning and dispatching system that groups operators into balanced subgraphs assigned to multiple CUDA streams, along with a code generation pipeline that produces portable, operator-fused definitions suitable for direct deployment.
— We conduct extensive experiments on various neural network structures, demonstrating PPD as a portable and high-performance parallel dispatching system for DL.

2 Background and Related Works

2.1 Operator Parallelism and Operator Fusion

In DL systems, operators are the basic computational units that perform specific mathematical operations. Each operator consists of multiple operations, which are the individual mathematical computations that compose the operator.

Intra-operator parallelism [10] refers to parallelization within a single operator, where multiple operations within the same operator are executed concurrently. Inter-operator parallelism refers to parallelization between independent operators that can run simultaneously without data dependencies.

Operator fusion is a compilation technique that combines multiple operators into a single kernel to reduce memory access overhead and improve cache utilization [11].

2.2 DL Infrastructure

Traditional DL frameworks like PyTorch [12] and TensorFlow [1] have established the foundation for neural network development and deployment. However, these frameworks face significant challenges in achieving optimal parallelization due to their operator-level scheduling and limited exploitation of hardware parallelism. Recent advances in DL compilation have attempted to address these limitations through more sophisticated optimization strategies.

PyTorch Inductor [12], introduced in PyTorch v2.0, represents a modern compilation approach that lowers neural network models directly to efficient PTX/SASS implementations using OpenAI Triton [13] as the backend. This compilation-based approach offers significant performance improvements over traditional eager execution.

Ding et al. proposed Hidet [14], which is added to PyTorch as a compile backend [15]. Different from Triton, Hidet learns basic kernel implementations from the efficient template-based kernel library CUTLASS[16] to construct neural networks by converting the neural network defined by PyTorch into a high-performance CUDA implementation rather than directly lowering it to the assembly layer. In addition, Hidet also provides a task-mapping-oriented programming paradigm,

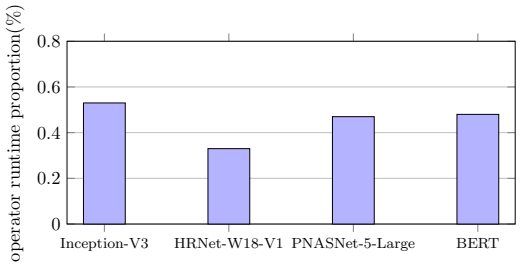


Fig. 1. Normalized execution time distribution of individual operators across different neural network models. Each operator consumes less than 1% of total execution time (approximately 0.5% on average), highlighting the fine-grained scheduling overhead issue that motivates our subgraph-level approach. Measurements are collected using our micro-benchmark harness on RTX 3090 with batch size 1; times are normalized per model to its end-to-end latency.

allowing CUDA novice users to quickly develop complex operators. Hidet further integrates basic operators through post-scheduling fusion to better reduce memory footprint.

However, like other DL compilers, Hidet cannot make use of multiple CUDA streams or CUDA Graph, leading to inadequate parallel scheduling.

2.3 Parallel Inference Optimization

Rammer [10] analyzes operators in the neural network graph into finer-grained scheduling units and assigns them to different hardware units (virtualized parallel devices) to implement inter-operator parallelism. However, Rammer's approach still suffers from significant scheduling overhead because it splits operators into fine-grained units by threadblocks on a single stream rather than using operator fusion or grouping strategies. The authors acknowledge that this fine-grained splitting and multi-stream usage can negatively impact GPU utilization due to excessive scheduling frequency.

2.4 Multiple CUDA Streams

The principle of CUDA stream was analyzed in [17–19] that using multiple CUDA streams can achieve better performance on some tasks. Recent studies [20–22] have demonstrated that effective multi-stream scheduling can significantly improve GPU utilization in DL workloads by overlapping computation and memory operations.

With the development of computational capability, GPU clock cycles consumed by CUDA kernels become shorter. Figure 1 illustrates that individual operators typically consume less than 1% of total execution time, creating excessive scheduling overhead when using traditional operator-level dispatching. This motivates our subgraph-level approach to reduce scheduling frequency. Additionally, directly assigning kernels to different CUDA streams may lead to L2 cache misses due to discontinuous intermediate data usage. Modern GPU architectures with advanced scheduling capabilities require careful consideration of stream management to avoid resource contention and maximize throughput. Empirically, Figure 8 shows that gains increase with batch size as the computation-to-launch ratio improves, and Section 9 confirms balanced stream utilization under our static pre-assignment, conversely, on sequential workloads.

2.5 Scheduling Mechanism

Static graph-based DL frameworks [1–3] use a dataflow model to describe tasks and schedule operators. As depicted in Figure 2(a), each node can represent either a computation or a data transfer operation, and the edges between nodes point from producers to consumers, representing

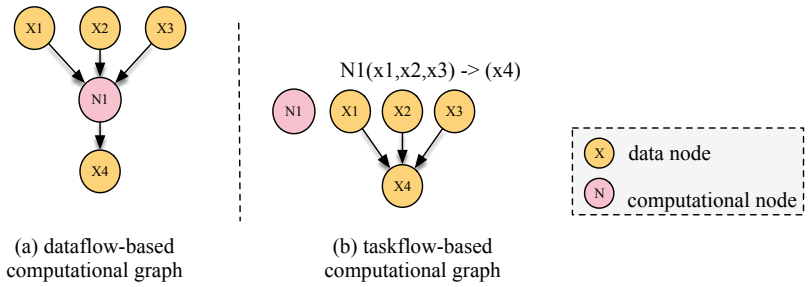


Fig. 2. Dataflow-based computational graph (a) and taskflow-based computational graph (b). The primary distinction between the two lies in whether to consider data as nodes in the computational graph. Taskflow-based computational graph does not require data nodes to participate in the topology, but only needs to be stored in a designated space and accessed by computational nodes.

the direction of dataflow. In this model, operators are dispatched according to a predetermined topological order, which limits exploiting parallelism. Dataflow-based methods are not adept at utilizing parallel capabilities because they predetermine the execution order in a sequential format. The executors will not be able to optimize this execution order for parallelism.

Taskflow-based methods utilize computational tasks as the fundamental scheduling unit, delegating data management to the user to attain high parallel computing performance. As illustrated in Figure 2(b), the taskflow-based computational graph does not necessitate data nodes as connections during construction, but solely links computational nodes. Inputs and outputs are transmitted through parameters, and nodes can only be computed when all dependent nodes are completed. Consequently, the taskflow-based approach supports dynamic scheduling and can achieve enhanced parallelism during the scheduling process.

2.6 Inter-operator Parallelism Works

While several works, including Nimble [23] and Rammer [10], have explored inter-operator parallelism, they represent fundamentally different approaches from PPD. These works employ operator-level dynamic scheduling with runtime stream assignment, while PPD uses subgraph-level static scheduling with compile-time optimization. Nimble focuses specifically on CNN acceleration within PyTorch, while Rammer employs wavefront scheduling for fine-grained operator parallelism. In contrast, PPD emphasizes hardware-aware static partitioning with portable code generation across diverse architectures, including modern transformers and diffusion models. These pioneering efforts, while foundational to the field, reflect earlier explorations in inter-operator parallelism, with development activity having ceased several years ago. This highlights the evolving landscape of DL system optimization and the need for contemporary approaches that address current hardware capabilities and the computational demands of modern large-scale models.

3 Motivation

In this section, we will delineate and summarize the limitations of current parallel optimization methods. Concurrently, we will introduce insights and designs that have inspired PPD for achieving high-performance inference of neural networks.

Existing DL compilers are capable of generating high-performance operator implementations, yet they overlook the runtime parallelism. The operators compiled by common DL compilers (such as TVM [5], Hidet [14], and OpenAI Triton [13]) are organized together through topological order, but the front-end runtime platforms (such as PyTorch [2], Tensorflow [1], and ONNX-Runtime [24])

still adopt the sequential execution order by default. The advantages of parallel structures of "wider" networks (e.g., Inception-V3 [6], HRNet [7], and NASNet [8]) cannot be fully utilized by devices.

