

FusedInf: Efficient Swapping of DNN Models for On-Demand Serverless Inference Services on the Edge

Sifat Ut Taki
University of Notre Dame
staki@nd.edu

Arthi Padmanabhan
Harvey Mudd College
arpadmanabhan@g.hmc.edu

Spyridon Mastorakis
University of Notre Dame
mastorakis@nd.edu

Abstract—Edge AI computing boxes are a new class of computing devices that are aimed to revolutionize the AI industry. These compact and robust hardware units bring the power of AI processing directly to the source of data—on the edge of the network. On the other hand, on-demand serverless inference services are becoming more and more popular as they minimize the infrastructural cost associated with hosting and running DNN models for small to medium-sized businesses. However, these computing devices are still constrained in terms of resource availability. As such, the service providers need to load and unload models efficiently in order to meet the growing demand. In this paper, we introduce *FusedInf* to efficiently swap DNN models for on-demand serverless inference services on the edge. *FusedInf* combines multiple models into a single Direct Acyclic Graph (DAG) to efficiently load the models into the GPU memory and make execution faster. Our evaluation of popular DNN models showed that creating a single DAG can make the execution of the models up to 14% faster while reducing the memory requirement by up to 17%. The prototype implementation is available at <https://github.com/SifatTaj/FusedInf>.

Index Terms—deep neural networks, optimization, serverless inference, edge computing

I. INTRODUCTION

Efficient DNN inference is crucial for making deep learning models practical in real-world applications on the edge [1], [2]. While DNNs are extremely capable of tasks like image recognition and speech translation, running them often requires significant computing power. This can be a bottleneck for deploying them on resource-constrained devices or in situations demanding fast response times. By optimizing DNN inference to use less power and run efficiently, we can unlock the potential of deep learning for a wider range of applications—from medical diagnosis on mobile devices to real-time obstacle detection in autonomous vehicles on edge.

Various cloud service providers are allowing users to deploy and run their models on the cloud and edge. However, owning and managing a virtual machine on the cloud for inference is very expensive—especially for small businesses, which do not always require the inference service. Hence, it is not economically viable for them to keep an inference service running idle. As a solution, cloud providers have developed serverless inference services for on-demand inference [3], [4]. This makes it easier to deploy and run DNN models; however, it severely complicates the process on the service provider’s

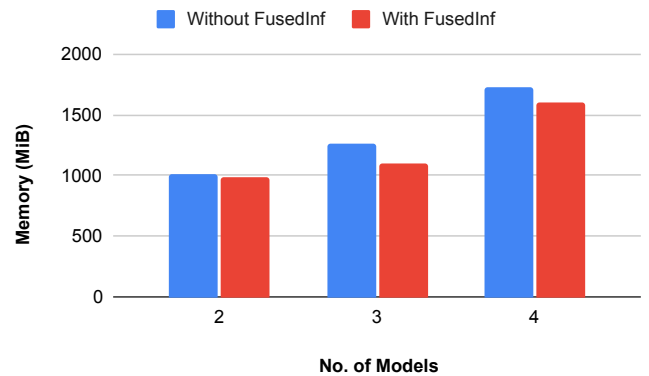


Fig. 1. Effectiveness of *FusedInf* when concurrently executing multiple DNN models for inference on the edge.

end—especially on the edge where resources are limited. The service providers need to constantly swap models on the limited resources of the edge boxes. Swapping models comes at the cost of an overhead of loading and unloading them every time a new model is queried by a user.

In order to address the aforementioned problem, we need to come up with a system that efficiently swaps models on edge devices. However, we should address a few challenges to ensure the security and correctness of each user and model. (C1) Can we ensure model correctness while retaining the accuracy of the models? (C2) Will the system efficiently work for a wide variety of models? (C3) How much overhead will there be? (C4) Can it be achieved while ensuring the privacy of the users?

FusedInf introduces a novel approach to efficiently swap models on the edge with very limited overhead while addressing the challenges mentioned above. *FusedInf* compiles a unified directed acyclic graph (DAG) of multiple models before loading them on the GPU memory. This facilitates the process of loading and querying multiple models at the same time. Figure 1 presents an overview of peak GPU memory usage when executing VGG16 [5], MobileNetv3 [6], DenseNet161 [7], and EfficientNetv2Large [8] for inference with and without *FusedInf*. The following are the major contributions of this paper:

TABLE I
EXAMPLES OF COMMERCIAL AI EDGE COMPUTING BOXES.

Vendor	Model	CPU	GPU	Memory
VVDN	Xavier NX	6-core NVIDIA Carmel ARM v8.2	NVIDIA Volta architecture with 384 NVIDIA CUDA cores and 48 Tensor cores	8GB
Advantech	EPC-R7300IJ	8-core ARM Cortex	NVIDIA Ampere GPU with 1024 CUDA cores and 32 Tensor Cores	16GB
FORLINX	FCU3001	6-core NVIDIA Carmel ARM v8.2	NVIDIA Volta architecture with 384 NVIDIA CUDA cores and 48 Tensor cores	16GB
EdgeMatrix	Advance Mk2	9th/8th Gen Intel Core i7	NVIDIA Tesla T4 with 2560 CUDA core and 320 Tensor cores	16GB
Azure Edge Stack	Pro 2	Intel Xeon Gold 6209U	NVIDIA A2 with 1280 CUDA cores and 40 Tensor cores	32GB

- We discuss the challenges and opportunities of optimizing CUDA operations for DNN model executions on edge.
- We introduce a multi-model DAG compilation technique to compile graphs for efficient loading of multiple models into the GPU memory.
- We demonstrate the effectiveness of this technique through a comprehensive evaluation of multiple DNN models.

II. BACKGROUND

The convergence of machine learning (ML) and edge computing is driving the development of new, powerful technologies specifically designed for edge boxes [9]–[11]. These compact computing devices process data locally—at the source of generation much closer to the users—rather than relying on communication with the cloud. This trend necessitates a paradigm shift from traditional cloud-optimized DNN models to lightweight and resource-efficient models tailored for resource-constrained edge devices [12]. One promising area of research is in the development of optimized AI models, which achieve acceptable accuracy with minimal computational power [13]. Additionally, the field is witnessing the emergence of frameworks specifically designed for edge deployment, focusing on optimizing AI workloads for edge hardware and enabling real-time, on-device inferencing. This synergy between AI and edge computing holds immense potential for applications across various sectors, from industrial automation and predictive maintenance to intelligent traffic management and personalized healthcare.

DNNs are revolutionizing various fields due to their ability to solve complex problems. However, training and running these computationally intensive models requires significant processing power. Compute Unified Device Architecture (CUDA) [14] plays a vital role in accelerating DNN models by leveraging the parallel processing capabilities of Graphics Processing Units (GPUs). CUDA provides a programming model and a set of development tools that allow developers to write DNN algorithms in languages like C++, exploiting the massive core count of GPUs for tasks like

matrix multiplication and convolution, which are fundamental operations in DNNs. Furthermore, libraries like CUDA Deep Neural Network library (cuDNN [15]) offer highly optimized implementations of commonly used DNN primitives, which further accelerate training and inference processes. The use of CUDA in DNN model training has significantly reduced training times and enabled the development of ever-more complex and powerful neural networks.

