2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11132809

# PASK: Cold Start Mitigation for Inference with Proactive and Selective Kernel Loading on GPUs

Xuanteng Huang*, Jiangsu Du†, Nong Xiao‡, Xianwei Zhang§
*School of Computer Science and Engineering*
*Sun Yat-sen University*
Guangzhou, China
*huangxt57@mail2.sysu.edu.cn, {†dujiangsu, ‡xiaon6, §zhangxw79}@mail.sysu.edu.cn

***Abstract*—Today, DNN inference is widely adopted, with numerous inference services being spawned from scratch across instances in scenarios such as spot serving, serverless scaling and edge computing, where frequent start-stops are required. In this work, we first delve into the inference workflow and uncover the origins of cold start when invoking a DNN model. Specifically, DNN execution is blocked by the kernel loading process to prepare the code object executing on GPU at the DL primitive library (e.g., cuDNN and MIOpen). To tackle this, we propose PASK, a kernel loading and reusing middleware to mitigate the widespread cold start issue. Unlike the reactive kernel scheduling policy used by existing frameworks, PASK adopts a proactive strategy to interleave code loading, kernel issuing and GPU computation to achieve higher hardware utilization. To further reduce the loading overhead, PASK recycles existing loaded kernels to accomplish the DNN operator, rather than introducing new kernels for every layer. Meanwhile, PASK categorically organizes the cached kernels to efficiently find the applicable kernel for reuse and thus minimize incurred runtime overhead. We implement and evaluate PASK atop of open source DNN inference engine and primitive library on off-the-shelf GPUs. Experiments demonstrate PASK is capable of alleviating the cold start overhead of popular DNN models with $5.62\times$ speedup on average.**

***Index Terms*—Cold Start, GPU, DNN, Inference**

## I. INTRODUCTION

The last decade has witnessed the unprecedented prosperity of deep learning and its widespread application in many aspects like computer vision [1]–[9], recommendation [10], and natural language processing [11]. Serving model inference has become the dominating workload in modern data centers, which accounts for more than 90% infrastructure costs in AWS [12] and 200 trillion daily invocations in Meta [13].

This trend also sparks the research efforts devoted into the ML inference infrastructures like request batching [14]–[17], model selection [18], [19], memory optimizations [20], tensor pre-fetching [21] and resource sharing [22], [23]. As underlying support, the accelerators like GPUs are trending to be much more powerful and the DNN operators are continuously optimized by experts for efficient computation [24]–[27]. As a result, DNN inference tends to be more hindered by the non-computation aspects, with a significant portion being spent on the models' cold starting process [28]. Fig. 1(a) compares the cold and hot execution times (ratios between the first and successive iterations) of representative DL models on both data center and consumer-grade GPUs from different vendors, where significant cold start slowdowns are commonly observed on respect platforms ($23.7\times$, $19.5\times$ and $31.3\times$ for MI100, A100 and 6900XT, respectively). Fig. 1(b) further demonstrates the breakdowns of the cold start overhead, which is classified into four parts by their execution ordering phases. Before launching a kernel onto the GPU, the host runtime (i.e., CUDA or HIP) is responsible for checking whether the kernel code object (e.g., compiled SASS instructions binary stream) is present in the managed host memory. If not, the runtime will load it from either the shared library or compiled ELF executables, then set memory permissions and look up the target symbol positions. It can be captured from Fig. 1(b) that the majority (65.8%) of cold start overhead is attributed to loading absent kernel code objects (`code loading`) while the computation (`GPU execution`) constitutes only a small proportion (8.4%). Therefore, the code loading should be considered alongside data pre-fetching [21], keep alive [29] and pre-warming [30] techniques to mitigate DNN cold start overhead.

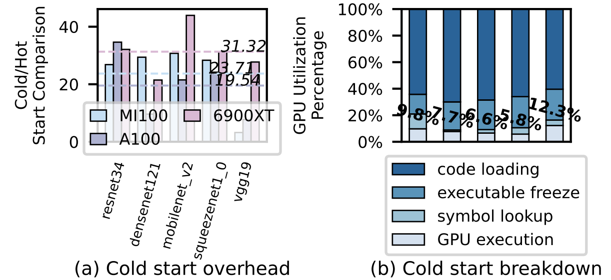


Fig. 1. DNN model cold start (a) overhead and (b) breakdown of typical models. The dashed lines in (a) represents averaged slowdowns and percentage numbers in (b) mark the GPU execution portion.

The cold start overhead originated from code loading is inevitable in scenarios like preemptive serving [31]–[36], serverless scaling [17], [20], [29], [37] or edge computing [30], [38], [39]. For example, the inference service has to be migrated across preemptive instances, or deployed in new instances to cope with request spikes. In edge or mobile devices, the inference service is forced to be suspended or swapped out due to limited hardware resources (e.g., memory capacity), and restarts later. Under these circumstances, code loading is unavoidable when starting a DNN model as there are no available loaded kernels in the system.

There are multiple proposed approaches to mitigate the non-computation overhead of DNN models [25]–[27], [40]. TASO [25], Rammer [26], Cocktailer [27] and MonoNN [40] apply some operator-level transformations on the models' computation graph for higher execution efficiency. Nevertheless, these methods focus on the *static* optimizations to produce a series of fast kernels running on GPUs, failing to consider the crucial runtime overhead of loading these kernels. There also exist designs sharing container [29], [41], runtime [30], [42] or tensor [20] to warm up the cold models, but

§ Xianwei is the corresponding author.
This research was supported by the National Key R&D Program of China under Grant NO. 2022ZD0115304, the National Natural Science Foundation of China #62472462, #62461146204, #62402534, #62332021, the GuangDong Basic and Applied Basic Research Foundation: 2023A1515110117, and was sponsored by CCF-Tencent Rhino-Bird Open Research Fund (CCF-Tencent RAGR20240102).



Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.

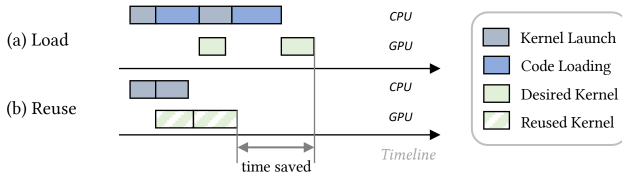


Fig. 2. Instead of waiting for the fresh loading (a), PASK reuses existing loaded kernels (b) to implement the DNN layers.

the sharing granularity is generally too coarse as DNN models are monolithic and cannot be efficiently transformed via dependency analysis for traditional services [43]. As such, extra steps are needed to transform the shared resources for accommodating with the cold start model. NNV12 [38] optimizes the DNN cold start process via selecting different kernels to implement the NN operator to reduce the tensor layout interchange overhead before execution. Nonetheless, the above existing methods ignore the kernel code object loading procedure, which actually constitutes the most significant portion of the cold start overhead for DNN models (Fig. 1).

In this work, we propose PASK, a kernel loading and reusing approach to mitigate the cold start latency of DNN inference. We first analyze the DNN inference serving procedure and attribute the cold start overhead to the mismatch between static model materialization and runtime kernel loading: inference framework determines the finalized kernel sequences from abstract model computation graph without considering the incurred loading overhead at runtime. To cope with this, PASK seeks the opportunities to *reuse* existing potentially less performant but already loaded kernels to materialize NN operator, thereby skipping the more substantial kernel loading procedure, as depicted in Fig. 2.

However, it's non-trivial to efficiently select the substitute from all cached kernels as checking whether the kernel is applicable for current problem can be pretty time-consuming. To this end, we design a categorical cache which stores the cached kernels based on their patterns to minimize the applicable checking overhead. We integrate PASK into the open source inference stack with AMD GPUs [1], and conduct evaluations on various DNN models which show remarkable performance improvements.

To summarize, this work makes the following contributions:

- We investigate the origins of DNN model cold start overhead, and uncover it is primarily due to the mismatch between static model materialization and runtime kernel loading.
- With the insights, we propose PASK, a kernel loading and reusing approach leveraging the already loaded kernels to finalize the NN operator, thus circumventing the substantial kernel loading process. Equipped with interleaved execution and categorical cache, PASK achieves higher GPU utilization meanwhile minimizing the incurred runtime overhead.
- We implement PASK atop of open source inference software stack, and evaluate using representative DNN models to demonstrate its ability to mitigate the cold start overhead.

## II. BACKGROUND AND MOTIVATION

### A. DNN Model Serving with GPU

GPU has become the *de-facto* accelerator to speed up DNN execution. A DNN model is composed of multiple layers with each defining a specific operation like convolution or pooling. To harness the hardware resources, DNN operators are offloaded onto GPUs for faster execution through various frameworks like TensorRT [44] for NVIDIA and MIGraphX [45] for AMD GPUs. These frameworks *materialize* the DNN layers with the computation graph optimizations [46]–[48] and tuned kernels [24], [49] for efficient execution.

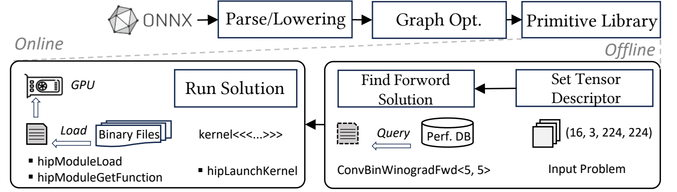


Fig. 3. DNN inference serving workflow. The serving framework firstly lowers the model and applies multiple optimizations on the requested model, then invokes operators from the primitive library to materialize the model.

**Offline preparation**. Fig. 3 depicts a typical workflow of DNN serving with GPU, where the requested model is submitted in the ONNX format containing multiple canonical operators [50] for the ease of model description and exchange. After receiving the model, the serving framework first converts the ONNX structure into lower descriptions with regard to the backend hardware, including GPU context creation, host-device synchronization setup, and tensor memory layout generation. Then the serving framework applies a series of hardware-independent graph-level optimization passes like *dead code elimination* and *common subexpression elimination* [45], [51] to reduce unnecessary computations. Finally, the serving framework exploits the highly-optimized kernels from vendor-provided primitive libraries (e.g., MIOpen and cuDNN) to accomplish the operators defined in the computation graph.

For each DNN layer (e.g., convolution), the serving framework sets the tensor descriptors needed by the primitive library with input *problem* (image and filter sizes, number of filters, data types etc.). Then the library finds the optimal *solution* [2] for the given problem from all applicable ones by querying an integrated database [52] which records the anticipated performance of each solution on current problem. After this, the lower operations and the determined optimal solutions for each layer are saved in framework-specific formats (e.g., `.trt` files for TensorRT and `.mgx` files for MIGraphX) for online inference request serving. The framework usually maintains a model registry [53] to store the lowered model and directly loads them when the request comes to avoid redundant lowering.