The current performance gains of operator compiling mainly come from its memory reuse to improve computing resource utilization [11]. These methods do not intend to enable kernels in networks to run in parallel if certain kernels are not permitted to be fused together. The inference and training processes of models continue to operate within the default stream, despite the widespread adoption of the CUDA stream concept. Rammer [10] elucidates the cause of lower performance when utilizing multiple CUDA streams, highlighting the oversight of synchronization overhead. Consequently, deploying these operators directly on multiple streams leads to performance degradation.

Inference frameworks are general [1, 15, 25], but usually have very large dependent projects, which hinder the deployment of model reasoning services. In addition, for hardware environments with limited resources, such as in-vehicle vision systems [26–28] and auxiliary medical care [29–31], there exist optimization solutions for model computation and storage [32–36]. However, the additional resource consumption associated with the runtime represents a significant overhead.

From the above inquiries, it is clear that further exploration of parallel optimization in neural network runtimes is valuable.

CUDA programs can achieve two types of parallelism [37]: kernel-level parallelism and grid-level parallelism. Kernel-level parallelism is commonly used in CUDA programming, allowing the same kernel to run in parallel on a single device by dividing blocks and threads. However, kernel-level parallelism still faces a synchronous busy waiting problem when independent kernels seek to run simultaneously to maximize device utilization. In such cases, grid-level parallelism becomes essential. Grid-level parallelism enables multiple cores to run simultaneously on the same device using CUDA streams. In DL scenarios, where there are numerous independent kernel running requirements, leveraging grid-level parallelism could potentially lead to performance improvements.

Many works have demonstrated that using grid-level parallelism can significantly improve GPU utilization in high-performance computing field [19, 21, 38]. Compared with kernel-level parallelism, it is more suitable for handling complex neural network workload. Exploring how to fully utilize multiple CUDA streams, minimize synchronization times resulting from kernel data dependencies, and enhance GPU utilization is a topic worthy of study.

To address these limitations, we propose partitioning small operator kernels into balanced subgraphs and deploying them on dedicated CUDA streams, effectively reducing launch overhead while enabling inter-operator parallelism.

In short, we summarize the observations on current parallel optimizations:

— The majority of neural network inference frameworks are built on a "dataflow" model rather than a "taskflow" model.
— The inference frameworks used in the industry currently do not include the estimation of each operator's performance in the computational graph, which hinders efficient parallel processing.
— The current research has not adequately optimized inter-operator parallelism and has failed to fully harness the grid-level parallelism of the GPU.
— For performance-constrained inference scenarios, there is a lack of resource optimization for runtime environment.

4 System Design

We introduce PPD, a portable and highly parallelized DL dispatching system designed to fully exploit the parallel processing capabilities of individual GPU devices and effectively enhance hardware

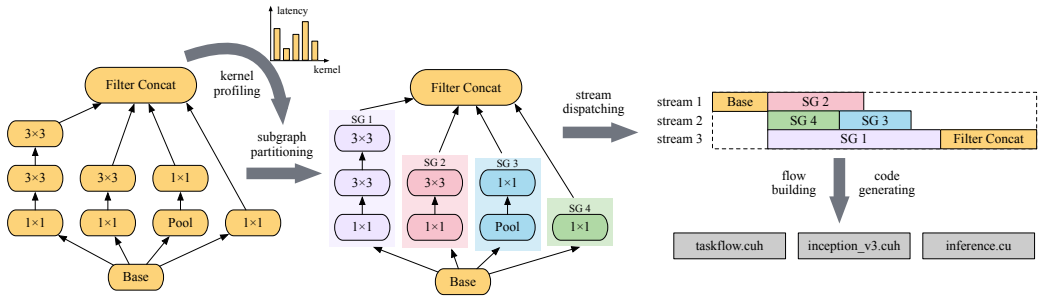


Fig. 3. The PPD pipeline takes the neural network model link relationships, parameters, and operator implementations. Take Inception-V3 as an example. The operator definitions undergo kernel profiling to obtain latency statistics. They are then partitioned into subgraphs (SG) based on the statistics. The subgraphs are assigned with different CUDA stream indexes in stream dispatching process and subsequently transformed into code through flow building and code generation, allowing them to be migrated to any CUDA-supported environment.

utilization. Besides, PPD supports deployment to any CUDA-compatible environment with minimal integration efforts, achieving significant performance improvements (1.36× → 2.28× speedup) with low runtime overhead (≤2.3%).

4.1 Overview

PPD employs a taskflow-based approach to partition the computational graph into balanced subgraphs that minimize data dependencies. Each subgraph is assigned to dedicated CUDA streams for concurrent execution, with a breadth-first order dispatching algorithm that pre-assigns stream IDs at compile time. PPD includes a code generator that produces portable C++ code with minimal dependencies, enabling deployment across different CUDA-compatible environments.

The overall pipeline is shown in Figure 3 and the workflow of PPD is as follows: Firstly, PPD utilizes the high-performance neural network operator kernel implementations and topology information as inputs. The Kernel Parser then fuses and compiles the meta information for kernels, which is subsequently parsed to obtain information such as kernel topology, data types of inputs and outputs, sizes, and device types. This information is then passed to downstream modules. The spatial distribution of CUDA streams on GPU hardware is managed by the GPU's internal scheduler, not by PPD. PPD assigns logical streams to help the hardware scheduler find parallelism opportunities. Whether different GPU units (like CUDA cores and Tensor cores) are used concurrently depends on the operators' characteristics and the GPU's resource allocation policies.

The Kernel Profiler conducts a mini-benchmark for each operator kernel, collecting the individual runtime of each kernel. The results of the mini-benchmark are then provided to the Subgraph Partitioner.

The Subgraph Partitioner primarily implements a partitioning algorithm that simultaneously considers load balancing among subgraphs and minimizes its out-degree. Meanwhile, the Stream Dispatcher provides a CUDA stream dispatch algorithm that maximizes parallelism by preallocating subgraphs to multiple CUDA streams.

Finally, the Flow Builder serves as a code generator module, integrating the partitioned graph, CUDA stream dispatch, and other information to construct C++ source codes into a taskflow-based computational graph.

4.2 Kernel Parser

We utilize the Kernel Parser to extract the implementation and description information of the fused kernels. PPD establishes the topological relationships of operators and uses computational dependencies in scheduling. This allows the execution engine to identify parallelizable nodes and execute them concurrently.

4.3 Kernel Profiler

The Kernel Profiler first acquires the implementation of kernels and conducts a mini-benchmark for each kernel. Calling overhead in Python can be significant, especially for kernels with very short runtimes on the device. Instead of designing Python interfaces to benchmark each kernel, we developed the Kernel Profiler as a code generator module that directly produces C++ source code for mini-benchmarks.

The mini-benchmarking of each operator is divided into three steps: (a) Allocating memory and faking input; (b) Invoking the kernel; (c) Deallocating memory. These three steps will be repeated for 100 epochs, recording only the time consumed by step (b) each time. Memory will be allocated and deallocated every epoch to mitigate the influence of cache on the results. Ultimately, the final result will be computed by averaging the results from each epoch.

4.4 Subgraph Partitioner

Scheduling individual kernels as the smallest scheduling unit for multiple CUDA streams can significantly increase the frequency and duration of kernel launches [10]. Most deep models are decomposed by the DL compiler into many operators more than the original definitions. Therefore, directly scheduling the compiled CUDA kernel onto multiple CUDA streams is detrimental to performance. One possible solution for problems with very short kernel execution times is to combine multiple kernels into a single large kernel (a.k.a. a subgraph) and execute on a single CUDA stream. This approach allows launching kernels together onto the GPU using the CUDA graph, thereby avoiding the latency from frequent launches.

When scheduling kernels onto CUDA streams, it is crucial to consider load balancing across streams. In Kernel Profiling, we conduct mini-benchmarking for each kernel to estimate its runtime. These statistics help estimate the runtime of each subgraph. Uneven CUDA stream workloads cause early subgraph completion and idle resources; balancing workloads is essential for optimal utilization[39, 40]. We, therefore, aim for subgraphs with similar computational time.