Within the NVIDIA ecosystem, cuDNN and CUDA APIs play significant roles in developing and deploying complex models. Their significance lies in two primary areas: performance optimization and developer efficiency. cuDNN leverages the parallel processing capabilities of NVIDIA GPUs, providing highly optimized implementations for fundamental deep learning operations like convolution, pooling, and activation functions. This offloading of computationally intensive tasks from CPUs to GPUs translates to significant speedups in training and inference times compared to CPU-only implementations. This performance boost is essential for training large-scale models with billions of parameters within reasonable timeframes. Furthermore, cuDNN simplifies the development process by offering pre-optimized kernels that integrate seamlessly with popular deep-learning frameworks like TensorFlow and PyTorch. This abstraction layer allows researchers and engineers to focus on model design and experimentation without going into low-level GPU programming, accelerating innovation in the field. As such, cuDNN’s performance optimizations and developer-friendly interface make it an indispensable tool for researchers and engineers pushing the boundaries of deep learning.

A. CUDA API Operations

One critical aspect to consider is the distinction between host (CPU) and device (GPU) memory. DNN models consist of weights, biases, and activation layers, all requiring memory storage. CUDA provides mechanisms for allocating memory on the GPU using functions like *cudaMalloc()* and *cudaMemcpy()*. However, simply allocating sufficient memory isn’t enough as Fragmentation (where allocated memory becomes

TABLE II
COMPARISON OF DIFFERENT MEMORY/LATENCY SAVING FRAMEWORKS.

Framework	Application	Requires model similarity	Impacts accuracy
Gemel [18]	Video analytics	Partial	Yes
HiveMind [19]	Concurrent training	Full	Yes
Mainstream [20]	Video analytics	Partial	Yes
AdaShare [21]	Multi-task learning	Partial	Yes
FusedInf	Serverless inference	No	No

scattered across the GPU memory space) can occur, which may hinder performance. Frameworks often employ memory pools to mitigate this issue, allocating contiguous memory blocks for better utilization [16].

Another key factor influencing memory usage is the data flow during DNN execution. Forward and backward passes in training involve numerous intermediate tensors representing activations and gradients. While some frameworks like cuDNN offer optimizations to reduce memory footprint, these intermediate tensors still consume significant resources. Techniques like checkpointing, where intermediate states are periodically saved to host memory, can be employed to free up GPU memory.

Furthermore, the choice of data type for model parameters significantly impacts memory requirements. While single-precision floating-point numbers (FP32) offer high accuracy, they can be memory-intensive for large models. Techniques like mixed-precision training, where computations are performed using lower precision formats like FP16, can significantly reduce memory usage without sacrificing substantial accuracy [17].

Beyond model parameters and intermediate tensors, other factors contribute to GPU memory consumption. Frameworks themselves allocate memory for internal data structures and workspace for cuDNN operations (such as CUDA CONTEXT). Moreover, depending on the complexity of the DNN architecture, additional memory might be required for storing activation functions and gradients.

Optimizing memory allocation for DNNs on GPUs requires a holistic approach. Techniques like memory profiling tools can help identify memory bottlenecks and guide optimization efforts. Utilizing techniques like gradient accumulation, where gradients for multiple mini-batches are accumulated before updating weights, can reduce memory requirements for backpropagation. Additionally, exploring alternative DNN architectures with lower memory footprints might be necessary for resource-constrained GPUs.

B. Recent Advancements

Model merging and operator fusion can optimize the concurrent execution of multiple DNN models on the edge. Gemel introduces a technique called model merging to improve

memory usage on edge devices for real-time video analytics on the edge. It merges similar layers from different models to reduce the overall memory footprint and the time it takes to swap data between host and GPU memory.

HiveMind, on the other hand, takes a similar approach to Gemel. HiveMind is a system designed to speed up the concurrent execution of multiple DNN models. It achieves this by grouping models into batches, then performing operator fusion across these models and sharing data efficiently. It utilizes a parallel processing system to execute this optimized group of models for faster performance. However, HiveMind requires manual model grouping. Similar to Gemel, HiveMind performs cross-model layer fusion when stateful operators in different models share the same underlying weights or when stateful operators have the same input and output shapes. This is an unlikely scenario in a serverless inference service where different users will query different models at a time.

Table II presents a comparison between our proposed approach and other DNN memory/latency saving frameworks. The aforementioned approaches save memory or time by sharing or reusing model layers and operators across multiple DNN models. This requires architectural similarity among those models. However, an on-demand serverless inference service on the edge may need to run a wide variety of models from different users that may contain little to no architectural similarity. Additionally, model similarity search introduces a significant overhead, which is not tolerable for fast and efficient model swapping on an edge device. Moreover, sharing layers across multiple models comes at a cost of reduced accuracy. As a service provider, it is important to ensure model correctness for serverless inference services as users expect no accuracy degradation during inference.

III. CHALLENGES & MOTIVATION

With the emergence of on-demand serverless inference services on the edge, service providers are expected to face a massive amount of traffic querying different models throughout the day [22]. However, commercial edge boxes are extremely resource-contained compared to cloud servers. Table I presents some of the commercially available edge AI computing boxes today. These edge boxes rely on NVIDIA CUDA technology for DNN computations.

A. CUDA Optimization Challenges

Optimizing the CUDA operations for serverless inference services is challenging as it is important to make sure the users are served correctly on time. In order to make a robust and efficient serverless inference system on edge, the following challenges need to be addressed:

C1: Retaining model correctness: As a service provider, it is important to ensure that the user-provided DNN models perform *exactly* the way it is meant to be. As a result, DNN model architecture-level optimizations are limited as techniques like layer merging and mixed precision execution are not feasible because they will impact the correctness of the individual DNN models.

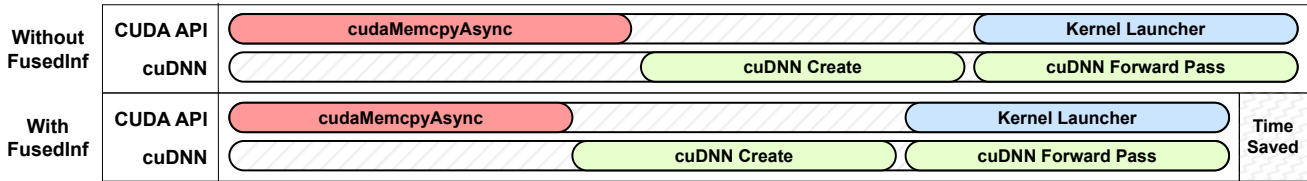


Fig. 2. DNN model execution timeline on a GPU.

