SUBGRAPH STATIONARY HARDWARE-SOFTWARE INFERENCE CO-DESIGN

Payman Behnam [*1] **Jianming Tong** [*1] **Alind Khare** [1] **Yangyu Chen** [1] **Yue Pan** [1] **Pranav Gadikar** [1]
Abhimanyu Rajeshkumar Bambhaniya [1] **Tushar Krishna** [1] **Alexey Tumanov** [1]

ABSTRACT

A growing number of applications depend on Machine Learning (ML) functionality and benefits from both higher quality ML predictions and better timeliness (latency) at the same time. A growing body of research in computer architecture, ML, and systems software literature focuses on reaching better latency/accuracy tradeoffs for ML models. Efforts include compression, quantization, pruning, early-exit models, mixed DNN precision, as well as ML inference accelerator designs that minimize latency and energy, while preserving delivered accuracy. All of them, however, yield improvements for a single static point in the latency/accuracy tradeoff space. We make a case for applications that operate in dynamically changing deployment scenarios, where no single static point is optimal. We draw on a recently proposed weight-shared *SuperNet* mechanism to enable serving a stream of queries that uses (activates) different *SubNets* within this weight-shared construct. This creates an opportunity to exploit the inherent temporal locality with our proposed *SubGraph* Stationary (SGS) optimization. We take a hardware-software co-design approach with a real implementation of SGS in **`SushiAccel`** and the implementation of a software scheduler **`SushiSched`** controlling which *SubNets* to serve and what to cache in real-time. Combined, they are vertically integrated into **`SUSHI`**—an inference serving stack. For the stream of queries **`SUSHI`** yields up to 25% improvement in latency, 0.98% increase in served accuracy. **`SUSHI`** can achieve up to 78.7% off-chip energy savings.

1 INTRODUCTION

The number of applications leveraging Machine Learning (ML) functionality continues to grow, as ML is successfully applied beyond image classification (Ovtcharov et al., 2015), object detection/recognition (Chen et al., 2017a; Ali et al., 2018), sentiment analysis (Jiang et al., 2020), and next word prediction (Sundermeyer et al., 2012). These applications are also increasingly latency sensitive. Their interactive experience depends on what fraction of prediction tasks are satisfied within the application-specified latency budget (typically in the 10-100 ms interactive latency range). Examples of such applications include self-driving cars (Gog et al., 2022), specifically the on-board software responsible for multi-modal sensory data processing, street sign detection (Tabernik & Skočaj, 2019), pedestrian detection (Liu et al., 2019), vehicle trajectory tracking (Deo & Trivedi, 2018), lane tracking (Datta et al., 2020), and Intensive Care Unit stability score prediction (Hong et al., 2020). These applications require the ability to serve trained ML models in a way that maximizes the fraction of queries completed within the application specified latency budget—defined as latency Service Level Objective (SLO) attainment. A unifying characteristic for this class of applications is that they simultaneously care about the quality (accuracy) and timeliness (latency) of ML inference served.

There has been a body of work successfully improving achievable latency/accuracy tradeoffs for specific Deep Learning models. Examples include multiple forms of quantization (Bai et al., 2018; Zhang et al., 2018; Pouransari et al., 2020; Fang et al., 2020), mixed DNN precision (Abdelaziz et al., 2021), compression (Iandola et al., 2016), pruning (Liu et al., 2018), latency-aware neural architecture search (Cai et al., 2018; Eriksson et al., 2021), just to name a few. However, fundamentally, all of these techniques optimize for a *single static* point in the latency/accuracy tradeoff space. Indeed, for a given deployment device, the outcome is typically a single static model that has a specific (latency, accuracy) tuple associated with it. We claim this is no longer sufficient.

We observe that the applications with acute latency-accuracy sensitivity typically operate in *dynamically* variable deployment conditions. These include variable query traffic patterns (e.g., variable number of patients triaged in the ICU or ER), on-device battery power level (e.g. bed-side compute

[*]Equal contribution [1]Georgia Institute of Technology, Atlanta, Georgia, USA. Correspondence to: Payman Behnam, Jianming Tong <payman.behnam@gatech.edu, jianming.tong@gatech.edu>.

Published as a Conference Paper at MLSys'23.

or battery-powered edge device), and query complexity (e.g., autonomous vehicle (AV) navigation of sparse suburban vs dense urban terrain).

Under such variable deployment conditions, a choice of *any* single static model from the latency/accuracy tradeoff space will be suboptimal. Indeed, a higher accuracy model may result in dropped queries during periods of transient overloads. The lower accuracy model may yield suboptimal prediction quality under low load—both unnecessarily under-performing. Inherently, the ideal solution would include dynamically picking a "best-fit" model from the latency/accuracy tradeoff space. For a specific latency constraint that varies over time, a just-in-time choice of the highest accuracy model satisfying this constraint is preferred. Thus, the ability to switch (or navigate) between points in the latency/accuracy tradeoff space in real-time is intuitively required for such applications.

We identify one such mechanism that enables this — weight-shared *SuperNets* (Cai et al., 2019) (§2.1). This neural network construct consists of multiple convolutional neural networks (CNNs) sharing common model parameters. It simultaneously encapsulates "deep and thin" models as well as "wide and shallow" within the same structure without weight duplication. These *SuperNets* can be used to activate different *SubNets* without explicitly extracting them into different independently stored models. This is highly efficient from the systems perspective, as it obviates the need to store these model variants separately (saving memory cost), and enables rapidly switching *SubNets* that are "activated" to serve different incoming queries.