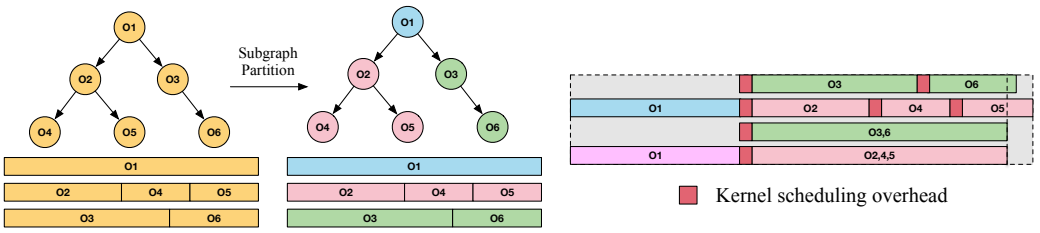
We employ the Subgraph Partitioner to partition the computational graph into balanced subgraphs using a topological sorting-based algorithm (Algorithm 1). The algorithm selects master nodes based on depth and predecessor count, then expands clusters by prioritizing nodes that minimize inter-cluster dependencies. We use mini-benchmarking results to evaluate cluster workload and halt construction when a predetermined threshold is exceeded, ensuring balanced subgraphs while maintaining the acyclic property for task execution.

Microbenchmarking procedure: Prior to code generation, we run a short on-device calibration on the target GPU to time individual operators and representative subgraphs using CUDA events (cudaEventRecord/Synchronize and cudaEventElapsedTime) under the target batch size (batch = 1). We perform a few warm-up runs followed by multiple timed iterations and take the median to obtain stable measurements. These measurements populate a device-specific cost model that (i) estimates subgraph runtime by aggregating constituent operator costs with a simple overlap model along disjoint paths and (ii) guides the selection of the number of CUDA streams for load balancing. This calibration is a one-time offline step and incurs negligible runtime overhead.

Figure 4(a) illustrates a computational graph comprising 6 kernels with varying time consumption. Utilizing this computational graph directly necessitates 6 kernel launches, and the current data

ALGORITHM 1: Pseudo Code for Subgraph Partitioner. M is the approximately maximum number of nodes contained in each cluster.

```
Input: G = (V, E), M
Result: cluster_list
ready = get_input_nodes(G);
depths = longest_distance_from_input(G, ready);
workload = get_workload_from_benchmark(G);
workload_threshold = sum(workload) / |V| * M;
in_deg = get_in_deg(G);
while ready != ∅ do
    ready.sort();
    master = ready.pop;
    current_cluster = master;
    current_boundary = ∅ ;
    for u ∈ master.successors do
        in_deg[u] - -;
        if in_deg[u] == 0 then
            ready.insert(u);
        else
            current_boundary.insert(u);
        end
    end
    while current_cluster.workload < workload_threshold do
        ready.sort();
        next= ready[0];
        if check_if_empty() then
            break;
        end
        next = ready.pop;
        current_cluster = current_cluster ∪ next;
        for u ∈ next.successors do
            in_deg[u] - -;
            if in_deg[u] == 0 then
                ready.insert(u);
                current_boundary.erase(u);
            else
                current_boundary.insert(u);
            end
        end
    end
    cluster_list.append(current_cluster);
end
```



(a) Independent kernel runtime (b) Subgraph runtime (c) Pipeline of two different runtime strategies

Fig. 4. (a) The operator link relationships and the time consumption of each operator in the original computational graph. (b) The computational graph after subgraph partition, with different colors indicating different subgraphs. (c) The kernel scheduling overhead of two different runtime strategies.

dependencies will result in 5 synchronizations (as denoted by the bold black arrows). Figure 4(b) depicts the computational graph after undergoing processing by the subgraph partition algorithm, which partitions the kernels according to the algorithm's process. The algorithm divides the six kernels into three subgraphs, ensuring their execution times are equal and minimizing the synchronization and launching frequency as much as possible (requiring only two synchronizations). As shown in Figure 4(c), subgraph partitioning results in a reduction of three instances of unnecessary kernel launching and synchronization, thereby reducing the overhead of kernel scheduling.

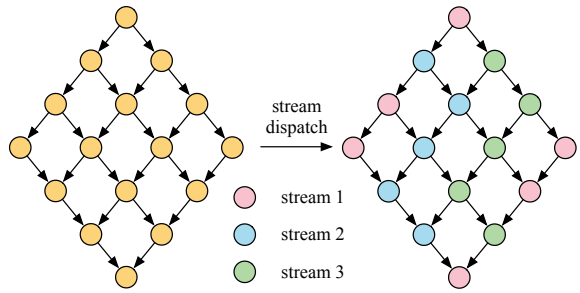
It is important to note that each subgraph will be scheduled as a CUDA graph at runtime. As the scheduling algorithm used by CUDA graph is closed-source, we are unable to accurately predict the exact runtime of each CUDA graph based on the underlying scheduling strategy. Therefore, we adopt a practical approach by aggregating the micro-benchmark results of all operators within a subgraph to estimate its runtime, which yields few end-to-end measurements across our evaluated models, providing adequate fidelity for load balancing despite uncertainty in CUDA's internal scheduling mechanisms.

4.5 Stream Dispatcher

We partition the computational graph into multiple subgraphs and minimize the connections between them to ensure improved subgraph-level parallelism in subsequent stages using the Subgraph Partitioner. Furthermore, to achieve load balancing, the Subgraph Partitioner utilizes the results of mini-benchmarks as weights to further refine the partitioning of subgraphs based on breadth-first search order. In the Stream Dispatcher, our aim is to leverage CUDA streams to achieve grid-level parallelism and further harness the parallelism capabilities of the devices.

The Stream Dispatcher operates by pre-allocating a CUDA stream to each subgraph that has been partitioned by the Subgraph Partitioner. When a subgraph becomes executable from the topology constraints, it will be launched directly on the specified stream. These subgraphs are constructed as cudaflows during the codegen phase, which essentially serves as a wrapper around CUDA Graphs proposed by Lin et al. [41].

The algorithm takes a **computational directed acyclic graph** (**CDAG**) as input, where each node in the DAG represents a **directed acyclic subgraph** (**DAS**) of one or multiple kernels that have been processed by the Subgraph Partitioner. The output of the algorithm is a mapping relationship that specifies which subgraph should be assigned to which CUDA stream to execute. The goal of the algorithm is to maximize the parallelism of kernel execution, i.e., to maximize the parallelism of a single device. The details of the algorithm are described in Algorithm 2 with an illustrative example presented in Figure 5.



(a) Before stream dispatch (b) After stream dispatch

Fig. 5. Illustration of stream dispatch algorithm. (a) the computational graph before stream dispatch; (b) the computational graph after stream dispatch, where the different colors of the nodes indicate that they have been assigned to different streams.

ALGORITHM 2: Pseudo Code for Stream Dispatcher. S stands for the number of CUDA streams.

```
   Input: G = (V, E), S
   Result: stream_index_dict
1  stream_index_dict = ∅;
2  assigned_stream_index = 0;
3  root = get_root_node_index(G);
4  queue = empty_queue_with_init_node(root);
5  while queue != ∅ do
6      u = queue.pop_front();
7      for v ∈ G[u].successors do
8          if v ∉ stream_index_dict then
9              stream_index_dict[v] = assigned_stream_index % S;
10             assigned_stream_index ++;
11             queue.push(v);
12         end
13     end
14 end
```

An alternative approach involves identifying the critical path in the entire computational graph [23] and assigning the nodes on the critical path to the same CUDA stream. While this approach can reduce the potential increase in memory footprint across CUDA streams, it may lead to some loss of load balancing capability. In our algorithm design, we opted not to adopt this approach to achieve optimal load balancing at a high level, as we have already endeavored to unify the computational complexity of the subgraph during subgraph partitioning.