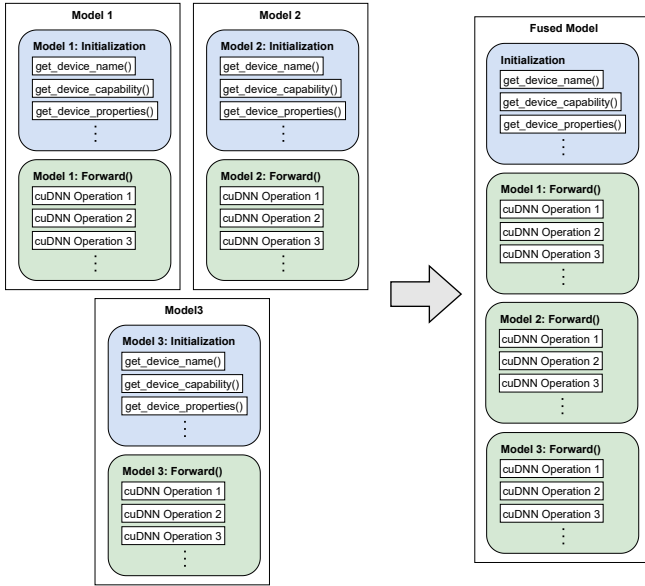


Fig. 3. Efficiency in function invocation when compiling a single DAG of multiple models.

C2: Compatibility with model variations: It is expected that a service provider will be receiving requests from a wide variety of users—each querying different DNN models with different weights. As such, optimizations like operator fusion among different models will not be possible as it requires the DNN models to have the same layer architecture, weights, and input shape.

C3: Minimal overhead: A serverless inference system performs queries on a DNN model for a limited amount of time before the models need to be swapped to serve another set of users. This swapping operation might need to be performed 100-1000 times a day depending on the users' demand and traffic load for a specific edge box. As such, any optimization that incurs a significant overhead cannot be employed in a serverless inference service on edge.

C4: Ensuring privacy: When performing inference on a DNN model using user input data, ensuring privacy is essential. Optimizations like operator fusion and layer merging can leak data from one model to another—violating user privacy and leaving a massive security vulnerability in the system.

B. Motivation

In order to build an effective system that can swap models efficiently on edge devices, we need to address the challenges discussed in the previous section. When executing multiple

DNN models, the number of core DNN operations (operations on each layer) should remain the same to ensure that each model produces the expected output. For example, the number of convolution operations and linear operations should remain the same when executing different computer vision DNN models. However, there are other function calls being made when loading the models into the GPU memory. Functions that are responsible for loading the libraries, initiating the models, configuring the devices, etc. As such, optimizations can be done when loading the models by eliminating a few of the redundant function calls. Subsequently, these optimizations can be generalized to all types DNN models since every DNN model initialization follows a similar set of function calls. As a result, there is no need for the models to be similar in order to optimize the model execution. Figure 3 presents an overview of the function calls when initializing multiple DNN models. If a single DAG is compiled with multiple model architectures, the functions needed to initialize the model will be called only once. This should make the model initialization process more efficient, which is crucial as the DNN models need to be initialized every time when swapping models. Moreover, segmented memory allocation could be inefficient when initializing models separately. This process can also be facilitated if a single DAG is compiled and initialized. So, faster memory allocations should also be possible by compiling the graphs efficiently. Finally, these should result in higher throughput—allowing more data to be moved in a given time. All of these optimizations should make model swapping more efficient on an edge device.

IV. SYSTEM DESIGN

In this section, we first discuss the overall design of *FusedInf* and how it operates on an edge node for serverless inference. Next, we discuss how *FusedInf* optimizes the CUDA operations while addressing the optimization challenges for efficient swapping of models on edge devices.

A. System Architecture

We developed a prototype of *FusedInf*, which can be deployed on commercial edge AI boxes. This framework is expected to be deployed on edge boxes handling thousands of requests a day querying a wide variety of DNN models. The framework has the following components:

DNN model repository: The DNN model repository stores the model architectures and the trained weights for all the DNN models on that particular edge AI computing box. User-registered DNN models are offloaded to the closest edge AI box and stored in the DNN model repository for performing

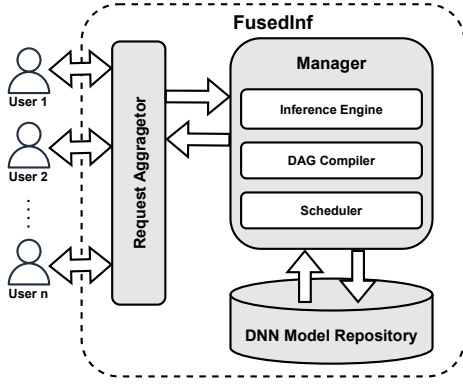


Fig. 4. *FusedInf* system architecture.

inference in the future. Since the framework is for serverless inference services on edge, DNN model architectures and the associated parameters need to be loaded on the GPU memory whenever there’s a request from the user of that particular model. Along with the DNN model architectures and weights, the model repository also stores the GPU memory requirement and the inference latency for each DNN model.

Request aggregator: The request aggregator validates and aggregates all the upcoming valid queries from different users and forwards them to the manager. Considering it is deployed in a highly demanding scenario, the request aggregator keeps aggregating the upcoming requests while the system is occupied processing current requests.

Manager: Manager is the core of the framework. It is responsible for controlling the entire system. It has three sub-components: a DAG compiler, an inference engine, and a scheduler. Depending on the aggregated requests, the manager determines how many models the DAG compiler can compile into a DAG. Since the DNN model repository stores the memory requirement information for each DNN model, the manager can estimate the number of DNN models to compile depending on the available GPU memory on the system. Once it is determined, the DAG compiler retrieves the DNN model architectures and weights from the DNN model repository and compiles the graph as demonstrated in Figure 5. Subsequently, the compiled DAG is forwarded to the inference engine where it utilizes the GPU for running the inferences using Algorithm 1.

FusedInf is vertically scalable with multiple GPUs. The Manager is capable of compiling and scheduling multiple DAGs at a time, keeping the resources occupied. When scheduling, *FusedInf* takes the model uptime into account and groups the models with short-term requests together. For example, models that require a single inference will be grouped together while models that require longer runtime will be grouped separately when compiling multiple DAGs. *FusedInf* is also capable of dynamically alerting a compiled DAG to swap a sub-graph if needed. When a sub-graph within the DAG requires swapping, it can be recompiled efficiently

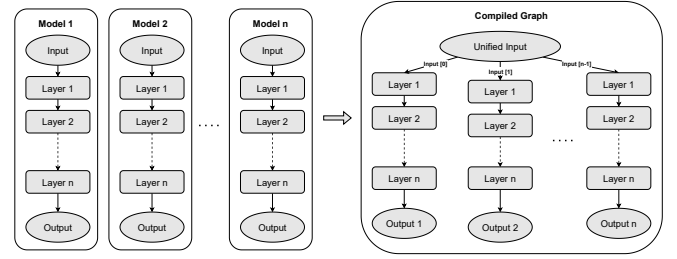


Fig. 5. Graph compilation process of *FusedInf* with multiple DNN models.

with the remaining sub-graphs.

The framework facilitates the swapping of DNN models on an edge device. Subsequently, *FusedInf* is suitable for scheduling on resource-constrained edge devices. For example, when an edge box is required to run more DNN models than it can fit in its GPU memory for a long period of time, *FusedInf* can batch the DNN models and efficiently swap the DAGs periodically. For long-term executions of multiple DNN models, the models are grouped into batches such that each batch can fit into the GPU memory. Subsequently, each batch of models is compiled into a corresponding DAG. The scheduler runs a fixed number of iterations (set by the service provider) and swaps it with the next DAG—following a round-robin scheduling algorithm.