On the hardware end, the need for real-time inference has led to a plethora of ML accelerators. A key optimization technique (e.g., "dataflow" (Chen et al., 2016)) leveraged by most accelerators involves *reusing* activations and/or weights across multiple computations, leading to architectures that can be classified as weight stationary, output stationary, input stationary, row stationary, and hybrid variations of these (Chen et al., 2016). These dataflows rely on neural network layers, specifically 2D convolutions, to be compute-bound. One challenge of serving *SubNets* with diverse shapes, however, as we identify, is the memory-bound nature of some of the *SubNets* (smaller FLOPS/Byte).

To address this challenge, we make a key observation that the weight-shared *SuperNet* mechanism inherently results in queries activating commonly shared *SubGraph*s within the same *SuperNets* structure[1]. Furthermore, we note a significant amount of *temporal locality* in the weights of the *SuperNets* re-used *across* queries. We identify this as an opportunity for a new kind of data reuse, which we name *SubGraph* Stationary (SGS) optimization—a technique we haven't seen used or proposed by any existing accelerator. We realize the benefits of SGS by implementing hardware caching support for weight reuse at the granularity of neural network *SubGraphs*.

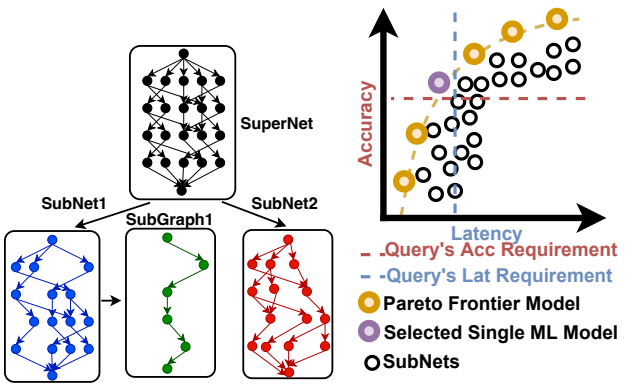
In addition to SGS-aware hardware implementation, we co-design an SGS-aware query scheduler that decides (a) which *SubNets* to activate for each query and (b) which *SubGraphs* to cache. We propose an algorithmic approach to make these control decisions based on (a) a query's specified accuracy constraint and (b) the current state of the accelerator (which we abstract). We demonstrate that these control decisions benefit from hardware state awareness, as baseline state-unaware caching leaves room for improvement. Finally, we propose an abstraction that enables the query scheduling policy to generalize, while remaining accelerator state-aware. The abstraction is captured by a black-box table (Fig. 4) that exposes the latency of activating a *SubNet* i as a function of a currently cached *SubGraph* j. We instantiate the concept of *SubGraph* Stationary (SGS) cross-query optimization in our vertically integrated inference serving stack, **SUSHI**, which includes (a) **SushiAccel**—a real FPGA implementation of hardware support for SGS-aware weight-shared *SuperNet* inference, and (b) **SushiSched** to make real-time control decisions on a stream of queries executed on **SushiAccel**, sequentially deciding for each query *SubNet* i to activate and (periodically) *SubGraph* j to cache on the accelerator. **SushiAccel** and **SushiSched** combined in **SUSHI** enable *agile* navigation of the latency/accuracy tradeoff space, reaching better latency/accuracy tradeoffs by leveraging the key property of "cross query" temporal locality inherent to weight-shared *SuperNets* with what we believe to be the first hardware-software co-design for weight-shared inference.

The key contributions of this paper can be summarized as follows:

- a concept of *SubGraph* Stationary (SGS) approach for hardware acceleration of DNN inference on weight-shared *SuperNets*.
- **SushiAccel**—a real SGS-aware FPGA implementation, with a simulator and design space exploration tools.
- **SushiSched**—a software query scheduler that operates in SGS-aware fashion, controlling which *SubNets* to activate and *SubGraphs* to cache in real time.
- **SUSHI**—a hardware-software co-designed inference serving stack, vertically integrating **SushiAccel** and **SushiSched**.
- **SushiAbs**—an abstraction that generalizes SGS-aware query scheduling to arbitrary accelerators, while retaining implicit accelerator state awareness.

Combined, **SUSHI** is able to achieve up to 25% query serv-

[1]We define *SubGraph* as a subgraph consisting of any subset of weights from the *SuperNets* connected together into a graph



(a) *SubNets* & *SubGraphs* concepts. (b) Latency-Acc. tradeoff

Figure 1. WS-DNN properties.

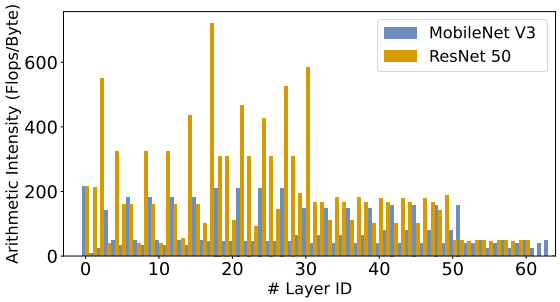


Figure 2. Arithmetic intensity for different layers of various DNNs. Lower arithmetic intensity leads to relatively higher *memory* intensity in MBV3 and ResNet50's latter layers.

ing latency improvement with 0.98% accuracy improvement. **SUSHI** can also save a significant amount of off-chip energy (78.7%) in simulation with realistic board configurations.