The dispatch algorithm is structured on a breadth-first search order to optimize CUDA stream assignment, maximizing parallelism. This is because subgraphs higher in the topological order, having fewer dependencies, are more likely to be assigned to earlier streams. It allows subgraphs to be launched in a stream as early as possible based on their dependencies, enhancing parallel execution. Our dispatching algorithm implicitly address data synchronization issues within the

algorithm itself. Synchronization refers to the necessity for CUDA streams to wait for all previous threads to complete before executing the next set of CUDA kernels, ensuring data consistency.

4.6 Flow Builder

In order to create a neural network inference framework that can be effectively utilized, we have developed the Flow Builder. As previously mentioned, to fully harness the parallel capabilities of the operator, we utilize two algorithms—subgraph partition and stream dispatch—in addition to requiring high-performance implementation of neural network operators. It's important to emphasize that these tasks are all carried out prior to model inference, and an efficient inference framework is also crucial.

Our design requirements for the inference framework include transitioning away from the dataflow-based to taskflow-based computational graph execution mode and implementing high-performance CUDA kernels and C++ callings.

Taskflow-based Computational Graph. To avoid reinventing the wheel and leverage the computational abilities of both CPUs and GPUs, we used Taskflow [42], a parallel computing framework to construct our inference programs. Taskflow proposed the **Task Dependency Graph** (**TDG**) programming model, which abstracts GPU tasks into a DAG for optimized scheduling. Taskflow has developed a CPU-centric scheduler and integrated several advanced and high-performance scheduling algorithms [43, 44]. As a result, Taskflow has become an established industrial standard for parallel computing frameworks.

Cudaflow [41], a sub-project of Taskflow, allows users to consolidate multiple kernel operations into a single Cudaflow. The Cudaflow implementation acts as a wrapper for CUDA graph, which significantly reduces the frequency of kernel launches and enhances GPU performance by fusing kernels. As a result, post-compiler fusion operators can still be completed quickly. Consolidating multiple kernels into a CUDA graph for execution can save time. Furthermore, when the model involves CPU operations, using CUDA graph can notably reduce overall time consumption. This is because, in scenarios where a sequence of operations requires both CPUs and GPUs, the use of multiple CUDA streams enables concurrent execution of different operations that utilize different computing resources.

Cudaflow achieves performance on par with CUDA Graph while enabling users to implement this functionality in a highly concise manner. Integrating multiple kernels into Cudaflow provides the following benefits:

— Reduce the time overhead of frequent CUDA kernel launches and to enable CUDA Graph launch to the specified CUDA stream.
— Taskflow can schedule Cudaflow as a subflow, converting kernel synchronization to CPU task scheduling, leveraging the CPU's scheduling algorithms.

As a pure parallel computing framework, Taskflow does not natively support neural network training and inference capabilities.

Code Generator. The flow builder incorporates a code generator module to create an executable taskflow. In Figure 6, we present the code generator process and demonstrate the resulting generated taskflow code snippet. We provide a simple example to illustrate the pipeline, which includes three cudaflows as subflows. The 'Definitions' section establishes fundamental related parameters, encompassing the definitions of TaskFlow, CudaFlow, and CudaStream wrappers

We use cudaflow to model each subgraph and delegate the management of dependency relationships to Taskflow, thus eliminating the need for synchronization operations. The generated code

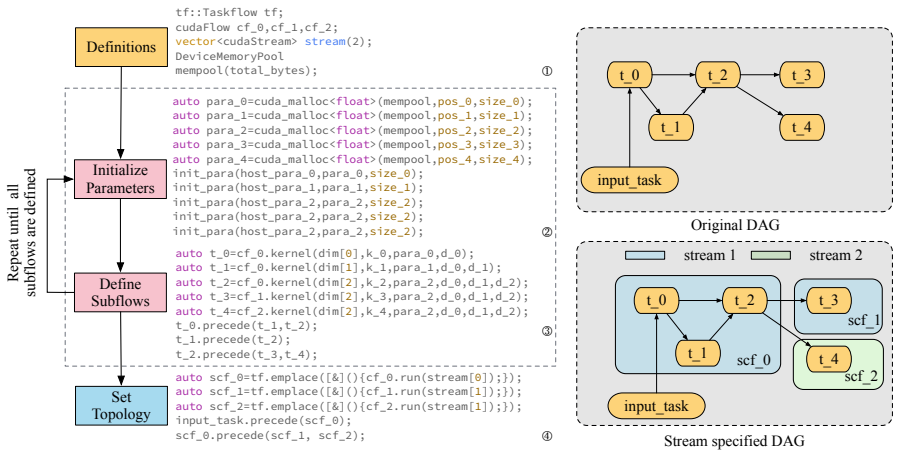


Fig. 6. Details of the code generator process and the resulting code of the flow builder. The generated code serves four primary functions: establishing basic definitions, initializing parameters, defining subflows, and outlining subflow topology. The illustration below provides a visual representation of the generated code.

Table 1. Baseline Versions

Method	TF-XLA	PyTorch	TVM	Hidet
Version	2.15.0	2.1.2	0.10.0	0.3.1.dev
Legacy (native)	Framework Stack			
IOS	Legacy stack; PyTorch 1.6.0			
Nimble	Legacy stack; PyTorch 1.7.1			

depicted in Figure 6 illustrates the Taskflow (a simulated input task precedes the subflow 0 (scf_0)), which represents the task flow constructed by the code presented above.

The flow builder generates code that includes: ① a set of definitions for cudaflows as subflows and a taskflow. The taskflow functions as a module for scheduling other subflows and operates on the CPU. Subsequently, the flow builder generates initialization functionalities for each subflow. We provide a detailed explanation of how one of the cudaflows is initialized and topologically linked through the generated code. The partitioning of subflows and their internal initialization information are derived from the subgraph partitioner. In ②, the parameter memory allocation and initialization for the subflow are generated initially. Through the design of succinct memory allocation, we pre-allocate all the parameters required by the kernels in a continuous memory segment. These parameters may consist of trained parameters or the outputs of previous kernels. Subsequently, in ③, the kernel implementations are assigned to the task nodes of the cudaflow. Following this, the inter-subgraph link relationships are generated. This part of the link relationships also originates from the subgraph partitioner. In ④, there is a process of dispatching CUDA stream IDs, with the IDs being assigned from the stream dispatcher.

5 Evaluation

In this section, we provide detailed evaluation results and compare them with other state-of-the-art frameworks to demonstrate the advancement and superior inference performance of PPD.

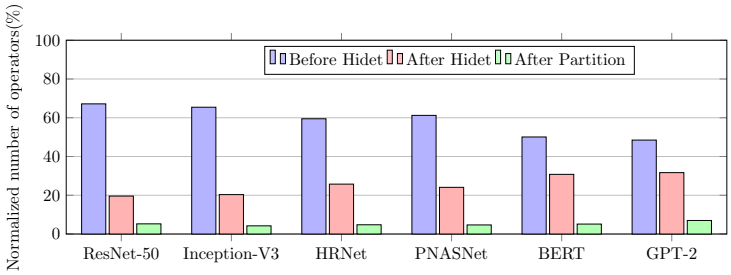


Fig. 7. We conducted a statistical analysis of the number of scheduling units at different compilation stages and showed the normalized number of scheduling units for six models at three different stages: (1) the blue bars represent the number of operators in the original computational graph (serving as the 100% baseline); (2) the orange bars represent the number of operators after optimization (such as operator fusion) by Hidet [15]; (3) the green bars represent the number of subgraphs after PPD partitioning. The number of scheduling units directly affects the frequency and overhead of multi-CUDA stream scheduling.

5.1 Experimental Setup

We used the C++17 standard and enabled O3 optimization when compiling the benchmarks and CUDA kernels produced by PPD. The versions of baseline methods are shown in Table 1. Note that PPD's performance gains are orthogonal to PyTorch's optimizations: while PyTorch focuses on high-level operator fusion and graph-level optimizations, PPD addresses lower-level parallelization opportunities at the CUDA stream level through custom kernel execution and stream management, making our benefits complementary to PyTorch's latest optimizations.