Algorithm 1 *FusedInf* Manager

Input: DNN models $\{m_1, m_2, \dots, m_n\} \in \mathcal{M}$, inputs $\{x_1, x_2, \dots, x_n\} \in \mathcal{X}$
Output: Predictions \mathcal{Y}
Initialization: $\mathcal{G} \leftarrow \emptyset, \mathcal{Y} \leftarrow \emptyset$
1: **for** $\forall m \in \mathcal{M}$ **do**
2: **for** $u_i, u_j \in m$ **do**
3: $\mathcal{G} \leftarrow \{u_i^m\}_{i=1}^n, \{u_i^m, u_j^m\}, \{\phi_i^m\}_{i=1}^n$
4: **for** $\forall x^m \in \mathcal{X}$ **do**
5: **for** $\forall u \in \mathcal{G}$ **do**
6: $\mathcal{Y} \leftarrow \phi_i^m(\sum_{j=1}^{L_i} u_{i,j}^m x_{i,j}^m)$
return \mathcal{Y}

B. *FusedInf* Optimization Techniques

FusedInf addresses the aforementioned challenges to achieve efficiency when swapping DNN models on an edge AI computing device using the following techniques.

Single process execution: When executing multiple DNN models at a time, creating separate processes is inefficient. When separate processes are invoked for each DNN model, they all require individual loading of essential libraries to execute the model in a GPU. As a result, it creates a significant overhead when loading a DNN model into GPU memory in terms of execution time and memory consumption. CUDA Multi-Process Service (MPS) is supposed to help with the process; however, it is still inefficient and unreliable [19]. *FusedInf* tackles this problem by creating a single process for all the models—eliminating the redundancy of loading the necessary libraries separately for each process.

TABLE III
BREAKDOWN OF DIFFERENT CUDA FUNCTIONS AND THEIR EXECUTION TIME WHEN INITIALIZED 7 DNN MODELS.

CUDA Function	Time without <i>FusedInf</i>	Time with <i>FusedInf</i>	Decrease
cuDeviceGet	740 ns	500 ns	32.4%
cuDeviceGetCount	921 ns	701 ns	23.9%
cuDriverGetVersion	251 ns	100 ns	60.2%
cudaGetDevice	4.12 ms	4.01 ms	2.7%
cudaGetDeviceCount	440 ns	390 ns	11.4%
cudaMalloc	443.8 ms	40.1 ms	91.0%
cudaMemcpyAsync	2.964 s	2.759 s	6.9%
cudaSetDevice	2.54 ms	2.50 ms	1.6%
cudaStreamIsCapturing	75.40 ms	72.36 ms	4.0%

Faster model initialization: When initializing multiple DNN models, some CUDA functions are redundantly called for each model. For example, functions like *cuDeviceGet()* and *cudaGetDevice()* are used to get device information and architecture compatibility. Other functions like *get_schema()* are called to fetch the schema of the model being initiated, *cudaGetDeviceCount()* is called to get the number of CUDA devices, *cuDriverGet()* to fetch driver information, etc. *FusedInf* eliminates the redundant calls by compiling and initializing a single DAG, which results in faster model initialization. Table III presents different function calls and their execution time with 7 different DNN models.

Fewer memory calls: When a DNN model is executed on a GPU, data needs to be moved from the host to the GPU. *cudaMemcpyAsync()* is a function in the CUDA Runtime API that allows transfers of data between host and device memory asynchronously. This enables the program to continue execution while the data transfer happens in the background, potentially improving overall performance. The function can be optionally linked to a specific CUDA stream, which helps manage the order of data transfers on the GPU. *FusedInf* optimizes this function call by compiling a single DAG of multiple models—resulting in fewer calls of this function. As a result, it facilitates the loading of models on edge devices.

Efficient memory allocation: *cudaMalloc()* is a function used in CUDA programming to allocate memory on the GPU. This function allows requests for a specific amount of space in GPU memory and then provides a pointer to that memory location. This pointer can then be used to transfer data to the GPU memory and perform computations on that data. By compiling a single DAG, *FusedInf* makes this memory allocation significantly faster. Our experiment with 7 DNN models suggests that *FusedInf* can make this operation 90% quicker and makes 12% fewer calls that save memory.

Higher throughput: By compiling a single DAG, *FusedInf* achieves higher throughput. Our experiment with 7 DNN models showed 1.44 GiB/s higher throughput. Figure 2 presents the CUDA operation timeline of DNN model execution on a GPU.

C. How FusedInf Addresses the Challenges

Adopting the aforementioned optimizations, *FusedInf* can facilitate the DNN model swapping operation—allowing a service provider to serve more users per day in highly demanding scenarios.

- *FusedInf* addresses the challenge of retaining model correctness (C1) by not altering the model architectures when compiling the DAG. The compiled DAG consists of sub-graphs of each model exactly the way a user had provided containing the exact number of DNN operators. As a result, the output of each model remains the same.
- The DAG compiler can compile any DNN model architecture, which ensures compatibility with model variations (C2). Section V presents the evaluation of *FusedInf* with a wide variety of model architectures to show the compatibility and adaptability of *FusedInf* in different applications. Moreover, *FusedInf* works with all types of DNNs since it does not depend on the individual model architecture or architectural similarities across fused models. It is designed to optimize CUDA functionality when initializing multiple models. Fundamentally, every DNN model initialization follows a similar set of function calls. As a result, the speed-up does not depend on any specific combination of models, and we have not noticed any slowdown with any model combinations.
- *FusedInf* leverages fast and efficient DAG compilation for minimal overhead. Moreover, it makes fewer memory calls, faster memory allocation, and achieves higher throughput. As a result, it addresses the challenge of achieving a minimal overhead (C3).
- Finally, *FusedInf* does not fuse or merge operators from models across. Each sub-graph processes its own input without sharing outputs from the layers, which addresses the challenge of ensuring privacy (C4).

V. EVALUATION

In order to evaluate our system, we selected a few popular DNN models. We picked 34 different models from 13 different model families. We ran our evaluations in three phases. In phase one, we started with one model and gradually increased the number of models up to 7 to see the impact of combining models. In the next phase, we swapped 5 models randomly for 10 different consecutive test cases. For both phases, we ran 100 iterations for each model and collected the peak GPU memory consumption and the total execution time. The execution time metric was an average of 5 runs, which included the model loading time and the time to run 100 iterations. In the third phase, we evaluated the dynamic sub-graph swapping capability of *FusedInf* by swapping a sub-graph after a certain iteration for 5 different sub-graphs in a DAG. The system used for evaluation consisted of an AMD Ryzen Threadripper 5955WX and an NVIDIA RTX 4090. Table IV presents the models used for different test cases for the second phase along with their ImageNet-1K accuracy. Following are the brief descriptions of the models used for evaluation.

TABLE IV
DNN MODELS USED FOR DIFFERENT TEST CASES.