2 Background and Motivation

We start with a background on weight-shared neural networks in §2.1. Then we motivate and expose the opportunity for hardware support of weight-shared supernet inference (§2.2). The need for hardware-software co-design follows from challenges in §2.3. The hardware-software abstraction in §2.4 is introduced for generality.

2.1 Weight-Shared Deep Neural Networks (WS-DNNs)

Recent advances in deep learning propose weight-shared deep neural networks (Cai et al., 2019; Sahni et al., 2021; Yu et al., 2020) that propose *SuperNet* structures can be used to enable inference on Deep Neural Networks (DNNs) across a diverse set of deployment scenarios (both dynamic and static). Weight-shared DNNs (WS-DNN) induce a rich trade-off between accuracy and latency (Fig. 1b). The inference in WS-DNN fundamentally changes the traditional view of optimizing inference latency, which was focused on a single forward pass query. Instead, WS-DNN's inference makes it possible to satisfy the latency/accuracy requirements for a *stream of queries* with each query potentially requesting a different point in the trade-off space. This positions WS-DNNs as a salient candidate for a variety of applications (Halpern et al., 2019; Hsieh et al., 2018; Reddi et al., 2020) and inference-serving systems (Romero et al., 2021a) that benefit from navigating latency/accuracy trade-off. The key property of these networks is that different DNNs (*SubNet*), which may differ in several elastic dimensions, including depth and width, partially share their weights as part of a single large DNN (*SuperNet*). As a result, the *SuperNet* contains all other *SubNets* within it (Fig. 1a). These *SubNets* can be directly used to render predictions without any further re-training. To get predictions from a specific *SubNet*, elastic dimensions are specified in order to select appropriate weights from the *SuperNet* for the forward pass.

These elastic dimensions typically include specification of the depth, the number of filters/channels of each convolutional layer and kernels. The elastic dimensions of the neural net architecture of the *SuperNet* are exploited to attain elasticity. A typical *SuperNet* architecture such as OFAResNet, OFAMobileNet is organized as a collection of stages. Each stage consists of repeating blocks, such as a Bottleneck block in OFAResNets. Each block in turn contains multiple convolution layers. The depth elastic dimension selects top $k \in [2; 4]$ blocks per-stage of the *SuperNet*. The expand ratio (another elastic dimension) selects top k kernels of the convolution layer in each block. As a result, the smallest *SubNet*'s weights are shared by all other *SubNets* and the weights of the largest *SubNet* contain all other *SubNets* within it. Hence, there's always some amount of common weight sharing between *SubNets*, with cardinality of overlap ranging from the smallest to the largest *SubNet*.

2.2 Need for Hardware Support for WS-DNN Inference

The goal of hardware acceleration for ML inference is to serve a query with minimal latency and maximal accuracy. This goal becomes even more pronounced for WS-DNN inference, where each query may be served with different latency/accuracy requirements (Fig. 1b) (Cai et al., 2019; Sahni et al., 2021).

Achieving this goal is challenging due to memory-boundedness of some of the convolutional layers (Kao et al., 2022; Siu et al., 2018). This is especially true for the more recent smaller models that have lower arithmetic intensity (FLOPS/Byte) and when they are deployed on bandwidth-constrained embedded boards (Wang et al., 2019; Wei et al., 2019; Chen et al., 2016; Jokic et al., 2020; Siu et al., 2018; Chen et al., 2022).

We quantify this in Fig. 2, where we observe that a large fraction of convolution layers running on a canonical edge

accelerator are memory-bound [2].

This is problematic, since a significant portion of end-to-end inference latency and energy consumption comes from memory-bound layers, given the high latency and energy cost of data movement from memory to the on-chip storage (Chen et al., 2016; Yuan et al., 2021).

Hence, for the same amount of FLOPS it is very important to convert memory-bound layers to compute-bound in order to reduce end-to-end inference latency and energy consumption.

To do so, we leverage our key insight that WS-DNN inference on a stream of queries exhibits temporal locality. As different queries use different *SubNets*, many of them reuse the same weights shared among those *SubNets*, by design. We employ this insight to help convert memory-bound layers to be more compute-bound. Conceptually, this can be accomplished by reusing the shared weights used by previous queries for the next query in a stream, knowing that they all activate *SubNets* within the same shared *SuperNet* structure. This creates an opportunity for reuse *across queries*, in sharp contrast to techniques commonly explored and exploited in the computer architecture community for a *single* query for intra-model optimizations, such as weight-stationary, row-stationary, input-stationary, and output-stationary (Chen et al., 2017b; 2016; Fleischer et al., 2018; Venkatesan et al., 2019).

We call this novel reuse as *SubGraph* Reuse, as common shared weights form a *SubGraph* (e.g., created as the intersection of computational graphs of any two served *SubNets*). Note that in this paper we distinguish between *SubGraphs* and *SubNets*. *SubNet* is a subset of a *SuperNet* that can be used for forward-pass inference to serve a query, while a *SubGraph* is a subset of *SubNet*. Note that any *SubNet* is a *SubGraph*, but not vice versa.

A natural way to leverage *SubGraph* Reuse is to have a dedicated cache in the hardware accelerator. However, it comes with several challenges that we discuss in §2.3.

2.3 Design Challenges in WS-DNN Inference Specialized Hardware

The proposed specialized hardware for WS-DNN-inference exploits the temporal locality and enables *SubGraph* Reuse. However, assigning a dedicated on-chip buffer comes with both software and hardware challenges.