Table 1 also summarizes IOS and Nimble under their latest upstream open-source native stacks. These two are reported as reference (non-parity) due to framework/runtime differences; their latencies appear in Table 2 and are labeled accordingly. A slash (/) denotes unsupported models on a given stack.

5.2 Workloads

Our evaluation includes diverse models (ResNet-50, Inception-V3, HRNet, PNASNet, BERT, and GPT-2-Large) to provide comprehensive coverage of different architectural paradigms. We include transformer-based models where PPD may not show advantages, ensuring unbiased assessment. For multi-stream effectiveness experiments, we focus on CNN models with significant inter-operator parallelism.

We focus our comparison on production-ready frameworks: PyTorch Inductor [12], Hidet [14], TensorFlow-XLA [45], and TVM [5], representing current state-of-the-art optimization approaches.

As illustrated in Figure 7, PPD significantly reduces scheduling units by organizing fine-grained operators into balanced subgraphs, lowering overhead and enabling efficient parallel execution.

5.3 Inference Overhead

We conducted a comprehensive performance evaluation across all models mentioned in Section 5.2. Note that we have selected the best performance results of PPD (optimized across different stream numbers, also with subgraph partitioning) in the specified model as the conclusion in the table. The complete parameter sweep results and optimization details are provided in Section 5.7. In all baselines, we employed CUDA Graph, with the exception of TF-XLA, as it does not fully support CUDA Graph. We further note that PPD's multi-stream scheduling operates at the CUDA

Table 2. Inference Latency Benchmark Results(ms)

Model	Input Shape	TF-XLA	PyTorch	TVM	Hidet	IOS	Nimble	PPD(ours)
ResNet-50	[1,3,299,299]	7.68	6.23	6.68	5.47	13.26	11.14	2.63
Inception-V3	[1,3,299,299]	24.63	10.14	14.31	7.86	20.87	10.22	5.79
HRNet-W18-V1	[1,3,128,768]	42.84	48.69	44.83	21.62	50.63	32.98	9.50
PNASNet-5-Large	[1,3,331,331]	/	45.99	47.49	44.58	55.59	46.94	26.99
BERT	[1,512]	24.78	31.53	28.65	37.72	/	/	24.93
GPT-2-Large	[1,32]	734.89	783.44	768.85	761.67	/	/	732.36

Columns "Nimble [23]" and "IOS [46]" are measured under their latest upstream native stacks and reported as non-parity reference; "/" indicates unsupported.

stream level and is orthogonal to framework-level fusion/compilation optimizations; therefore, improvements are complementary to advances in PyTorch or other frameworks.

5.4 Overall Performance Analysis

As shown in Table 2, this evaluation includes CNN models that are favorable to parallel decomposition, transformer models (BERT and GPT-2-Large) that present varying challenges for our approach, ensuring an unbiased assessment of PPD's capabilities and limitations across different architectural paradigms. Our method achieved the best or near-best performance on most baseline models, indicating that our framework significantly accelerates model inference compared to previous works. Using the best-performing baseline for each model, we achieved speedups ranging from 1.36× to 2.28× across CNN models. For transformer architectures, the results demonstrate architecture-dependent performance: while BERT shows minimal improvement due to its sequential nature (PPD: 24.93 ms vs. TF-XLA: 24.78 ms), GPT-2-Large is comparable, slightly faster with PPD than the best baseline (732.36 ms vs. 734.89 ms from TF-XLA). This difference in transformer performance can be attributed to GPT-2-Large's larger computational workload providing more opportunities for effective parallel decomposition, while BERT's smaller size and highly optimized implementations in frameworks like TF-XLA limit the benefits of multi-stream scheduling. These results confirm that PPD's effectiveness scales with model complexity and computational intensity. For ResNet-50 specifically, the observed gains primarily come from (i) reducing kernel-launch and synchronization overhead by grouping many small operators into subgraphs and issuing them as CUDA Graphs, and (ii) overlapping sparse branch-level concurrency in dimension-changing residual blocks, where the projection shortcut (1×1) convolution runs in parallel with epilogue/prologue stages in the main path. We also report IOS and Nimble as reference columns measured under their latest upstream native stacks (non-parity). On CNNs, their latencies are generally higher than Hidet and PPD—reflecting older operator implementations and runtime overhead—while for transformer models they are unsupported ("/"). These reference numbers provide additional context but are not strict apples-to-apples with production-ready stacks.

Moreover, an additional stream can enable three-way overlap across the downsampling branch, the tail of the main path (BatchNorm/ReLU/add), and the prologue of the subsequent block. While this mechanism can make sn = 3 outperform sn = 2 in some settings, the end-to-end optimum depends on batch size, hardware, and the resulting partitioning. In our ResNet-50 setting with batch size 16, Table 3 shows that sn = 2 is slightly better, which is consistent with diminishing returns beyond sn = 2–3 due to reduced exploitable parallelism and stream contention.

Due to the varying degrees of parallelism in the computational graphs obtained after compilation for different models, we observed different speedup ratios across various models. PNASNet-5-Large is not available in TF-XLA due to the lack of implementation. The performance of PPD

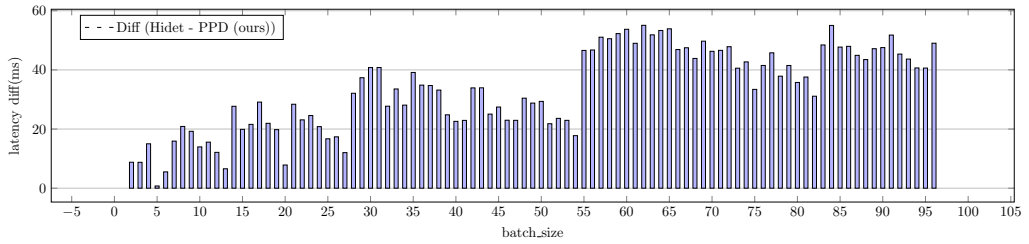


Fig. 8. Difference of inference latency between Hidet and PPD (ours) on HRNet, the batch size increases from 1 to 96.

Table 3. Inference Latency(ms) with Batch Size = 16

Model	Hidet	PPD(sn = 1)	PPD(sn = 2)	PPD(sn = 3)	PPD(sn = 4)
Inception-V3	84.86	81.54	79.29	82.4869	**80.89**
HRNet	180.33	172.00	**162.12**	167.89	172.02
PNASNet-5-Large	592.136	606.80	583.53	**563.81**	608.36
ResNet-50	56.10	52.04	**51.99**	52.44	52.00

is slightly inferior to TF-XLA in BERT, attributable to TF-XLA's higher-performance operator implementations.

5.5 Larger Batch Sizes

To evaluate multi-stream scheduling effectiveness, we focus on CNN models that demonstrate significant inter-operator parallelism, excluding transformer models due to their predominantly sequential computation patterns. We use batch size 16 as it represents a commonly used configuration that effectively utilizes GPU parallelism while maintaining reasonable memory consumption. We compared our method with Hidet as our most efficient baseline, with results shown in Table 3.

The abbreviation "sn" stands for stream numbers, indicating the number of CUDA streams used in our method. For experimental control, the maximum operator number threshold for each subgraph is set to 20.

As demonstrated in Table 3 and Figure 8, PPD consistently outperforms Hidet under large batch conditions (batch size ≥ 16), with performance advantages becoming increasingly pronounced as batch size increases. This confirms that PPD is particularly effective in large batch inference scenarios through full utilization of GPU's parallel capabilities.

To further demonstrate the effectiveness of PPD, HRNet is selected as the benchmark. The batch size was increased from 1 to 96 to observe the time disparity between PPD and Hidet. The result is shown in Figure 8. We obtained an average improvement of **32.8 ms** in our experiments with batch size ranging from 1 to 96.

As shown in Figure 8, PPD exhibits limited performance advantages when the batch size is small due to two factors: (1) insufficient computational density to fully exploit parallel processing capabilities, and (2) scheduling overhead representing a larger proportion of total computation time, reducing net performance gains. This batch-size trend corroborates our subgraph-level design: higher compute-to-launch ratios amplify the benefit of CUDA Graph launches and provide larger windows for inter-operator overlap under static stream assignment.