**Online inference serving**. When the inference request arrives, the framework invokes the corresponding optimized model and runs the determined solutions sequentially by launching the underlying kernels onto the GPU. Nevertheless, there are often cases where the code object of the desired kernel is absent due to the lazy loading behavior [54] which delays the loading until launch. Therefore, the primitive library has to *load* the code objects for the missing kernels from compiled binary files (e.g., dynamic linked ELF), resulting in unbearable cold start latency for model inference. Only after the code objects are loaded into memory will the kernels be issued to the GPU for computation. Such a reactive launch and lazy loading behavior leads to substantial latency at model cold start. Note that the loading delay comes from the mismatch between the offline operator lowering and the runtime solution loading: the primitive library determines solutions from the GPU performance perspective, meanwhile introducing extra loading overhead when launching missing kernels. In

[1]This can be implemented on NVIDIA platforms as well if more flexible interfaces are accessible.

[2]One solution may contain multiple kernels to transform input/output tensor layout/precision, conduct primary computation and epilogue reduction.



Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.

principle, the gap can be effectively bridged if one can proactively interleave the code loading and kernel launch executions, and reuse already loaded kernels to accomplish the computation rather than loading missing kernels from scratch each time.

### B. Generalized Solution for Wider Applicability

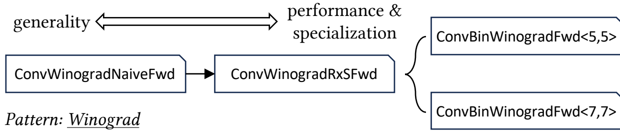


Fig. 4. Generality-performance trade-offs for *Winograd* solutions. Specialized solutions are more performant but applicable for specific problems, while generic solutions can be applied to more problems but are less efficient.

**Solution specialization**. Fig. 4 demonstrates some exemplified convolution solutions embedded in MIOpen. The solution usually follows certain *patterns* to conduct the computation. For example, solutions using *Winograd* [55] accomplish the convolution via matrix multiplication, and apply some transformations on both the input image and filters to lessen the number of multiply-and-add instructions. Other commonly used patterns include *GEMM*, *DirectConv* and *ImplicitGEMM*. Solutions with the same pattern typically follow similar workflows to accomplish the computation. Besides, there are further optimization spaces to exploit if certain problem fields are deterministic before the model starts (i.e., compile time). In Fig. 4, instead of taking the `ConvWinogradNaiveFwd` solution which accepts inputs with any dimension, one can alter to invoke `ConvBinWinogradRxSFwd` if the input is deemed to be a 2D image. By avoiding unnecessary loops and boundary checks, the latter hence brings in higher GPU performance, but at the expense of generality. Furthermore, the library opts for leveraging more specialized solution `ConvBinWinogradFwd<5,5>` if the filter size is also a specific value ($5 \times 5$), which helps organize the shared memory layout and better overlap the memory access and computation within the kernel. The highly specialized solutions produce better GPU performance tailored for the specific problem, but they are not applicable to other problems as there are more constraints. Currently, the primitive libraries (cuDNN, MIOpen) tend to harness the specialized solutions from the performance perspective, and fallback to the more general solutions only if no specialized solution is located. Inevitably, this leads to more solution loads at the model cold start, as the library may select an uniquely specialized solution for each individual layer with varying problem description.

**Solution applicability**. To ensure successful executions and correct results of DNN inference, the primitive library should only return *applicable* solutions for a given problem. The applicability checking process is time-consuming as it includes complicated logic such as workspace size checking, input format match, environment variable and hardware capability validation. In MIOpen, each solution implements the `IsApplicable` interface for the library to check whether it can be used to accomplish the computation for a given problem without any constraint violations (e.g., data type mismatch, out-of-bound memory access). Apparently, the incurred runtime overhead can be significantly alleviated if the applicable checks can be reduced to quickly identify a valid substitutive solution.

## III. Design

In this section, we present the components and workflow of PASK. Fig. 5 illustrates the overview of our design, where PASK receives the lowered model as its input. During model parsing, PASK chooses to proactively load the solutions once they are parsed and starts the execution immediately once they are loaded (section III-A). For an absent solution, PASK checks the applicability of loaded solutions to seek the opportunities of reusing existing solution to accomplish the computation, thereby skipping to load the missing one (section III-B). Upon reusing, PASK integrates a categorical solution cache (section III-C) to organize existing loaded solutions and provides an interface to efficiently select an applicable solution for the given problem. Finally, we elaborate the implementation of PASK atop of open source inference engine and DL primitive library (section III-D).

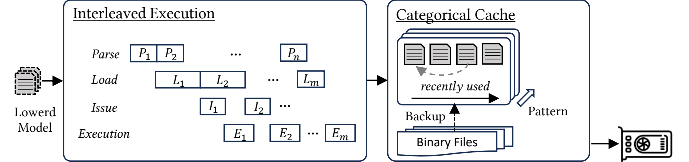


Fig. 5. Design overview. PASK receives the lowered model (a sequence of $n$ solutions) as input, and then proactively parses, loads, issues the DNN layer to improve the GPU utilization via interleaved execution. Upon absent solution, PASK seeks the opportunities to reuse the loaded solutions and organizes them based on their patterns for efficient applicability checking.