Hardware Challenges: Due to the resource-restricted nature of many deployment devices, the cache size may be too small to cache entire *SubNets*. Thus, the hardware must operate at a *finer caching granularity* of arbitrary *SubGraphs* instead. Deciding the size of the dedicated on-chip buffer is non-trivial.

Small buffer size leads to marginalizing the ability to exploit temporal locality. Larger dedicated on-chip buffer limits compute area as well as other on-chip buffer sizes that are leveraged for weight/row/input/output stationary optimizations.

Furthermore, the *SubGraph* Stationary depends on the compute/memory boundness of the convolution workload, which is further related to the off-chip bandwidth and throughput of the hardware. Therefore, the variation of the bandwidth and throughput will also affect the best cache size, which introduces more factors for consideration in the trade-off space.

Software Challenges: We argue that the latency of served *SubNets* depends on the *SubGraph* cached in the on-chip buffer. Fig. 3 provides a toy example to illustrate that: (a) a deep and thin *SubNet* gets a lower latency with a cached *SubGraph* containing more layers compared to other cached *SubGraphs* with fewer layers and wider bottleneck blocks, and (b) a wide and shallow *SubNet* achieves lower latency with a cached *SubGraph* with wider and fewer layers (matching its shape). This creates two challenges in software: (a) *SubNet* selection decision to serve the current query must be *aware* of the currently cached *SubGraph* (state), and (b) the cached *SubGraph* itself should be updated based on previously served *SubNets* for optimized latency. In other words, the software needs to make *cache-state aware* decisions to select the appropriate *SubNet* and update the cached state based on temporally local (e.g., recent) *SubNets* that were used to serve recent queries.

2.4 Hardware-Agnostic Software Scheduling

One final goal is to achieve generalizability for the software scheduler while retaining accelerator state awareness. The scheduler policy design could then generalize to any hardware that is able to support WS-DNN inference. Hence, there is a need to decouple the scheduler from the hardware, i.e., the change in the hardware should not require any changes in the scheduler policy code. We propose an abstraction between the software scheduler and the hardware accelerator that exposes latencies of serving a set of *SubNets* over a set of cached *SubGraphs*. We show that this gives the policy sufficient information about the hardware state in an accelerator-agnostic fashion. We discuss the mechanism of achieving this while managing the spatial complexity of such a lookup table in §3. We instantiate this mechanism in **`SushiSched`**, which we can now develop and improve upon independently on any hardware accelerator.

[2] In the same network, relatively lower arithmetic intensity corresponds to higher chances of becoming memory bound.

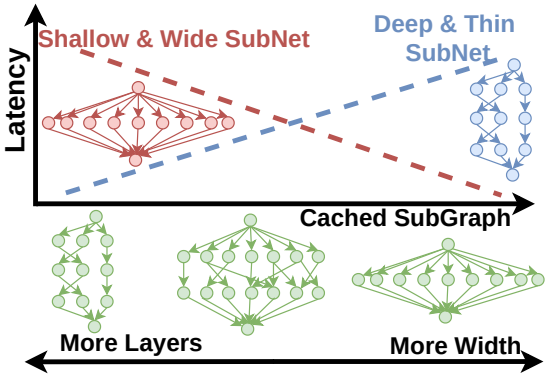


Figure 3. Latency of two different *SubNets* as a function of different cached *SubGraphs*. Different cached *SubGraphs* are optimal for different served *SubNets* with a non-trivial relationship based on the similarity of NN architecture parameters. **SushiSched** captures this similarity with a distance measure in §3.

3 System Design & Architecture

SUSHI serves a stream of queries with different latency/accuracy requirements. It consists of three major components — scheduler (**SushiSched**), abstraction (**SushiAbs**), and accelerator (**SushiAccel**) as shown in Fig. 4. **SUSHI** exploits novel *SubGraph* Reuse enabled via the interaction of its three components to serve queries with higher accuracy subjected to latency constraints or lower latency subjected to accuracy constraints. We describe our proposed **SushiAbs** and **SushiSched** below. **SushiAccel** is described in §4 in detail. The terminology used in this paper is captured in Fig. 5.

3.1 SUSHI's System Architecture

We describe the interaction between **SUSHI**'s components. Fig. 4 demonstrates a query path in **SUSHI**. The query enters the system with a certain latency and accuracy constraint. Then, the **SushiSched** makes a two-part control decision. First, it selects an appropriate *SubNet* (i.e., SN_t) that can serve the current query q_t. It makes this subnet selection with the help of **SushiAbs**. **SushiAbs** provides the scheduler with the ability to perform latency estimation when a specific *SubNet* is served with a given *SubGraph* cached. **SushiAbs** exposes this state in an accelerator-agnostic fashion. Second, **SushiSched** decides the next cached-*SubGraph*. The exact algorithm for this control decision is described in Alg. 1. **SushiSched** control decision is then enacted by **SushiAccel**. The selected *SubNet*, next cached-*SubGraph*, and query-data are sent to the **SushiAccel**. **SushiAccel** performs inference of the query using the selected *SubNet*. Model weights that are not already SGS-cached as part of the cached *SubGraph* are fetched from off-chip to on-chip buffer space. Finally, the accelerator returns the results of performing inference on *SubNet* to **SushiSched** and enacts the *SubGraph* caching control decision.