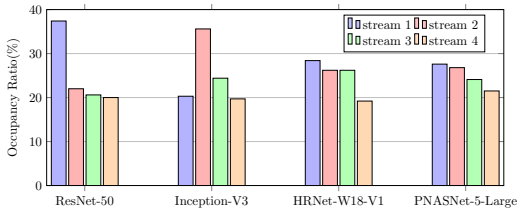


Fig. 9. Occupancy ratios of four CUDA streams across various models.

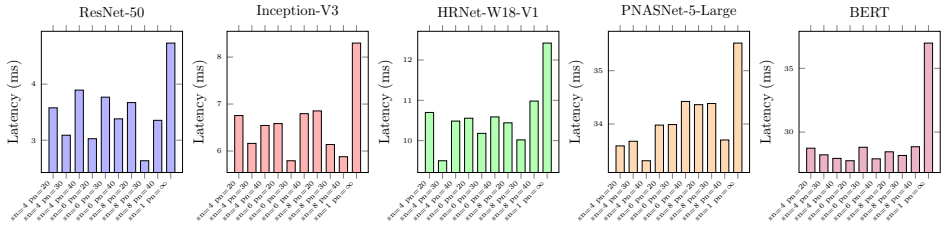


Fig. 10. Latency of different stream numbers (sn) and different numbers of nodes in subgraphs (pn).

5.6 Load Balancing on CUDA Streams

To verify that the stream dispatch algorithm can evenly distribute kernels to CUDA streams, we used the NVIDIA Nsight Systems tool to statistically analyze the usage ratio of each CUDA stream for four models when the stream number is set to 4 (we fixed each subgraph to contain no more than 20 original kernels in the experiment).

We set the batch size to 16 for each model to ensure more accurate statistics. This batch size provides sufficient computational workload to generate enough kernel execution events for reliable CUDA stream utilization measurement. The larger batch size also ensures stable GPU utilization patterns, reducing measurement variance that could occur with smaller batches due to insufficient computational density.

Figure 9 shows our statistical results, and through the stream dispatch algorithm, we have achieved a balanced workload distribution. It is worth noting that there is an uneven distribution of CUDA workload in ResNet-50 and Inception-V3 (with a difference of 17.4% and 15.9% between the maximum and minimum ratios, respectively). This is because the parallelism of the computational graph obtained after subgraph partitioning for the above two network structures is not high, resulting in long-tail nodes in the computation. In the results of HRNet and PNASNet, which are more complex structures, such computational workload has a relatively small impact on load balancing. Overall, the near-flat ratios validate our calibration-driven subgraph partitioning and static stream pre-assignment.

Note that BERT and GPT-2 are excluded from this batch size analysis to clearly demonstrate multi-stream parallelization benefits. Their sequential computation characteristics would obscure the parallel execution advantages that this experiment aims to highlight.

5.7 Ablation Studies

In this section, we present experimental conclusions that showcase the effectiveness of the proposed methods through the control of parameter configuration and functionality.

In Figure 10, we initially analyzed the performance of different numbers of nodes (pn) in multiple CUDA streams within different subgraphs. The figure illustrates the inference time when pn = 20,

Table 4. Latency of Various Models Under Different Stream Numbers (sn)

Model Input Shape	ResNet-50 [1,3,299,299]	Inception-V3 [1,3,299,299]	HRNet-W18-V1 [1,3,128,768]	PNASNet-5-Large [1,3,331,331]	BERT [1,512]
sn = 1 w/ sp	4.7243	7.9743	18.9922	27.9159	37.1384
sn = 2 w/ sp	4.3555	**5.7895**	12.7522	**26.9894**	27.7226
sn = 3 w/ sp	**2.6331**	5.9763	11.3431	28.5906	**24.9347**
sn = 4 w/ sp	3.5968	6.073	**9.4991**	29.3287	28.4919
sn = 5 w/ sp	3.7818	6.3404	10.431	29.7535	28.172
sn = 6 w/ sp	3.6149	6.3457	10.461	29.9947	28.2319
sn = 7 w/ sp	3.5505	6.4354	10.7548	29.8218	27.6618
sn = 8 w/ sp	3.6911	6.6365	10.8446	29.8773	27.5596
sn = 1 w/o sp	4.8026	8.0309	21.0204	44.6492	37.0626
sn = 2 w/o sp	4.0821	5.92	13.4099	31.3191	30.2379
sn = 3 w/o sp	3.4105	6.0032	12.764	29.0829	28.6554
sn = 4 w/o sp	3.6456	6.0545	12.2473	29.3825	28.3376
sn = 5 w/o sp	3.623	6.3546	12.3026	30.3767	27.7931
sn = 6 w/o sp	3.6591	6.2141	12.2203	31.6037	28.8222
sn = 7 w/o sp	3.6716	6.5608	12.3347	32.6117	27.5544
sn = 8 w/o sp	3.6597	6.6264	12.7061	29.6521	27.9695

30, and 40, as well as sn = 4, 6, and 8. Additionally, we included the performance statistics for the case of only one subgraph and one CUDA stream (pn = ∞, sn = 1). It demonstrates that there are varying performance results under different configurations of sn and pn, but when the number of subgraphs is 1, the performance is weaker than in other cases. This phenomenon validates that using multiple streams for computation can enhance performance.

To further validate the effectiveness of subgraph partitioning, we conducted an ablation experiment specifically targeting the partitioning algorithm. Table 4 shows the latency of each model under different stream numbers (sn) and whether the subgraph partition algorithm is used. For this comparison, we use batch size 1 to evaluate the performance in real-time inference scenarios, which are critical for edge computing and latency-sensitive applications. Single-batch inference represents a common deployment requirement where immediate response is prioritized.

In Table 4, "w/ or w/o sp" indicates whether the result is obtained using the subgraph partition algorithm or not. This is equivalent to setting the operator count of each subgraph to 1. For experiments using subgraph partitioning, each subgraph contains no more than 10 nodes.

Notably, Table 4 reveals that BERT demonstrates mixed results where no partitioning (w/o sp) outperforms partitioning (w/ sp) in 4 out of 8 stream configurations. This phenomenon reflects the architecture-dependent nature of subgraph partitioning effectiveness. BERT's transformer architecture with self-attention mechanisms creates predominantly sequential computation flows that limit parallel decomposition opportunities. When parallelization benefits are minimal, the scheduling and synchronization overhead of subgraph partitioning can outweigh the advantages, resulting in performance degradation. This analysis demonstrates that while PPD excels with CNN models featuring significant inter-operator parallelism, sequential architectures like transformers may require different optimization strategies.

The most noticeable observation is that when subgraph partition is not utilized and the number of CUDA streams is 1, the time consumption is typically the longest. This is attributed to the frequent launching of CUDA kernels, aligning with Rammer's experimental findings [10]. Our analysis indicates a smaller inference latency when employing the subgraph partition algorithm.

Table 5. Overhead of PPD

Model	System Overhead
Inception-V3	1.8%
HRNet	1.1%
PNASNet-5-Large	$\leq 1\%$
ResNet-50	2.3%
BERT	1.6%
GPT-2-Large	$\leq 1\%$

This reduction is a result of the partition algorithm significantly decreasing the number of scheduled operators (as depicted in Figure 7), and scheduling the subgraph as a CUDA graph can reduce launch time.

Hardware-wise, multi-stream execution better utilizes GPU's parallel architecture by allowing independent subgraphs to execute concurrently on different SM units and pipelines, whereas single-stream execution creates resource idle periods due to sequential dependencies.

Our partitioner is hardware-aware via a short calibration on the target device. We measure operator/subgraph latency and overlap to build a device-specific cost model, which then drives both subgraph cuts and the number of CUDA streams. As a result, architectural differences—SM count and kernel concurrency limits, memory hierarchy and bandwidth, and scheduling overhead—are naturally reflected in the chosen partition without per-device heuristics. Qualitatively, compute-rich and low-overhead GPUs tend to prefer higher stream counts and coarser-grained partitions, while bandwidth- or concurrency-limited devices benefit from finer-grained cuts and conservative parallelism to avoid contention. This calibration is a one-time offline step and does not add runtime overhead.