Model		Accuracy		Test Cases									
		Top 1	Top 5	1	2	3	4	5	6	7	8	9	10
AlexNet	AlexNet	56.522	79.066		X	X					X		X
VGG	VGG-11	69.02	88.628				X						X
	VGG-13	69.928	89.246										X
	VGG-16	71.592	90.382								X		
	VGG-19	72.376	90.876		X					X			X
	VGG-11 with batch normalization	70.37	89.81								X		
	VGG-13 with batch normalization	71.586	90.374			X							
	VGG-16 with batch normalization	73.36	91.516	X									
ResNet	VGG-19 with batch normalization	74.218	91.842				X						
	ResNet-18	69.758	89.078	X						X			
	ResNet-34	73.314	91.42			X	X	X					X
	ResNet-50	76.13	92.862		X					X			
	ResNet-101	77.374	93.546								X	X	
SqueezeNet	ResNet-152	78.312	94.046				X				X		
	SqueezeNet 1.0	58.092	80.42								X		
DenseNet	SqueezeNet 1.1	58.178	80.624					X					
	Densenet-121	74.434	91.972										
	Densenet-169	75.6	92.806										
	Densenet-201	76.896	93.37			X							
Inception	Inception v3	77.294	93.45										
GoogLeNet	GoogLeNet	69.778	89.53										
ShuffleNet	Densenet-161	77.138	93.56	X									
	ShuffleNet V2 x1.0	69.362	88.316			X	X						
MobileNet	ShuffleNet V2 x0.5	60.552	81.746					X					
	MobileNetV2	71.878	90.286	X				X	X				X
	MobileNet V3 Large	74.042	91.34		X								X
ResNeXt	MobileNet V3 Small	67.668	87.402										
	ResNeXt-50-32x4d	77.618	93.698										X
	ResNeXt-101-32x8d	79.312	94.526							X	X		
Wide ResNet	Wide ResNet-50-2	78.468	94.086		X				X				X
	Wide ResNet-101-2	78.848	94.284					X					
MNASNet	MNASNet 1.0	73.456	91.51								X	X	
	MNASNet 0.5	67.734	87.49										
EfficientNet	EfficientNet V2 Large	85.808	97.788	X									

We used VGG models [5], short for Visual Geometry Group models, which are a family of convolutional neural networks (CNNs) known for their simplicity and effectiveness in image recognition tasks. We used AlexNet [23]—named after Alex Krizhevsky, which is a convolutional neural network (CNN) architecture that revolutionized image recognition in 2012. We also picked ResNet [24] (short for Residual Neural Network)—a deep learning architecture specifically designed to address the vanishing gradient problem that can hinder training in very deep neural networks. Furthermore, we picked SqueezeNet [25], which is designed for efficiency. Unlike AlexNet and VGG models with their numerous layers, SqueezeNet achieves AlexNet-level accuracy for image classification with significantly fewer parameters. Next, we picked DenseNets [7], which are a type of CNN architecture known for their efficient use of parameters and strong feature propagation. We also selected GoogLeNet developed by researchers at Google and InceptionNet [26], which was built upon the success of GoogLeNet with more complex CNN architectures utilizing the Inception module as a core component. ShuffleNet [27] was also picked, which is a convolutional neural network architecture specifically designed for deployment on mobile and other resource-constrained devices. Next, we picked MobileNet [6], which is a lightweight convolutional

neural network architecture designed for mobile and embedded devices. Furthermore, we selected ResNeXt [28] that builds upon the success of ResNet (Residual Network) architecture, introducing a new concept called cardinality. Wide ResNet (WRN) [29] was another family we picked, which is a variant of the popular ResNet architecture specifically designed to address limitations associated with very deep networks. For automatic neural network architecture search applications on mobile devices, we picked MNASNet [30]—Mobile Neural Architecture Search Net. Finally, we picked EfficientNet [8], which is a family of convolutional neural networks (CNNs) designed to achieve a balance between accuracy and efficiency.

Baseline (without *FusedInf*): For comparison, we adopted a baseline that executes multiple DNN models with a single script. From our observation, we realized that running separate processes for each model in the GPU is extremely inefficient because each process invokes a separate CUDA Context. Each CUDA Context takes about 500 MiB of additional GPU memory, consuming extra time and memory for each model. In order to present a fair comparison, we initiated multiple DNN models from a single script, which invokes a single CUDA Context for all of the models—ensuring optimal GPU memory consumption. Moreover, the script contains a trigger that checks for pending requests in the request queue. As soon

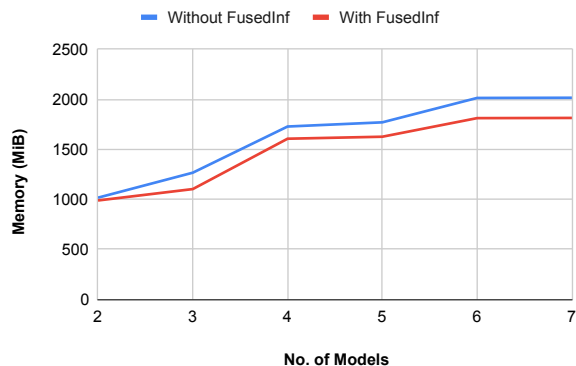


Fig. 6. GPU memory consumption of different model combinations from 1 to 7 for 100 iterations.

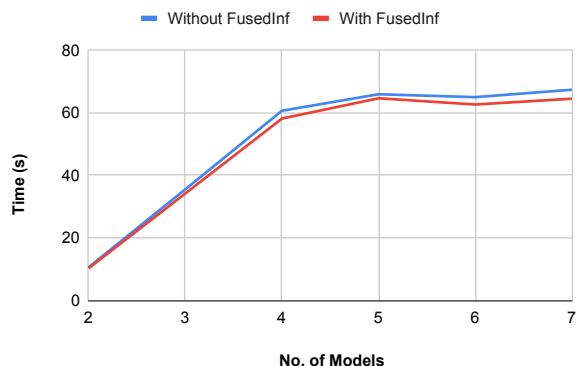


Fig. 7. Total execution time of different model combinations from 1 to 7 for 100 iterations.

as a model finishes its execution, the script checks for pending requests in the queue and initiates a new model if there is enough GPU memory. The Request Aggregator is responsible for aggregating requests from the users and putting them in the queue.

Profiling Tools: We used 2 different profiling tools to profile and collect data: Nvidia Nsight System and Pytorch Profiler.