3.2 Abstraction

SushiAbs abstracts the ability to perform latency estimation for a given *SubNet* as a function of a cached *SubGraph* in an accelerator-agnostic fashion. It enables **SushiSched** to make cached*SubGraph* aware control decisions. As these control decisions are performed on the critical path of the query, this enabling abstraction must be efficient both w.r.t. space (R1) and time (R2).

Indeed, the set of all possible cached-*SubGraphs* is exponentially large for WS-DNNs ($>> 10^{19}$) (Cai et al., 2019). Thus, to achieve (R1), the abstraction limits the set of all possible cached *SubGraphs* to a significantly smaller set $\mathcal{S}$, such that $|\mathcal{S}| << 10^{19}$. The size of *SubGraphs* in $\mathcal{S}$ are selected to be close to the cache size. Hence, at any point in time, **SushiAccel** always caches *SubGraphs* from $\mathcal{S}$ and **SushiSched** also selects a *SubGraph* to cache from $\mathcal{S}$ as well. The abstraction achieves (R2) by using a lookup table data structure with *SubNets* as rows and *SubGraphs* as columns. Hence, it takes the least amount of time to get latency-estimate of *SubNet* i for a given *SubGraph* j. The size of the lookup table is given by $O(|\mathcal{S}|.|\mathcal{X}|) \approx O(|\mathcal{S}|)$ where $\mathcal{X}$ denotes the set of serving *SubNets*, since we expect $O(|\mathcal{X}|) \approx O(1)$.

3.3 SushiSched Design

Algorithm 1: Scheduling Algorithm

Input: *SubNet* to be served SN_i, $i \in [1...N]$, *SubGraph* to be cached G_j, $j \; \varepsilon \; [1...M]$, Latency table $L[i][j]$.
Result: *SubNet* to be served and *SubGraph* to be cached.
Calculate *SubNet* to be served for every query $q_t = (A_t, L_t)$, $t \in [0...Q]$ and *SubGraph* to be cached every Q iterations;
$AvgNet$ = [0,0,0...0]; $CacheState = \varnothing$;
while q_t **do**
  if *policy* == *STRICT_ACCURACY* **then**
    id_x = argmin$_{latency}(L[i][CacheState]$ $\forall i \in [0...N]$ s.t. SN_i.accuracy $>= A_t$);
  else
    id_x = argmax$_{accuracy}(L[i][CacheState]$ $\forall i \in [0...N]$ s.t. SN_i.latency $<= L_t$);
  end
  for *every* Q *queries* **do**
    $AvgNet$.update(SN_{id_x}, Q)
    CacheState = argmin$_{Dist}$(Dist(G$_j$,AvgNet)) $\forall j \in [0...M]$);
  end
end

On the software side, the scheduler receives a stream of

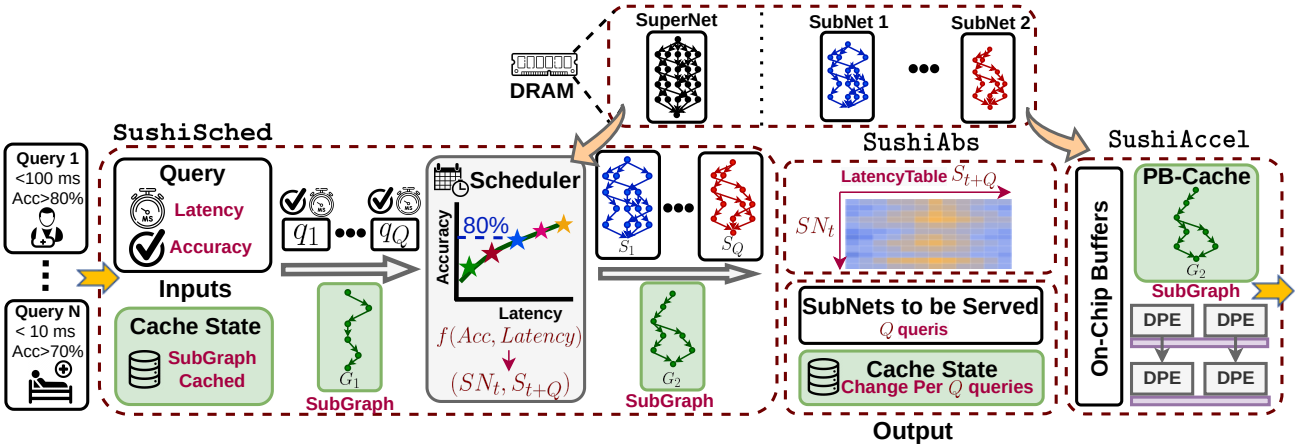
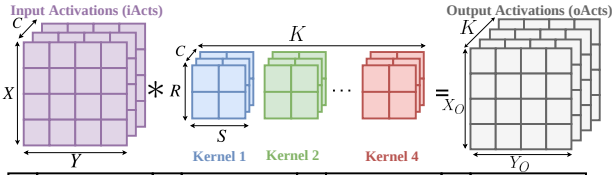


Figure 4. System architecture overview. Given a stream of queries annotated with (Accuracy, Latency) pairs $q_1, .., q_Q$ and the current cache state C_1, the scheduler chooses the *SubNet* to be served SN_i for each i'th query and next cache state G_2 after every Q queries.



C	Input channel	R	Height of kernel	X	Height of iActs	X_O	Height of oActs
K	Number of kernel	S	Width of kernel	Y	Width of iActs	Y_O	Width of oActs
C_P	Parallelism in the input channel			K_P	Parallelism in the kernel		

Figure 5. **SUSHI** terminology and variable definitions.

queries, where each query is annotated with an (Accuracy, Latency) pair, denoted (A_t, L_t). In this section we will describe exactly how the scheduler makes its *SubNet* selection and *SubGraph* caching control decisions.