### A. Proactively Interleaved Execution

We introduce the interleaved execution to cope with the the inefficient execution hindered by the reactive launch and lazy loading pattern mentioned in section II-A. During parsing layers in DNN computation graph, PASK spawns two extra host threads to handle code loading and kernel issuing, respectively (Fig. 5). For a layer $i$, once it gets parsed ($P_i$, de-serialize the framework-specific model file to obtain the determined solution), the loading thread immediately starts to load the code objects of the desired solution ($L_i$). Once finish, the loading thread notifies the issuing thread to issue the kernels of the loaded solution ($I_i$) and execute them on GPU ($E_i$). This proactive design loads the necessary code objects in advance (once the parse finishes), therefore launches of the corresponding kernels incur no loading overhead. Besides, PASK starts the kernels at the earliest feasible time right after the loading. For comparison, current reactive serving frameworks start to launch the solutions only after all layers are parsed, and load the absent code objects on demand during launch. PASK mitigates the inefficiency with proactive interleaved execution, thereby achieving higher GPU utilization than traditional reactive inference engines.

As the DNN layer parsing is much faster than the code loading and execution, there will be a layer $m$ where all the $n$ layers are parsed and all preceding layers, including $m$, have finished their executions on GPUs (Fig. 5). We denote layer $m$ as the *milestone* where PASK unconditionally loads the missing solutions into cache set $\mathcal{S}_m$ before $m$ and selectively reuses the cached solutions after $m$. Note the minimal duration of a model execution is no less than the parsing process, hence there are free time slots for PASK to load before $m$ solutions, which act as cached alternatives for the selective reuse in subsequent layers.

### B. Selective Solution Reuse

After the milestone layer $m$, PASK quickens the execution of the remaining layers by avoiding unnecessary solution loads with selective adoption of cached solution. *Algorithm* 1 lists the solution reusing procedure where the EXECUTIONPLAN function describes the layer-wise reuse decision procedure. When the target solution $s_i^*$ ($i > m$) in layer $i$ is parsed, PASK checks whether the corresponding



Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.

code object has been already loaded. If so, $s_i^*$ will be directly used to complete the computation defined in layer $i$ without incurring any loading overhead. Otherwise, PASK firstly attempts to acquire the applicable alternative solution by querying the cached solution set $\mathcal{S}_m$ (section III-C). Once a valid substitutive solution $\hat{s}_i$ exists, PASK will reuse it to accomplish the computation in layer $i$ rather than loading the original absent solution $s_i^*$ from scratch.

If, unfortunately, no valid solution can be found in the cached set $\mathcal{S}_m$, PASK has no choice but to load the desired solution $s_i^*$.

**Algorithm 1** DNN execution with solution reuse

```
Input:
    n: number of layers in the DNN model
    m: the milestone layer (section III-A)
    S_m: solution cache set at layer m
    p_i: problem to be solved in layer i
    s_i^*: statically optimal solution in layer i
 1: function EXECUTIONPLAN(S, L, i)
 2:     for layer i from m + 1 to n do
 3:         if s_i^* ∈ S_m then                      ▷ s_i^* already loaded
 4:             compute layer i with s_i^*
 5:         else
 6:             ŝ_i ← GETSUBSOLUTION(S, s_i^*, p_i)
 7:             if ŝ_i ≠ null then          ▷ reusable solution found
 8:                 run layer i with substitutive solution ŝ_i
 9:             else                          ▷ no applicable solution
10:                 wait for s_i^* loading
11:                 add s_i^* to S_m
12: function GETSUBSOLUTION(S, s^*, p)
13:     list ← S[s^*.pattern]
14:     for s ∈ list's most recently used solution do
15:         if s.IsApplicable(p) then
16:             lru.update(s)
17:             return s
18:     return null
```

### C. Categorical Solution Cache

In *Algorithm* 1, PASK queries the solution cache for a loaded solution to be reused for implementing the operator. As discussed in section II-B, it is expensive to lookup the applicability with current problem for every loaded solution in the cache. PASK thus adopts a categorical solution cache design to effectively find an applicable solution from all existing loaded ones.

In PASK, the loaded solutions are categorically organized in separated lists based on their *patterns* (section II-B). When a solution is loaded/invoked, it will be inserted/moved to the head of list. The GETSUBSOLUTION function in *Algorithm* 1 details how PASK handles the query for a substitutive solution given the desired solution $s^*$ and problem $p$. It seeks for $\hat{s}$ from the list with the same pattern as $s^*$ in the the most recently accessed (load or lookup) order. If one applicable solution is found in the categorical list, it is returned and moved to the list head (Fig. 5). Otherwise, the cache returns $null$, indicating there are no applicable solutions for current problem.

PASK adopts this categorical design to minimize the number of lookups (i.e., solution applicability checking for a problem) when searching for $\hat{s}$. When a specialized solution for problem $p$ is missing, it is more likely to find a more general one with the same pattern to solve $p$ (Fig. 4). If there are no applicable solution in one categorical list, PASK regards the query failed and returns $null$ directly to skip more lookups for other categories. We search the solutions in the recent access order similar to the LRU cache, as the neighboring layers in DNN models usually possess similar problem descriptions. Therefore, recently used solutions are more likely to be applicable to current problem.

### D. Implementation

We implement PASK on the basis of the open source inference engine MIGraphX [45] and DL primitive library MIOpen [56] for

TABLE I
EVALUATED DNN MODELS.

| Model | # Primitive Layers | Type | Abbr. |
|---|---|---|---|
| AlexNet [1] | 5 | Img. Rec. | alex |
| VGG16 [2] | 16 | Img. Rec. | vgg |
| ResNet34 [3] | 14 | Img. Rec. | res |
| RegNet_Y_800MF [58] | 28 | Img. Rec. | reg |
| EfficientNet_B7 [4] | 58 | Img. Rec. | eff |
| Faster_R-CNN [5] | 16 | Obj. Det. | rcnn |
| SSD300 [59] | 27 | Obj. Det. | ssd |
| FCN [6] | 18 | Sem. Seg. | fcn |
| UNet [60] | 37 | Sem. Seg. | unet |
| VIT_B_16 [7] | 1 | ViT | vit |
| Swin_B [8] | 1 | ViT | swin |
| Swin_V2_B [9] | 1 | ViT | swin2 |

AMD GPUs. PASK acts as a transparent middleware for end users.