The experiment in Table 4 encompasses tests for different numbers of CUDA streams. Our findings reveal that for different models, there exists an optimal number of CUDA streams to achieve the best performance. Generally, using only one single CUDA stream leads to inferior performance compared to multi-stream configurations. This is because different models exhibit varying degrees of inter-operator parallelism after partitioning - models with higher parallelism can effectively utilize more CUDA streams, while models with limited parallelism show diminishing returns as the number of streams increases.

5.8 System Overhead

In this section, we analyze the system overhead introduced by PPD. We use VTune [47] to analyze the additional system overhead apart from libcuda.so and initialization operations. The overhead measurements focus on runtime scheduling overhead and do not include one-time preprocessing stages performed offline.

We conducted experiments using Inception-V3, HRNet, PNASNet-5-Large, ResNet-50, BERT, and GPT-2-Large. We accumulated the proportion of overhead in Table 5. The table shows that the scheduling and other CPU overhead of PPD accounts for at most 3% of the overall time. The difference in system overhead between different models is not significant, as smaller models require less computational overhead, resulting in smaller scheduling costs. Overall, the scheduling performance of PPD has minimal impact on the performance of model inference tasks. This is based solely on the actual CPU computational load and does not account for the asynchronous CPU computations during GPU processing. Therefore, the actual scheduling overhead is even smaller in the overall runtime.

Table 6. Compatibility of PPD in Different Specified Environments

Env	Operating System	CUDA/HIP Version	GPU
♡	Ubuntu 22.04 (WSL[1])	CUDA 11.8	NVIDIA GeForce RTX 3060
♢	Ubuntu 22.04	CUDA 12.1	NVIDIA GeForce RTX 3090
♣	Ubuntu 22.04	CUDA 11.0	Denglin Goldwasser II L256
♠	Ubuntu 22.04	HIP 6.4	AMD Instinct MI210

5.9 Portability

To substantiate the assertion that PPD is lightweight to be expeditiously ported across diverse system environments, we have also empirically verified the portability of PPD through a series of rigorous experiments.

The output of PPD comprises a set of model-specific CUDA kernels' `.cuh` files, the Taskflow header-only library and its representation of the model, as well as a collection of calling methods for reasoning. This ensemble of outputs manifests as a static source code file, tailored to GPUs that support CUDA. Given the heterogeneity of operating systems and runtime library versions employed in disparate environments, the source code necessitates recompilation to adapt to these divergent settings.

Pursuant to the prevailing experimental environment conditions, we have designed four distinct scenarios (as depicted in Table 6) and subsequently migrated the output of PPD pertaining to the aforementioned model to these environments. Our findings underscore the remarkable portability of the PPD framework, which can be seamlessly integrated across a wide range of system configurations with minimal overhead. All the scenarios we designed support CUDA, and most of the GPUs used are accelerators from the NVIDIA GeForce series.

We have also adapted for AMD GPUs. We noticed that HIP proposed by AMD is similar to CUDA, providing a C++ runtime API [48]. Besides, the APIs provided by HIP are almost the same as those provided by CUDA, which makes our migration work easy. We implemented the conversion program from CUDA Stream and CUDA Graph to HIP Stream and HIP Graph, and verified the feasibility of PPD on a server equipped with AMD Instinct MI210.

In particular, we introduced the Denglin Goldwasser II L256 [49], an inference accelerator card produced by the Chinese chip company Denglin Technology. Denglin provides a set of supporting libraries compatible with standard CUDA programming, dlCU, which provides a set of APIs compatible with CUDA Driver and Runtime and a compiler dlcc compatible with CUDA C/C++.

To demonstrate PPD's relevance to modern AI workloads, we include two additional contemporary architectures: Vision Transformer (ViT-Base/16) [52] and UNet [53] since they are representative of current trends in computer vision and generative AI. While the original references for these architectures may be dated, they remain foundational to current AI applications: ViT has become the standard architecture for modern vision transformers, and UNet serves as the core building block for state-of-the-art diffusion models like Stable Diffusion [54] and FLUX.1 [55]. Our evaluation focuses on PPD's ability to handle the computational patterns of these architectures rather than their absolute novelty, demonstrating that our subgraph-level approach is architecture-agnostic and effective for modern AI workloads. Table 7 presents comprehensive performance evaluation across

[1]Windows Subsystem for Linux (WSL).

Table 7. Cross-platform Performance Evaluation of PPD (Inference Latency in ms)

Model	Input Shape	♡ 3060	♢ 3090	♣ L256 [2]	♠ MI210
ResNet-50 [50]	[1,3,299,299]	7.94	2.63	27.58	1.75
Inception-V3 [6]	[1,3,299,299]	17.58	5.79	106.84	2.78
HRNet-V1 [7]	[1,3,128,768]	24.37	9.50	243.74	4.21
PNASNet-Large [9]	[1,3,331,331]	61.64	26.99	681.12	8.19
BERT [51]	[1,512]	54.18	24.93	723.78	7.56
ViT-Base [52]	[1,3,224,224]	51.60	19.86	601.28	5.85
UNet [53]	[1,3,224,224]	8.97	3.01	31.57	3.46

four distinct hardware environments. The results demonstrate PPD's consistent effectiveness across different GPU architectures and software stacks. The ♠ AMD MI210 configuration demonstrates successful cross-vendor portability through our HIP adaptation layer.

We further discuss the cross-platform performance characteristics. The observed gap on Denglin L256 arises primarily from architectural and software-stack differences, including fewer compute resources and lower kernel concurrency limits, as well as a younger toolchain. Likewise, performance differences between devices reflect differences in memory hierarchy/bandwidth and scheduling/-driver overheads. Since PPD's partitioning is driven by on-device calibration, the resulting subgraph cuts and stream counts adapt to each platform accordingly; our focus here is portability and the effectiveness of multi-stream scheduling rather than peak vendor-specific tuning.

6 Conclusion

The existing dataflow-based DL compilers primarily concentrate on optimizing operator performance and intra-operator parallelism. Furthermore, the runtime engine in the inference process partially involves generating a topological sequence before scheduling execution, overlooking the parallelism between operators. Some other approaches employ a single CUDA stream parallel scheduling method, failing to fully exploit the additional parallel optimization offered by multiple CUDA streams and the CUDA Graph's CUDA kernel launch latency optimization. Our proposed PPD is a taskflow-based, portable and highly parallel dispatching system for DL. By partitioning the computational graph into subgraphs to obtain a set of subgraphs with approximate runtime and maximal data independence, we aim to reduce CUDA kernel launch time. PPD's stream dispatch algorithm maximizes parallelism. In experimental validation, PPD achieves a speedup of 1.36× to 2.28× compared to previous techniques on validated models. We also verified the portability and minimal runtime overhead. PPD offers an end-to-end lightweight deployment solution, enabling the generation of CUDA code for all required model operators and the taskflow definition. This allows for seamless integration into any project.

7 Future Works

The limitation of PPD is that it requires users to manually set the stream number and the number of operators in each subgraph. These values may rely on the user's experience or the optimal values obtained from multiple performance benchmarks in a specific hardware environment, or through grid-searching to determine the best parameter combination. To offer users a streamlined model inference solution from PPD, these hyperparameters can be eliminated by introducing an adaptive

[2]The Denglin L256's lower throughput primarily stems from architectural and software-stack factors—fewer compute resources and lower kernel concurrency limits, memory hierarchy/bandwidth differences, and a younger toolchain/driver stack—rather than the host environment. Our cross-platform study focuses on portability and functional correctness, not vendor-specific peak tuning.

solution. We plan to develop a lightweight online controller that jointly tunes the number of CUDA streams and the subgraph size threshold using short feedback loops, thereby removing manual hyperparameters.