Per-query *SubNet* (SN_t) Selection. As shown in Fig.4, the scheduler decision is guided by two primary considerations: (i) serve strictly higher accuracy and (ii) serve strictly smaller latency, which can be specified by the user. In case of strictly higher accuracy, the scheduler can choose from the feasibility set of all *SubNets* with accuracy $\geq A_t$. **SUSHI** serves a *SubNet* that has minimum latency among all the *SubNets* that have accuracy $\geq A_t$. Note that, it may be possible that the served latency might not satisfy the latency constraint of $\leq L_t$. In case of strictly lesser latency, the scheduler serves a *SubNet* that has maximum accuracy among all the *SubNets* that have latency $\leq L_t$. Similarly, it is possible that the served accuracy might not satisfy the accuracy constraint of $\geq A_t$. Notice that the accuracy for a given *SubNet* is fixed, whereas the latency depends on the *SubGraph* cached into the PB. The scheduler employs a $Latency-Table$ to get the latency values for *SubNet* given a cache state.

Across-query *SubGraph* Caching (S_{t+Q}). The scheduler needs to decide what *SubGraph* to cache after every Q queries (S_{t+Q}). To make this decision, the scheduler needs to represent the *SubGraphs* and *SubNets*, use the information from the past Q queries, and predicts the next *SubGraph* that should be cached into the PB.

Encoding *SubGraph* NN Architecture. The scheduler represents both the *SubNets* and the *SubGraphs* as a vector as shown in Fig. 6. The scheduler uses the number of kernels K_i and the number of channels C_i of every layer i to create a vector of size $2N$ for N layered neural network. For instance, the vectorized representation for a 3-layered neural network would be $[K_1, C_1, K_2, C_2, K_3, C_3]$.

Amortizing Caching Choices. The scheduler keeps a running average of the past Q *SubNets* that were served by the scheduler as shown in Fig.6 (middle). The running average serves as a good indicator of the kernels and the channels that were frequently used in the *SubNets* that were served for the past Q queries. If some kernels or channels were frequently used in the past Q *SubNets*, the values corresponding to these kernels or channels will be high in the vectorized representation. Notice that, the running average can be considered as an approximation of the intersection operation, but with more information. Doing intersection purely loses the information for the kernels and the channels that were frequent but not present in all the *SubNets*; however, averaging helps us to preserve this information.

Predicting the Next *SubGraph* (S_{t+Q}). The scheduler employs the distance from the running average of the past Q queries to predict the next *SubGraph* to be cached as shown in Fig.6. The scheduler caches the *SubGraph* that has the minimum distance from the average *SubNet*. Minimum distance ensures that the most frequent kernels and channels will be cached into the PB. In case fitting all of them is not possible, minimum distance from average *SubNet* ensures that we are picking the best fit *SubGraph* in terms of frequently occurring channels and kernels in the *SubNets* served by the scheduler. The algorithm for performing both the scheduler decisions is described briefly in Algorithm 1. **SushiSched** receives input from the user including *SubGraphs*, *SubNets*, $LatencyTable$. AvgNet is

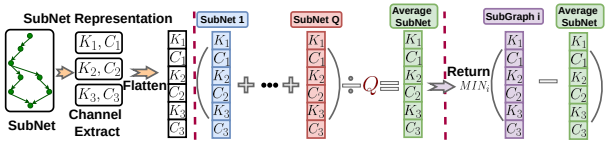


Figure 6. The scheduler represents each neural network as a vector using the number of kernels and channels for each layer. The scheduler maintains a running average of the *SubNets* that were served for the past Q queries. For every Q queries, the scheduler caches the *SubGraph* that is the closest to the average *SubNet*.

the running average of the served *SubNets*. The cache state is set to a random *SubGraph* initially. The **SushiSched** decides the *SubNet* to be served for a given query when the accuracy is a hard constraint i.e. serving strictly better accuracy. The **SushiSched** can also decide the *SubNet* to be served if the latency is a hard constraint i.e. serving strictly lesser latency. It updates the running average of the *SubNets*. Finally, the **SushiSched** determines the *SubGraph* that is closest to the AvgNet and caches it into the PB.

4 SushiAccel Implementation

4.1 Hardware Design Challenges

As discussed earlier in §2 and §3, to support *SubGraph* Stationary, we propose to augment DNN accelerators with a custom cache called Persistent Buffer (PB). The introduction of PB leads to a new design space because it competes for a finite on-chip buffer capacity (that needs to be partitioned across input activation, weight, and output activation tiles, and also shared weights).

To guarantee the best performance of hardware design on such a design space, we have to develop the parameterizable hardware template with the support of different hardware configurations.

4.2 Architectural Components

In this part, we introduce components of **SushiAccel** (Fig. 7) and how it supports all proposed data reuse in Fig. 8.

4.2.1 Compute Array

Dot Product Engine (DPE). The key building block of DNN accelerators is the ability to compute *dot-products*. For example, the Google TPU systolic array (Jouppi et al., 2017) computes fixed-size dot products in each column by keeping weights stationary and forwarding (streaming) inputs from one column to the other, NVDLA (NVIDIA, 2016) employs dedicated dot product engines (DPEs) of size 64, while flexible accelerators (Kwon et al., 2018; Qin et al., 2020) have DPEs of configurable sizes (enabled via all-to-all connectivity between the buffers and PEs). In this work, we picked fixed-size DPEs of size 9. Larger kernels will be breakdown into a serial of 3×3 kernels and get flattened across the multipliers for reduction using the adder tree. As for small kernels (1×1), C dimension will be flattened across multipliers to leverage input channel parallelism.