We enhance MIGraphX to support the interleaved execution (section III-A). MIGraphX abstracts the computation graph by a list of instructions, which define the DNN operators and their corresponding input/output tensors. For interleaved execution, PASK spawns 3 host threads responsible for parsing, loading and issuing kernels, respectively (Fig. 5). We leverage single-producer-single-consumer (SPSC) channels to coordinate the interactions among host threads in MIGraphX. Instructions are fed to the loading thread when finishing the parse and leaves for the interleaved execution in the FIFO order. For each callee kernel of the solution, PASK invokes `hipModuleLoad` and `hipModuleGetFunction` (Fig. 3) to proactively load the target ELF sections from the binary files and determine location of the desired symbol by the kernel name.

We further augment MIOpen to support the solution reuse and loading skip in PASK. When the inference framework invokes `miopenRunSolution`, PASK follows *Algorithm* 1 to reuse the potentially applicable loaded solution for the computation in current layer. PASK implements the reuse logic for the common convolution, pooling and activation primitives in MIOpen.

## IV. EXPERIMENT SETUP

**Testbed**. We evaluate PASK in a sever equipped with AMD EPYC 7773X CPU and AMD Instinct MI100 GPU, which possesses 32 GB VRAM and 120 Compute Units. The DNN inference server is implemented atop of MIGraphX and MIOpen as the graph optimization engine and DL primitive library backended by ROCm 6.0.2.

**Workloads**. We collect representative DNN workloads (Table I) from multiple fields (e.g., image recognition, object detection, semantic segmentation and vision transformer) implemented the PyTorch model zoo as served models in inference requests, where the vision transformer models are also used to emulate the transformer-based language models. Models are exported as ONNX format from PyTorch under the default configuration and then fed into MIGraphX for compute graph optimizations, and finally lowered to the corresponding GPU operations. The number of MIGraphX layers for each model is also listed in Table I. We follow the common image settings used in the ImageNet [57] dataset.

**Evaluated schemes**. To faithfully validate the effectiveness of PASK, we evaluate the above models under the following schemes:

- *Baseline*: the default inference workflow used by MIGraphX inference framework which loads every absent solutions.
- *NNV12*: the cold start mitigation design in NNV12 [38] to avoid input/output tensor layout transform.
- *Ideal*: hot execution of the DNN model, with all the desired solutions already resident.
- *PaSK*: full design of PASK, including interleaved execution and categorical solution cache.
- *PaSK-I*: PASK with only interleaved execution module being enabled, no solution reuse mechanism is enabled.



Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.

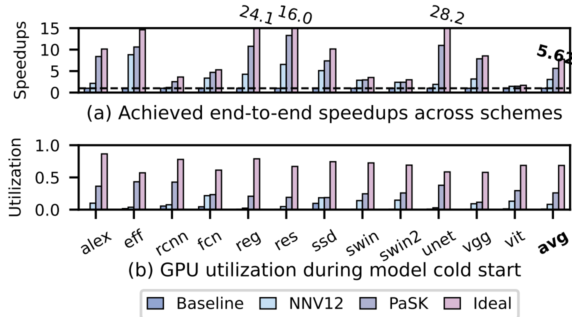


Fig. 6. (a) End-to-end cold start speedups for evaluated models under different schemes; (b) GPU utilization (fraction of time when GPU is active for computing) during model cold start.

- *PaSK-R*: PASK with solution reusing only, which exhaustively checks the applicability of every cached solution.

## V. Results and Analysis

### A. Cold Start Mitigation

We first investigate the achieved cold start mitigation effect under different schemes. Fig. 6(a) compares the end-to-end cold start speedups under *Baseline*, *NNV12*, *PaSK* and *Ideal* schemes, where *PaSK* achieves 5.62× speedups over *Baseline*. Compared to *Baseline*, *NNV12* possesses only moderate speedups (3.04×). *NNV12* mitigates the cold start by avoiding tensor layout transforms (e.g., NCHW → NHWC) between layers, limiting the optimize opportunities on specific layers. Furthermore, we can observe that *PaSK* performs better on models with more primitive layers (`eff`, `unet`, `reg` and `ssd`), as there are more opportunities for PASK to reuse the existing substitutive solution to replace the absent one to accomplish the computation. On the other hand, transformer models (`vit`, `swin` and `swin2`) perceive less end-to-end cold start latency improvement compared to other convolution-based models. As transformer models primarily comprise of GEMM operators from the BLAS library, there is only one primitive layer provided by MIOpen (Table I). Hence, there are less reuse chances as PASK could be invoked only when the primitive routine is called.

TABLE II
Cold start speedup with varying inference batch sizes.

| Batch Size | 1 | 4 | 16 | 64 | 128 |
|---|---|---|---|---|---|
| *NNV12* | 3.04× | 2.82× | 2.49× | 1.91× | 1.74× |
| *PaSK* | 5.62× | 5.24× | 4.55× | 3.91× | 3.10× |
| *Ideal* | 7.75× | 7.23× | 6.94× | 6.78× | 6.41× |

We further demonstrate the cold start improvements with varying batch size for each inference request in Table II. With enlarged batch size, GPU is better utilized and the proportion of duration consumed on code loading is thereby lower. Besides, hot execution is still far more efficient compared to cold start, while *PaSK* still outperforms *NNV12* with a large margin (210%).