A. Impact on Time and Memory with Different Number of Models

In the first phase, we experimented with 7 DNN models. We gradually increased to 7 DNN models starting from 1 DNN model. As such, we ran 7 different experiments for this phase, each with 100 iterations per model. The size of the inputs for the models was (3, 224, 224)–images of 3 channels with a dimension of 224×224 . Initially, we picked the VGG16 model with batch normalization. When we ran 100 iterations of this single model, it consumed 954 MiB of GPU memory and the total time was 4.3 seconds. Next, we ran VGG16 with MobileNetV3 large. Without *FusedInf*, the two models peaked at 1016 MiB in GPU memory usage and the total execution time was 10.57 seconds. For the two models with *FusedInf*, the peak GPU memory usage was 988 MiB and the total execution time was 10.26 seconds. *FusedInf* was able to save 28 MiB (2.8% less) of GPU memory and 0.31 seconds (3% less) of execution time with two DNN models. Afterward, we

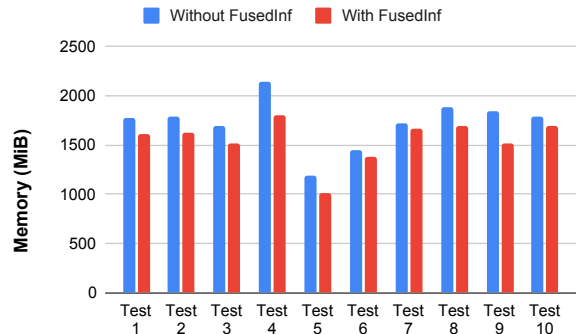


Fig. 8. GPU memory consumption of 5 randomly picked models for 10 different test cases after 100 iterations each.

ran VGG16, MobileNetV3Large, and DenseNet161 together. With these three models, the peak GPU memory usage was 1266 MiB without *FusedInf* and 1102 MiB with *FusedInf*. This time, *FusedInf* saved 164 MiB of GPU memory and 1.33 seconds of execution time with three DNN models. In the next run, we ran VGG16, MobileNetV3Large, DenseNet161, and EfficientNetV2Large together. This run peaked at 1728 MiB in GPU memory usage and 1606 MiB GPU memory usage without and with *FusedInf*, respectively. In terms of total execution time, the time was 60.71 seconds and 58.2 seconds, respectively. As such, *FusedInf* saved 122 MiB (7.6% less) of GPU memory and 2.51 seconds (4.3% less) of execution time with 4 DNN models. For the fifth test, we included ResNet18 with the rest of the 4 DNN models—running VGG16, MobileNetV3Large, DenseNet161, EfficientNetV2Large, and ResNet18 at the same time. When the system was running there 5 DNN models, the peak GPU memory usage was 1770 MiB without *FusedInf* and 1626 MiB with *FusedInf*, which saved 144 MiB of GPU memory (a reduction of 9%). In terms of execution time, it took 66 seconds without *FusedInf* and 64.7 seconds with *FusedInf*—a reduction of 1.29 seconds (2% less) with 5 DNN models. The following run included AlexNet to the rest of the 5 DNN models, making a combination of 6 DNN models. For this run, the GPU memory consumption was 2014 MiB without *FusedInf* and 1812 MiB with *FusedInf*. As such, *FusedInf* saved 202 MiB of GPU memory (11.1% less). When running 6 DNN models at the same time, the execution time without *FusedInf* was 65.06 seconds, and with *FusedInf*, it was 62.7 seconds—reducing the time by 2.36 seconds (3.8% less) for 6 DNN models. Finally, we ran 7 DNN models (VGG16, MobileNetV3Large, DenseNet161, EfficientNetV2Large, ResNet18, AlexNet, and SqueezeNet). Without *FusedInf*, the 7 DNN models used 2016 MiB of GPU memory, and the execution time was 67.44 seconds. With *FusedInf*, the GPU memory usage went down to 1812 MiB and the execution time was 64.56 seconds. As such, *FusedInf* was able to save 202 MiB of GPU memory (11.1% less), and the execution time was 2.88 seconds quicker (4.5% less). Figure 8 and 9 present the GPU memory consumptions and total execution time of this phase, respectively.

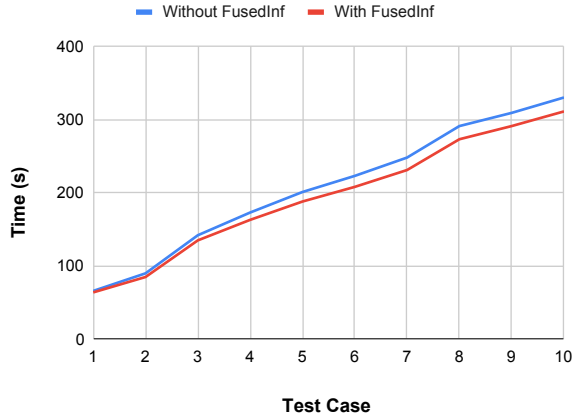


Fig. 9. Total execution time of 5 randomly picked models for 10 different test cases after 100 iterations each.

B. Impact on Time and Memory with Consecutive Swaps

In this next phase of experiments, we randomly picked 5 models out of the 34 DNN models belonging to 13 different model families. We ran 10 consecutive test cases randomly picking 5 DNN models each time and swapping them. This evaluation presents the effectiveness of the scheduler that swaps out DAGs after a fixed number of interactions (100 in this case).

We collected the same metrics as we did for the first phase. The first test case was inducted with AlexNet, VGG16 (with batch normalization), ResNet18, DenseNet161, MobileNetV2, and EfficientNetV2 Large. This test case demonstrated a reduction of 164 MiB in peak GPU memory usage (9.27% less) and 2 seconds faster execution (3.13% less). The second test case was inducted with AlexNet, VGG11, VGG19, ResNet50, MobileNetV3 Large, and Wide ResNet-50-2. This test case demonstrated a reduction of 160 MiB in peak GPU memory usage (8.96% less) and 3 seconds faster execution (14.3% less). In the third case, AlexNet, VGG13 (with batch normalization), ResNet34, Densenet-201, and ShuffleNet V2 x1.0 were picked randomly. This test case demonstrated a reduction of 172 MiB in peak GPU memory usage (10.17% less) and 2 seconds faster execution (4% less). The fourth test case was randomly picked VGG-11, VGG-19 (with batch normalization), ResNet-34, ResNet-152, and ShuffleNet V2 x1.0. This test case demonstrated a reduction of 344 MiB in peak GPU memory usage (16.03% less) and 3 seconds faster execution (10.71% less). In the fifth case, we randomly picked ResNet-34, SqueezeNet 1.1, ShuffleNet V2 x0.5, MobileNetV2, and Wide ResNet-101-2. This test case demonstrated a reduction of 174 MiB in peak GPU memory usage (14.6% less) and 3 seconds faster execution (12% less). The sixth test case included VGG19, ResNet-18, ResNet-50, MobileNetV2, and Wide ResNet-50-2. This test case demonstrated a reduction of 66 MiB in peak GPU memory usage (4.56% less) and 2 seconds faster execution (10% less). The seventh case was inducted AlexNet, VGG11 (with batch normalization), ResNet-101, SqueezeNet 1.0, and ResNeXt-101-32x8d. This

test case demonstrated a reduction of 58 MiB in peak GPU memory usage (3.36% less) and 2 seconds faster execution (8.7% less). The eighth case included VGG16, ResNet-101, ResNet-152, ResNeXt-101-32x8d, and MNASNet 1.0 randomly. This test case demonstrated a reduction of 198 MiB in peak GPU memory usage (10.47% less) and 1 second faster execution (2.38% less). The following test case was conducted with AlexNet, VGG11, ResNet-34, Wide ResNet-50-2, and MNASNet 1.0. This test case demonstrated a reduction of 324 MiB in peak GPU memory usage (17.63% less) but about the same execution time. In the final test, VGG13, VGG19, MobileNetV2, MobileNet V3 Large, and ResNeXt-50-32x4d were randomly chosen. This test case demonstrated a reduction of 92 MiB in peak GPU memory usage (5.14% less) and 1 second faster execution (5% less).