Parallelism. To further increase the throughput, we instantiate a 2D array of DPEs to boost the throughput by leveraging parallelism and reuse as shown in the Fig. 8. As for the parallelism, the number of row indicates the total number of kernels being processed in parallel in DPE Array, i.e. kernel-level parallelism (K_P). While the number of column stands for total number of input activation (iAct) channels being processed in parallel, i.e. channel-level parallelism (C_P).

Both iActs and weights take the same interface to save the wire cost and improve scalability. In the vertical axis, both weights and iActs pass through DPEs of different rows in the store-and-forward fashion. During the weights forwarding, DPE will keep targeted weights stationary. Then, iActs will be streamed and get processed. In the horizontal axis, we replicate the same DPE independently to process different iActs channels and add an extra adder tree to reduce results from DPEs in the same row.

4.2.2 On-chip Buffers and Supported Data Reuse

We designed a custom on-chip buffer hierarchy to both store data in the layout preferred by the DPE array and support reuse opportunities not leveraged by the DPE array. The entire on-chip storage is divided into multiple separate buffers for different types of data as illustrated by different colors in Fig. 7.

Persistent Buffer (PB). The PB is designed to enable *SubGraph* Reuse. For example, **SushiAccel** loads the *SubGraph* (kernel 1) in Fig. 8d from off-chip memory only once and stores it inside PB, such that it could be reused when switching between *SubNet* 1 and *SubNet* 2.

Dynamic Buffer (DB). The DB is a typical on-chip storage to store the distinct weights of the requested *SubNet*. By adopting a PB, only non-common weights need to be fetched from the off-chip to the on-chip storage. For example, in Fig. 8d, all kernels except the common part (kernel 2 to kernel N) will be loaded into DB when targeting at *SubNet* 1, and will be replaced by kernel M to kernel $M + N$ when switching into *SubNet* 2. The DB is implemented as a ping-pong buffer, as indicated by DB1 and DB2 in Fig. 7, to hide the latency of fetching distinct weights from the off-chip DRAM.

Streaming Buffer (SB). SB is designed to store entire iActs and support *iAct Reuse - Multiple kernels.* (Fig. 8b).

Line Buffer (LB). LB works as a serial to parallel conversion (Wang et al., 2021) because the line buffer takes a single pixel from SB and moves it internally. Therefore, iActs data among different sliding windows will be reused

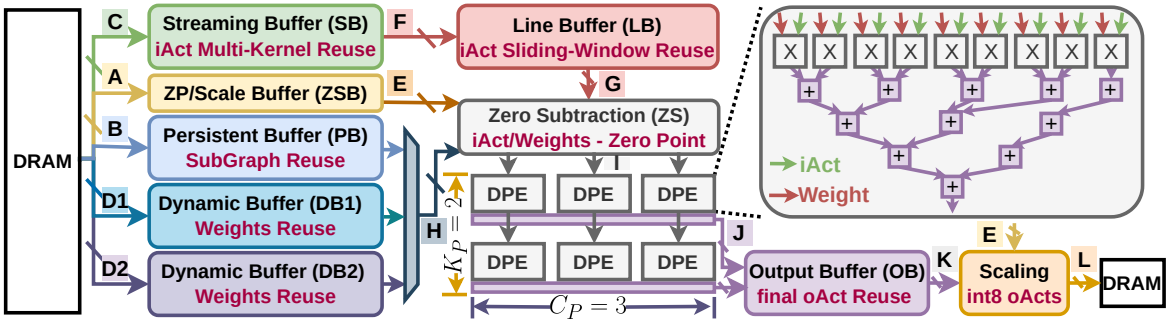
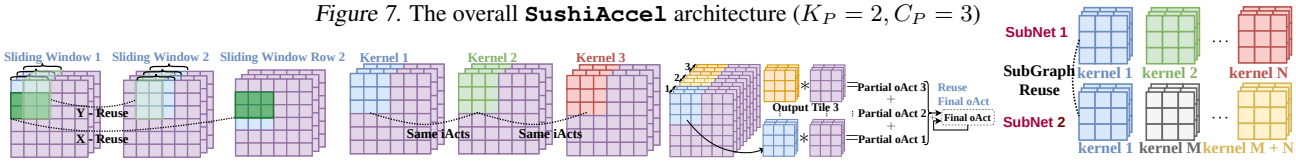
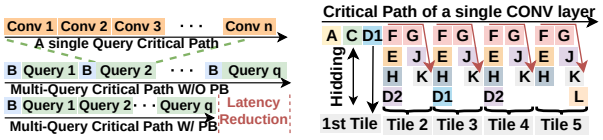


Figure 7. The overall **SushiAccel** architecture ($K_P = 2, C_P = 3$)



(a) iAct Sliding Window Reuse. (b) iAct Multi-Filter Reuse. (c) oAct Reuse. (d) *SubGraph* Reuse

Figure 8. Data reuse opportunities in serving different *SubGraphs* leveraged within **SushiAccel**.



(a) Multi-query timeline. (b) Single Conv timeline.