### B. Utilization and Breakdown

To understand the effectiveness of PASK, we adopt the hardware utilization (the proportion of time when GPU is active relative to the entire inference duration) as the metric to explain how PASK boosts the cold start performance of DNN models on GPU. Fig. 6(b) illustrates the achieved utilization under the various schemes, where we can capture that the GPU active time accounts for only a few portion (8.2%) of model execution for *NNV12*. As the majority of time is spent on kernel code loading, the cold start latency is several times higher than the one in hot start. *PaSK* avoids this substantial overhead by reusing the applicable solutions from rather than always loading the missing one, and effectively elevates the utilization to 25.9% on average. For *Ideal*, since all solutions are already cached, there is no loading overhead when launching the desired kernels onto the GPU thus the accomplished utilization is up-to 68.5% on average.

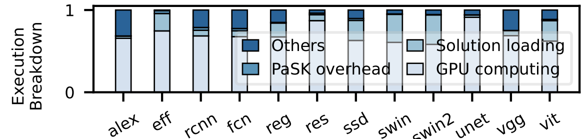


Fig. 7. Model cold start breakdown for *PaSK*.

Figure 7 further depicts the breakdowns of times consumed on different stages during model execution. We compare the ratios of times spent on GPU computing, solution loading, PASK overhead and others (e.g., host-device synchronization, model parse). The portion of time consumed on solution loading (11.2% on average) is acceptable compared to the cold start execution without dedicated management (Fig. 1). For transformer models (`vit`, `swin`, `swin2`), the solution loading proportion is relatively larger than other models, as the major operations are GEMMs. Moreover, the extra time incurred by PASK takes only 1.3% of the entire execution on average, indicating that PASK can efficiently retrieve the applicable alternative solution with negligible overhead.

### C. Ablation Studies and Cache Statistics

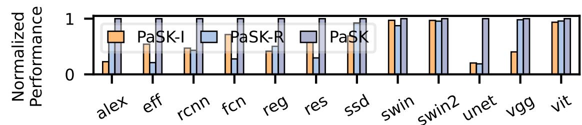


Fig. 8. Normalized performance of variants compared to *PaSK*.

In this section, we compare *PaSK* with its variants *PaSK-I* and *PaSK-R* to validate the effectiveness of solution reusing and categorical cache. For *PaSK-I*, the cold start latency reduction comes from the interleaved execution. Regarding *PaSK-R*, it adopts the naive cache which exhaustively searches all cached solution to find the optimal one, suffering from spending too much time in the applicability checking which is time-costly (section II-B). Fig. 8 illustrates the achieved proportional performance normalized to *PaSK*. For models perform worse in *PaSK-I* (`alex`, `vgg`, `unet`), we find the GPU execution durations before the milestone layer are short, thus there are less opportunities for *PaSK-I* to overlap the loading and execution process in these models. And we can only observe nuances of performance for transformer models, as they contain only one DL primitive operator (Table I) thereby possess few reuse opportunities.

To further investigate the impact of categorical cache, we collect the cache statistics during cold start. The cache hit rate describes the averaged possibility of successfully finding a applicable solution following the GETSUBSOLUTION routine in *Algorithm* 1. Fig. 9(a) shows the hit rate of categorical cache in PASK across models (69.7% on average) where transformer models are omitted as they contain only one primitive operator. We can capture that models comprising of more operators possess higher hit rates, as there are more chances for PASK to load the desired solution into the categorical cache following *Algorithm* 1. For subsequent queries, it is more likely to find a applicable solution in a cache with more entries.

Fig. 9(b) presents the averaged number of lookups per query (i.e., number of solutions whose applicability are checked), where the



Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.

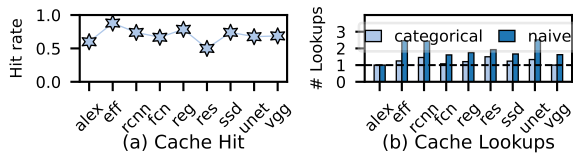


Fig. 9. Cache hit rate and averaged number of solution lookups for each hit.

categorical organization only uses 1.22 lookups per query while the naive searching requires 1.89 on average. As introduced in section II-B, checking the solution's applicability for a given problem is time-costly. Therefore, the naive solution searching strategy may result in severe performance downgrade due to enormous applicable checking overhead. It can be drawn from Fig. 9(b) that models with more lookups in the naive cache (`eff`, `unet`, `res`, `rcnn`) suffer from more pronounced performance downgrade in *PaSK-R* in Fig. 8. And the categorical organization in PASK can effectively reduce the number of required lookups per query, thus further boost the achieved performance.

## VI. DISCUSSIONS

**Library supporting**. As discussed earlier, the vision transformer models (`vit`, `swin`, `swin2`) possess less performance boosts from PASK because the major computation is matrix multiplication in the attention module [61]. Currently PASK is implemented atop of MIOpen, which contains the primitive operations used in DNN models (e.g., convolution, pooling), while the GEMM kernels are provided by the BLAS library (e.g., hipBLAS). Therefore, PASK can only identify and reuse a small portion of layers in the transformer models and offers limited cold start latency reduction for them. Note that it is trivial to extend PASK to support GEMM operators and accommodate with transformer models if more engineering efforts can be devoted to apply the similar modifications to hipBLAS. As hipBLAS and other vendor libraries (e.g. hipSPARSE [62] for sparse operators in GPU) following the same find-execution pattern (Fig. 3).