Figure 8 and 9 presents the peak GPU memory consumption and the total execution time for the consecutive test cases, respectively. On average, *FusedInf* saves 2 seconds per 30 seconds of operation executing 5 models. After 10 consecutive runs with 5 different models, the total execution time with *FusedInf* was 311 seconds, compared to 330 seconds without *FusedInf*. Following this trend, service providers will be able to save about 3 hours and 12 minutes per day, allowing them to run approximately 2000 more models per day.

C. Impact on Time and Memory with Individual Model Swaps in a DAG

In the final phase, we evaluated the sub-graph (model) swapping feature of *FusedInf*. We started with 5 different models and swapped each after completing the required iterations. This evaluation presents the effectiveness of the dynamic adaptive DAG compiler. *FusedInf* is capable of compiling a part of the DAG without the need to recompile the entire DAG.

The initial DAG contained VGG-19 with Batch Normalization, ResNet-50, MobileNet V3 Large, ResNeXt-50 (32x4d), and MNASNet 1.0. The first 25 iterations were 0.8 seconds faster (3.42%) consuming 198 MiB less (16.69%) GPU memory. After the 25th iteration, we swapped VGG-19 with EfficientNet where *FusedInf* was ahead by 2.07 seconds (3.13%) consuming 154 MiB less memory (13.82%). After 50 iterations, we swapped ResNet-50 with Inception v3, where *FusedInf* was ahead by 4.91 seconds (4.08%) consuming 152 MiB less memory (13.67%). After 75 iterations, we swapped ResNeXt-50-32x4d with SqueezeNet 1.1, where *FusedInf* was ahead by 5.7 seconds (3.36%) consuming 154 MiB less memory (15.01%). After 100 iterations, we swapped MobileNet V3 Large with VGG-16, where *FusedInf* was ahead by 7.71 seconds (3.58%) consuming 152 MiB less memory (9.95%). Finally, after 125 iterations, we swapped MNASNet 1.0 with Wide ResNet-101-2, and ran 25 iterations, where *FusedInf* was ahead by 8.76 seconds (3.28%) consuming 152 MiB less memory (7.54%).

Table V presents the results after each swap. The GPU memory presents the peak consumption after each swap. The time presented is the total execution time after each swap. From the results, we can see that *FusedInf* was able to save

TABLE V
MEMORY AND TIME OF DIFFERENT SWAPS WITHIN A DAG.

Models in DAG	Swap 1	Swap 2	Swap 3	Swap 4	Swap 5	Swap 6	
	VGG-19 with BN	<i>EfficientNet V2</i>	<i>EfficientNet V2</i>	<i>EfficientNet V2</i>	<i>EfficientNet V2</i>	<i>EfficientNet V2</i>	<i>EfficientNet V2</i>
	ResNet-50	ResNet-50	<i>Inception v3</i>	<i>Inception v3</i>	<i>Inception v3</i>	<i>Inception v3</i>	
	MobileNet V3 L	MobileNet V3 L	MobileNet V3 L	MobileNet V3 L	<i>VGG-16</i>	<i>VGG-16</i>	
	ResNeXt-50-32x4d	ResNeXt-50-32x4d	ResNeXt-50-32x4d	<i>SqueezeNet 1.1</i>	<i>SqueezeNet 1.1</i>	<i>SqueezeNet 1.1</i>	
MNASNet 1.0	MNASNet 1.0	MNASNet 1.0	MNASNet 1.0	MNASNet 1.0	<i>WideResNet-101</i>		
Memory without <i>FusedInf</i> (MiB)	1384	1268	1264	1180	1680	2168	
Memory with <i>FusedInf</i> (MiB)	1186 (-16.69%)	1114 (-13.82%)	1112 (-13.67%)	1026 (-15.01%)	1528 (-9.95%)	2016 (-7.54%)	
Time without <i>FusedInf</i> (s)	24.23	68.18	125.16	175.14	223.25	275.66	
Time with <i>FusedInf</i> (s)	23.43 (-3.41%)	66.11 (-3.13%)	120.25 (-4.08%)	169.44 (-3.36%)	215.54 (-3.58%)	266.9 (-3.28%)	

an average of 160 MiB of memory (12.05%) and a total of 8.76 seconds (3.28%) of total execution time.

VI. DISCUSSION

FusedInf is designed to facilitate DNN model swapping on resource-constrained edge boxes. From the evaluation results, we can see that *FusedInf* is capable of efficiently swapping DNN models in various use cases. The evaluation in Section V-A demonstrates that the greater the number of models, the more efficient *FusedInf* can be in terms of execution time and memory consumption. The evaluation in Section V-A shows that the total execution time with *FusedInf* was 19 seconds faster after 10 consecutive runs with 5 different models running for 100 iterations per cycle. This translates to 3 hours and 12 minutes saved per day, allowing a service provider to perform approximately 2000 more swaps per day, which is a significant number. Finally, Section V-C demonstrates the efficiency of swapping individual DNN models within a compiled DAG. *FusedInf* can not only efficiently compile a DAG of multiple models but also recompile a DAG efficiently by swapping a particular sub-graph (model) from the DAG.

VII. RELATED WORK

There are a few production-level inference systems offered by popular machine learning frameworks. TorchServe is one of the most popular inference engine services provided by PyTorch [31]. TorchServe is a flexible and high-performance tool designed specifically for serving PyTorch deep learning models in production environments. TorchServe provides a built-in web server that allows applications to make predictions using the deployed model through a set of easy-to-use REST APIs. TensorFlow serving is another tool by TensorFlow [32] for deploying inference engines on the edge. It is a software library designed specifically for deploying machine learning

models trained with TensorFlow in production environments on the edge. Moreover, it provides both RESTful API and gRPC interfaces for clients to interact with the models, making it accessible to various edge development environments. ONNX, short for Open Neural Network Exchange, functions as an open-source format for representing machine learning models. It acts as a common language, enabling a seamless exchange of models between different deep-learning frameworks. This interoperability empowers developers to train models in their preferred framework (like TensorFlow or PyTorch) and then deploy them on various edge platforms or runtimes that support ONNX [33]. Clipper is a low-latency online prediction serving system proposed by Crankshaw et. al. [34]. Clipper is a system designed to take machine learning models from various frameworks and optimize their performance for real-world use. It sits between applications and the models, simplifying deployment and using techniques like caching and batching to deliver predictions faster and more accurately.

Popular cloud service providers are also allowing users to deploy and run DNN models on their cloud or edge infrastructure by providing their own services. AWS SageMaker [35] is a cloud-based platform offered by Amazon Web Services (AWS) specifically designed to streamline the ML workflow. It simplifies the process of building, training, deploying, and managing ML models on the edge. Azure Machine Learning (Azure ML [36]) is a cloud-based service offered by Microsoft. It streamlines the entire machine learning lifecycle, from data preparation and model training to deployment and monitoring on the edge. Vertex AI [37] by Google Cloud simplifies the AI development process on the edge by providing a central platform for data management, model training, and deployment.