On the other hand, Taskflow currently necessitates users to pre-specify a CUDA stream ID for stream dispatch. Alternatively, we can maintain additional stream status statistics in Taskflow and directly schedule tasks to idle CUDA streams. This approach can more comprehensively utilize the mechanism of replacing synchronization operations after the completion of CUDA stream execution with Taskflow task scheduling on the CPU, thereby further enhancing GPU utilization. In parallel, we aim to enrich the calibration-based cost model to explicitly account for cache/bandwidth contention and SM residency/occupancy constraints across concurrent streams, enabling more accurate partitioning and stream selection on diverse hardware.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: A system for {Large-Scale} machine learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, and S. Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems* 32 (2019).

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv:1512.01274. Retrieved from https://arxiv.org/abs/1512.01274

[4] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[5] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, and A. Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. 11 (2018), 20. *arXiv preprint arXiv:1802.04799*.

[6] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826.

[7] Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, et al. 2020. Deep high-resolution representation learning for visual recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43, 10 (2020), 3349–3364.

[8] Xu Qin and Zhilin Wang. 2019. Nasnet: A neuron attention stage-by-stage net for single image deraining. arXiv:1912.03151. Retrieved from https://arxiv.org/abs/1912.03151

[9] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 19–34.

[10] 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 881–897.

[11] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: Application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[12] Peng Wu. 2023. PyTorch 2.0: The journey to bringing compiler technologies to the core of PyTorch (keynote). In *Proceedings of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. 1–1.

[13] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[14] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 370–384.

[15] Team Hidet. 2023. Introducing hidet: A deep learning compiler for efficient model serving. Retrieved from https://pytorch.org/blog/introducing-hidet/

[16] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. CUTLASS. (Jan. 2023). Retrieved from https://github.com/NVIDIA/cutlass

[17] Mohammed Sourouri, Tor Gillberg, Scott B. Baden, and Xing Cai. 2014. Effective multi-GPU communication using multiple CUDA streams and threads. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 981–986.
[18] Paweł Czarnul. 2020. Investigation of parallel data processing using hybrid high performance CPU+ GPU systems and CUDA streams. *Computing and Informatics* 39, 3 (2020), 510–536.
[19] Jan Novotnỳ, Karel Adámek, and Wes Armour. 2021. Implementing CUDA streams into astroaccelerate—a case study. arXiv:2101.00941. Retrieved from https://arxiv.org/abs/2101.00941
[20] Wencheng Han, Hao Li, Maoguo Gong, Jianzhao Li, Yiting Liu, and Zhenkun Wang. 2022. Multi-swarm particle swarm optimization based on CUDA for sparse reconstruction. *Swarm and Evolutionary Computation* 75 (2022), 101153.
[21] Weiguang Pang, Xiantong Luo, Kailun Chen, Dong Ji, Lei Qiao, and Wang Yi. 2023. Efficient CUDA stream management for multi-DNN real-time inference on embedded GPUs. *Journal of Systems Architecture* 139 (2023), 102888.
[22] Cédric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland. 2024. CUDASTF: Bridging the gap between CUDA and task parallelism. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. IEEE, 1–17.
[23] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and parallel GPU task scheduling for deep learning. *Annual Conference on Neural Information Processing Systems* 33 (2020), 8343–8354.
[24] ONNX Runtime developers. 2018. ONNX Runtime. (Nov. 2018). Retrieved from https://github.com/microsoft/onnxruntime
[25] Lei Chen. 2021. *Deep Learning and Practice with Mindspore*. Springer Nature.
[26] Vit Libal, Jonathan Connell, Gerasimos Potamianos, and Etienne Marcheret. 2007. An embedded system for in-vehicle visual speech activity detection. In *Proceedings of the IEEE Workshop on Multimedia Signal Processing*. IEEE, 255–258.
[27] Kyeong Hoon Jung and Kang Yi. 2018. Vision-based blind spot monitoring using rear-view camera and its real-time implementation in an embedded system. *Journal of Computing Science and Engineering* 12, 3 (2018), 127–138.
[28] Zoran Nikolić. 2014. Embedded vision in advanced driver assistance systems. In *Proceedings of the Advances in Embedded Computer Vision*. Springer, 45–69.
[29] Keping Yu, Liang Tan, Long Lin, Xiaofan Cheng, Zhang Yi, and Takuro Sato. 2021. Deep-learning-empowered breast cancer auxiliary diagnosis for 5GB remote E-health. *IEEE Wireless Communications* 28, 3 (2021), 54–61.
[30] A. Sinha, A. Sharma, L. A. P. Melek, and D. Caviglia (Eds.). 2023. Smart embedded systems: Advances and applications.
[31] Chaoying Tang, Shuhang Xia, Mengen Qian, and Biao Wang. 2021. Deep learning-based vein localization on embedded system. *IEEE Access* 9 (2021), 27916–27927.
[32] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. arXiv:1802.05668. Retrieved from https://arxiv.org/abs/1802.05668
[33] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. 2023. A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Transactions on Intelligent Systems and Technology* 14, 6 (2023), 1–50.
[34] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2022. A survey of quantization methods for efficient neural network inference. In *Proceedings of the Low-Power Computer Vision*. Chapman and Hall/CRC, 291–326.
[35] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. 2021. Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights. *Proceedings of the IEEE* 109, 10 (2021), 1706–1752.
[36] Dayou Du, Gu Gong, and Xiaowen Chu. 2024. Model quantization and hardware acceleration for vision transformers: A comprehensive survey. arXiv:2405.00314. Retrieved from https://arxiv.org/abs/2405.00314
[37] Jason Sanders and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-purpose GPU Programming*. Addison-Wesley Professional.
[38] H. Yang, S. Shen, Z. Liu, and Y. Zhao. 2023. cuXCMP: CUDA-accelerated private comparison based on homomorphic encryption. *IEEE Transactions on Information Forensics and Security* 19 (2023), 3581–3592.
[39] Shanjiang Tang, BingSheng He, Shuhao Zhang, and Zhaojie Niu. 2016. Elastic multi-resource fairness: balancing fairness and efficiency in coupled CPU-GPU architectures. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. 875–886.
[40] Chen Zhao, Wu Gao, Feiping Nie, and Huiyang Zhou. 2021. A survey of GPU multitasking methods supported by hardware architecture. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2021), 1451–1463.
[41] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU computation using task graph parallelism. In *Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par)*. 435–450.
[42] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2021), 1303–1320.

[43] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing pipeline parallelism using control taskflow graph. In *Proceedings of the ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 283–284.
[44] Chun-Xun Lin, Tsung-Wei Huang, and Martin DF Wong. 2020. An efficient work-stealing scheduler for task dependency graph. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 64–71.
[45] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. (2017).
[46] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. IOS: Inter-operator scheduler for cnn acceleration. *Machine Learning and Systems* 3 (2021), 167–180.
[47] Intel Corporation. 2018. *Intel VTune Profiler*. Retrieved from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html/
[48] AMD Inc. 2025. *AMD HIP Documentation*. Retrieved from https://rocm.docs.amd.com/projects/HIP/en/latest/
[49] ([n. d.]). Retrieved from https://denglinai.com
[50] 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 (CVPR)*. 770–778.
[51] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805
[52] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv:2010.11929. Retrieved from https://arxiv.org/abs/2010.11929
[53] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional networks for biomedical image segmentation. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 234–241.
[54] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 10684–10695.
[55] Black Forest Labs, Stephen Batifol, Andreas Blattmann, Frederic Boesel, Saksham Consul, Cyril Diagne, Tim Dockhorn, Jack English, Zion English, Patrick Esser, Sumith Kulal, Kyle Lacey, Yam Levi, Cheng Li, Dominik Lorenz, Jonas Müller, Dustin Podell, Robin Rombach, Harry Saini, Axel Sauer, and Luke Smith. 2025. FLUX.1 Kontext: Flow Matching for In-Context Image Generation and Editing in Latent Space. arXiv:2506.15742. Retrieved from https://arxiv.org/abs/2506.15742

Received 3 March 2025; revised 11 October 2025; accepted 15 October 2025