**More factors for kernel specialization**. Furthermore, PASK could be combined with other mechanisms by considering more factors. For instance, recent researches [63], [64] leverage the mix-precision components integrated in accelerators [65] to speedup the inference since DNN models usually have better tolerance on significance loss and numeric error. As the DNN operators are also specialized for tensor data types (`fp16`, `int8`), one may choose to still use high-precision data types if the corresponding kernels are already loaded while the low-precision ones are not. This offers the opportunities to accelerate the inference with high-precision computation by avoiding the overhead to load the absent kernels tailored for low-precision data types, even though the latter provide better performance on GPU. PASK can be generalized and applied to scenarios where the GPU kernels are specialized for some factors like input/output layout, data type, tensor tile size.

**Loading desired solutions**. Although PASK selectively skips the originally desired solutions to avoid the loading overhead, one can still load the skipped solutions during the intervals between two consecutive inference requests. Based on cloud workload traces [66], there are several seconds on average between the interval of two requests scheduled into the same inference instance. The interval duration is sufficient for PASK to proactively load the previously skipped solutions into the managed cache. Synergistically, they can act the candidates for reuse, or the desired solution which are already loaded for subsequent requests.

## VII. RELATED WORKS

### A. Cold Start Mitigation of DNN Models

Mitigating the cold start of DNN model has become the research appetite in recent years. Optimus [29] tackles the ML serverless container cold start problem by proposing the inter-function model transform approach which reuses operators from models of idle containers to compose the target model. Tetris [20] improves the serverless DNN inference latency and system container density via reusing duplicated tensors across instances. Pagurus [41] designs the algorithm to calculate the software dependency similarities between containers and then harness the idle container to build the intermediate image ready for helping to start the container rather than building from scratch. NNV12 [38] selectively loads the transformed weights in advance to avoid the tensor layout interchange overhead, and distributes the load and computation to different hardware units on edge/mobile devices.

These existing works share the container [29], [37], [41], service runtime [30], or model weights [20] among models. However, DNN models monolithic and cannot be transformed separately. Hence, some extra steps are needed to transform the shared resources to accommodate with the target model (e.g., modify the structure of warm models [29]). There are also general cold start mitigation approaches designed for cold microservice or serverless environments [37], [41], [67]–[70], but they fail to investigate the inference workflow and propose dedicated optimizations for DNN model invocations.

### B. DNN Inference Optimizations

As the new computing paradigms thrive in recent years, cold start mitigation approaches have been proposed for preemptive spot [31], [33], [35], [41] computing and serverless computing [37]. For DNN model inference services under these two circumstances, the cold start process is inevitable as the serving instance has to be migrated across instances, or spawned in a new environment to satisfy the resource demands. PASK can be further integrated into the spot serving and serverless frameworks to mitigate the cold start issues with dedicated kernel loading management.

To harness the GPU capability for accelerating DNN computation, performance experts attempt to optimize operator graphs and model workflows to alleviate the memory bandwidth bottlenecks. TASO [25], Rammer [26], Cocktailer [27] and MonoNN [40] are graph transformation methods applying the intra- and inter-operator optimizations for the better hardware utilization. However, these approaches solely consider the kernel efficiency on GPU, and ignore the overhead of loading the multifarious generated kernels. PASK can be used to enhance these methods by assisting with some runtime information (e.g., the loaded kernels) to generate a more efficient execution plan for cold models.

## VIII. CONCLUSION

In this work, we investigate the DNN model inference procedure and attribute cold start latency to the reactive scheduling strategy and kernel loading overhead. In order to mitigate the cold start latency, we propose PASK, a transparent middleware to schedule model kernels in a proactive strategy, and seek the opportunities to reuse existing loaded solutions organized categorically. We implement PASK on basis of the open source inference framework and primitive library and evaluate PASK with several popular DNN models. Detailed experiments demonstrate that PASK can effectively alleviate the cold start overhead for DNN model inference with 5.62× speedups compared to the default workflow used by the inference frameworks.



Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.

## References


[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[3] Kaiming He et al. Deep residual learning for image recognition. In *CVPR*, 2016.

[4] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.

[5] Shaoqing Ren et al. Faster R-CNN: towards real-time object detection with region proposal networks. *PAMI*, 2017.

[6] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

[7] Alexey Dosovitskiy et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[8] Ze Liu et al. Swin transformer: Hierarchical vision transformer using shifted windows. In *ICCV*, pages 9992–10002, 2021.

[9] Ze Liu et al. Swin transformer V2: scaling up capacity and resolution. In *CVPR*, 2022.

[10] Maxim Naumov et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv*, 2019.

[11] Jacob Devlin et al. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

[12] Deliver high performance ML inference with AWS Inferentia. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf.

[13] Michael J. Anderson et al. First-generation inference accelerator deployment at facebook. *arXiv*, 2021.

[14] Ahsan Ali et al. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC*, 2020.

[15] Pin Gao et al. Low latency RNN inference with cellular batching. In *EuroSys*, 2018.

[16] Gyeong-In Yu et al. Orca: A distributed serving system for transformer-based generative models. In *OSDI*, 2022.

[17] Chengliang Zhang et al. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *ATC*, 2019.

[18] Sohaib Ahmad et al. Proteus: A high-throughput inference-serving system with accuracy scaling. In *ASPLOS*, 2024.