In 2021, Romero et. al. proposed INFaaS [38]—an automated model-less inference serving system. INFaaS simplifies

deploying DNN models for real-time use. Instead of developers choosing specific DNN models for each task, INFaaS automatically selects the best option based on the desired performance (latency) and accuracy trade-off. Applications send their requests to INFaaS through a user-friendly interface (Front-End). The core system (Controller) then analyzes these requests and picks the most fitting model variant for the job. This chosen variant, along with the actual query, is then directed to a Worker machine. Finally, the Worker leverages the appropriate hardware components (Hardware Executors) to execute the inference task and sends the results back to the application. However, in its current implementation, a Worker is only responsible for handling queries of one DNN model, which is not very efficient for the resource-contained edge devices.

AWS SageMaker recently introduced serverless inference services [39] for deploying DNN models that automatically scale resources based on incoming requests. This eliminates the need for manual server management and is ideal for workloads with unpredictable traffic patterns, as it only allocates resources when needed. This saves costs compared to constantly running servers and simplifies deployment for small to medium-sized businesses. Vertex AI Pipeline [40] by Google Cloud offers similar functionalities. Vertex AI Pipelines eliminates the need to manage machine learning projects and allows users to build automated workflows that handle everything from training the DNN models to monitoring their performance—all without needing to manage servers.

AMPS-inf [41] is a framework developed to exploit the serverless inference services to mitigate the management and overall cost while meeting the response time. AMPS-inf achieves that by using a technique of model partitioning. This involves breaking down the model into smaller pieces that can be run independently. It then formulates a Mixed-Integer Quadratic Programming problem to determine how many partitions to split the model into and how to assign those partitions to be run on serverless functions. The appropriate resources for each serverless function (e.g., memory, CPU) are acquired by solving this problem, AMPS-Inf aims to find the most cost-effective way to run the inference tasks while still meeting the required response time.

Ali et. al. proposed a framework called BATCH [42] for latency performance and cost-effectiveness of machine learning inference. BATCH uses an optimizer to provide inference tail latency guarantees and cost optimization and to enable adaptive batching support. To meet the service level objectives BATCH adopts adaptive parameter tuning, which allows it to dynamically adjust the batching parameter based on the objectives defined by the user.

TETRIS is a serverless platform specifically designed for running deep learning inference tasks efficiently. It tackles the common problem of high memory usage in serverless environments by using a combination of techniques. TETRIS automatically shares resources like the tensors used by different inference tasks and reclaims unused memory and schedules serverless instances efficiently to minimize wasted resources. It

also ensures the required performance standards of a serverless inference system.

VIII. CONCLUSION

FusedInf is designed to efficiently load and query multiple DNN models concurrently at the same time. The purpose of *FusedInf* is to facilitate the serverless inference services on the edge in order to serve more users per day by saving time when initializing and executing DNN models. The proposed framework achieves this by efficiently compiling a single DAG of multiple DNN models, which requires fewer memory calls, accelerates memory allocation, and achieves higher throughput. As such, *FusedInf* can allow a serverless inference service provider to serve more users at a time when there is a high traffic of queries for a wide variety of different DNN models. In the future, we will experiment with more model architectures and try to employ opportunistic approaches to find the best model combination for the most efficient utilization of GPU memory and function calls.

REFERENCES

- [1] P. McEnroe, S. Wang, and M. Liyanage, "A survey on the convergence of edge computing and ai for uavs: Opportunities and challenges," *IEEE Internet of Things Journal*, vol. 9, no. 17, pp. 15 435–15 459, 2022.
- [2] M. Kamruzzaman, "New opportunities, challenges, and applications of edge-ai for connected healthcare in smart cities," in *2021 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2021, pp. 1–6.
- [3] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Amps-inf: Automatic model partitioning for serverless inference with cost efficiency," in *Proceedings of the 50th International Conference on Parallel Processing*, 2021, pp. 1–12.
- [4] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Influss: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 768–781.
- [5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
- [6] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
- [7] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.
- [8] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.
- [9] J. Yao, S. Zhang, Y. Yao, F. Wang, J. Ma, J. Zhang, Y. Chu, L. Ji, K. Jia, T. Shen et al., "Edge-cloud polarization and collaboration: A comprehensive survey for ai," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 7, pp. 6866–6886, 2022.
- [10] F. Firouzi, B. Farahani, and A. Marinšek, "The convergence and interplay of edge, fog, and cloud in the ai-driven internet of things (iot)," *Information Systems*, vol. 107, p. 101840, 2022.
- [11] Y. Wu, "Cloud-edge orchestration for the internet of things: Architecture and ai-powered data processing," *IEEE Internet of Things Journal*, vol. 8, no. 16, pp. 12 792–12 805, 2020.
- [12] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.
- [13] P. Guo, B. Hu, and W. Hu, "Mistify: Automating {DNN} model porting for {On-Device} inference at the edge," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 705–719.

- [14] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
- [15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.
- [16] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating gpu memory consumption of deep learning models," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1342–1352.
- [17] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.
- [18] A. Padmanabhan, N. Agarwal, A. Iyer, G. Ananthanarayanan, Y. Shu, N. Karianakis, G. H. Xu, and R. Netravali, "Gemel: Model merging for {Memory-Efficient}, {Real-Time} video analytics at the edge," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 973–994.
- [19] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia, "Accelerating deep learning workloads through efficient multi-model execution," in *NeurIPS Workshop on Systems for Machine Learning*, vol. 20, 2018.
- [20] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger, "Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 29–42. [Online]. Available: <https://www.usenix.org/conference/atc18/presentation/jjiang>
- [21] X. Sun, R. Panda, R. Feris, and K. Saenko, "Adashare: Learning what to share for efficient deep multi-task learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 8728–8740, 2020.
- [22] "Cisco annual internet report (2018–2023) white paper," <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>, accessed: 2024.
- [23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.
- [24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [25] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.
- [26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: <http://arxiv.org/abs/1409.4842>
- [27] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.
- [28] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.
- [29] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.
- [30] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2820–2828.
- [31] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.
- [32] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: <https://www.tensorflow.org/>
- [33] J. Bai, F. Lu, K. Zhang *et al.*, "Onnx: Open neural network exchange," <https://github.com/onnx/onnx>, 2019.
- [34] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 613–627. [Online]. Available: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw>
- [35] "Machine learning service - amazon sagemaker," <https://aws.amazon.com/sagemaker/>, accessed: 2024.
- [36] J. Barnes, "Azure machine learning," *Microsoft Azure Essentials. 1st ed, Microsoft*, 2015.
- [37] "Vertex ai - machine learning platform," <https://cloud.google.com/vertex-ai>, accessed: 2024.
- [38] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFAAS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 397–411. [Online]. Available: <https://www.usenix.org/conference/atc21/presentation/romero>
- [39] "Amazon sagemaker serverless inference," <https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html>, accessed: 2024.
- [40] "Introduction to vertex ai pipelines," <https://cloud.google.com/vertex-ai/docs/pipelines/introduction>, accessed: 2024.
- [41] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Amps-inf: Automatic model partitioning for serverless inference with cost efficiency," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3472456.3472501>
- [42] A. Ali, R. Pincioli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.