[19] Daniel Mendoza, Francisco Romero, and Caroline Trippel. Model selection for latency-critical inference serving. In *EuroSys*, 2024.

[20] Jie Li et al. Tetris: Memory-efficient serverless inference through tensor sharing. In *ATC*, 2022.

[21] Yangyang Feng et al. Mobius: Fine tuning large-scale models on commodity GPU servers. In *ASPLOS*, 2023.

[22] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *EuroSys*, 2024.

[23] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined GPU scheduling. In *SOSP*, 2023.

[24] Tianqi Chen et al. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[25] Zhihao Jia et .al. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[26] Lingxiao Ma et al. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *OSDI*, 2020.

[27] Chen Zhang et al. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *OSDI*, 2023.

[28] Zhihao Bai et al. Pipeswitch: Fast pipelined context switching for deep learning applications. In *OSDI*, 2020.

[29] Zicong Hong et al. Optimus: Warming serverless ML inference via inter-function model transformation. In *EuroSys*, 2024.

[30] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. Pocket: ML serving from the edge. In *EuroSys*. ACM, 2023.

[31] Jiangfei Duan et al. Parcae: Proactive, liveput-optimized DNN training on preemptible instances. In *NSDI*, 2024.

[32] Xupeng Miao et al. Spotserve: Serving generative large language models on preemptible instances. In *ASPLOS*, 2024.

[33] Sanjith Athlur et al. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys*, 2022.

[34] Luo Mai et al. Kungfu: Making training in distributed machine learning adaptive. In *OSDI*, 2020.

[35] John Thorpe et al. Bamboo: Making preemptible instances resilient for affordable training of large dnns. In *NSDI*, 2023.

[36] Insu Jang et al. Oobleck: Resilient distributed training of large models using pipeline templates. In *SOSP*, 2023.

[37] Yanan Yang et al. Infless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS*, 2022.

[38] Rongjie Yi et al. Boosting DNN cold inference on edge devices. In *MobiSys*, 2023.

[39] Kaihua Fu et al. Qos-aware irregular collaborative inference for improving throughput of DNN services. In *SC*, 2022.

[40] Donglin Zhuang et al. Mononn: Enabling a new monolithic optimization space for neural network inference tasks on modern gpu-centric architectures. In *OSDI*, 2024.

[41] Zijun Li et al. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *ATC*, 2022.

[42] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Daydream: Executing dynamic scientific workflows on serverless platforms with hot starts. In *SC*, 2022.

[43] Hanfei Yu et al. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *ASPLOS*, 2024.

[44] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[45] AMD MIGraphX Documentation. https://rocm.docs.amd.com/projects/AMDMIGraphX/en/latest/.

[46] Size Zheng et al. Chimera: An analytical optimizing framework for effective compute-intensive operators fusion. In *HPCA*, 2023.

[47] Ganesh Bikshandi and Jay Shah. A case study in CUDA kernel fusion: Implementing flashattention-2 on NVIDIA hopper architecture using the CUTLASS library. *arXiv*, 2023.

[48] Ao Li et al. Automatic horizontal fusion for GPU kernels. In *CGO*, 2022.

[49] Introduction to torch.compile — PyTorch Tutorials. https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

[50] ONNX Operators - ONNX documentation. https://onnx.ai/onnx/operators/.

[51] Optimization — NVIDIA Triton Inference Server documentation. https://docs.nvidia.com/deeplearning/triton-inference-server/archives/triton_inference_server_1150/user-guide/docs/optimization.html.

[52] Using the performance database — MIOpen 3.2.0 Documentation. https://rocm.docs.amd.com/projects/MIOpen/en/latest/conceptual/perfdb.html.

[53] Model Repository — NVIDIA Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_repository.html.

[54] Nathan Otterness and James H. Anderson. Exploring AMD GPU scheduling details by experimenting with "worst practices". *Real Time Syst.*, 2022.

[55] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *CVPR*, 2016.

[56] MIOpen documentation. https://rocm.docs.amd.com/projects/MIOpen/en/latest/.

[57] Jia Deng et al. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[58] Ilija Radosavovic et al. Designing network design spaces. In *CVPR*, pages 10425–10433. Computer Vision Foundation / IEEE, 2020.

[59] Wei Liu et al. SSD: single shot multibox detector. In *ECCV*, 2016.

[60] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015.

[61] Ashish Vaswani et al. Attention is all you need. In *NeurIPS*, 2017.

[62] hipSPARSE documentation. https://rocm.docs.amd.com/projects/hipSPARSE/en/latest/.

[63] Ji Lin et al. AWQ: activation-aware weight quantization for on-device LLM compression and acceleration. In *MLSys*, 2024.

[64] Yilong Zhao et al. Atom: Low-bit quantization for efficient and accurate LLM serving. In *MLSys*, 2024.

[65] NVIDIA Tensor Cores: Versatility for HPC & AI. https://www.nvidia.com/en-us/data-center/tensor-cores/.

[66] Qizhen Weng. Beware of fragmentation: Scheduling gpu-sharing workloads with fragmentation gradient descent. In *ATC*, 2023.

[67] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *ASPLOS*, 2022.

[68] Ashraf Mahgoub et al. ORION and the three rights: Sizing, bundling, and prewarming for serverless dags. In *OSDI*, 2022.

[69] Zijun Li, Quan Chen, and Minyi Guo. Pagurus: Eliminating cold startup in serverless computing with inter-action container sharing. *arXiv*, 2021.

[70] Dong Du et al. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS*, 2020.




Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.