Figure 9. **SushiAccel** dataflow overview.

inside the LB, i.e. LB supports *iAct Reuse - Sliding Window Overlap* (Fig. 8a). We augment the naive line buffer to support stride by enabling sliding windows skipping.

Output Buffer (OB). OB provides in-place accumulation for oActs of different channels such that only the final oActs will be sent off-chip to save data movement of partial sums.

ZP/Scale Buffer (ZSB). ZSB serves as the on-chip storage for zero point and scale for quantized inference.

4.3 SushiAccel Dataflow

4.3.1 *Latency Reduction from Inter-Query Dataflow*

The inter-query processing timeline of **SushiAccel** is shown in Fig. 9a where stage B indicates the movement of the common *SubGraph* from off-chip to on-chip PB. The latency saving of **SushiAccel** comes from eliminating the redundant off-chip *SubGraph* access, as illustrated in Fig. 9a where **SushiAccel** reduces common *SubGraph* off-chip access (stage B) to only once in the critical path instead of multiple times in design w/o PB.

4.3.2 *Hiding Latency from Intra-layer Dataflow*

Within each convolution layer, **SushiAccel** processes a convolution layer in the granularity of weight tiles shown in Fig. 9b. Different stages (i.e., A-L) are defined in Fig. 7 that represent the movement of specific data. To further hide off-chip data access latency from critical path, we implement a double distinct weights buffer (ping-pong dynamic buffers DB1 and DB2 shown in Fig. 7) to hide the off-chip latency of fetching distinct weights behind the computation latency. This is indicated by stages D1 and D2 that are hidden from stages F-G-J-K shown with arrows in Fig. 9b.

5 Experimental Results

5.1 System Setup

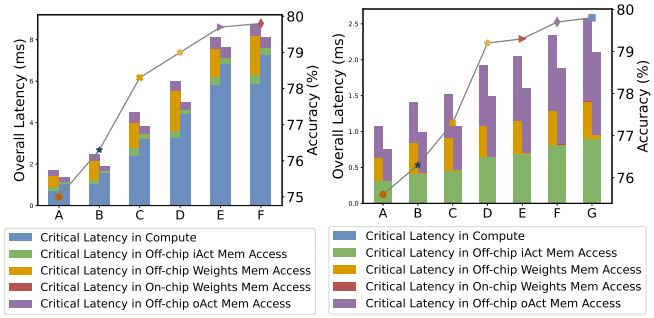
Workload: We choose weight shared version of ResNet50 and MobV3 as two *SuperNets* (Cai et al., 2019). To evaluate **SUSHI** with full range on the pareto-frontier, we pick a sequence of 6 and 7 *SubNets* from ResNet50 and MobV3, respectively.

The sizes of ResNet50 *SubNets* range from the [7.58 MB, 27.47 MB] while the sizes of MobV3 *SubNets* range from [2.97 MB, 4.74 MB]. Shared weights take up 7.55 MB and 2.90 MB for ResNet50 and MobV3, separately [3]. *SubNets* are obtained using the procedure mentioned in OFA (Cai et al., 2019).

Metrics: Latency in this section refers to the end-to-end serving latency of a given model, while accuracy refers to the top-1 accuracy. Both accuracy and latency are defined for *SubNets* only. *SubGraphs* are only used for the caching purpose as a subset of *SubNets*.

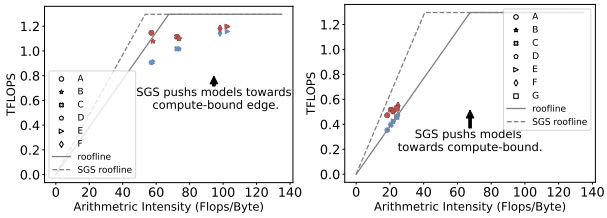
Architecture Analytic Model: We have developed an analytic model which estimates the behavior of **SushiAccel** to explore design space by configuring the architecture with parameters.

[3]Weights, input activations, and zero points are quantized to int8, and the quantization scale is quantized into int32.



(a) ResNet50. (b) MobV3.

Figure 10. Potential latency reduction with SGS (two bar per *SubGraph*, left: w/o PB; Right: w PB)



(a) ResNet50. (b) MobV3.

Figure 11. SGS pushes memory-bound to compute-bound layers.

Our model accurately predicts the latency trend of **SushiAccel** using profiled latency of **SushiAccel** on both workloads, enabling us to perform an exhaustive search of all parameter combinations within specified constraints. This approach allows for the identification of optimal configurations for improved performance in both simulation and real-world deployment.

Roofline Analysis We also extended a roofline analysis tool to study the effect of PB on the boundness of **SushiAccel** under different workloads.

Deployment Platforms: We implemented the proposed **SushiAccel** on two FPGA including ZCU104 (5 W) and Alveo U50 (75 W). We compare our **SushiAccel** w/ PB and w/o PB with Xilinx DPU and CPU (Intel i7 10750H, 45 W).

Scheduler Simulator: We have developed **SushiSched**, which runs on the CPU and guides the **SushiAccel** on how to serve the current query and (a) what *SubGraph* to serve and (b) *SubNet* to be placed in PB.

5.2 SUSHI Impact on Arithmetic Intensity

To understand the benefits of SGS, we perform roofline analysis as shown in Fig. 10 and Fig. 11, where roofline represents the normal roofline curve. And SGS-roofline virtually improves the overall off-chip bandwidth by saving off-chip data access, leading to an improved roofline curve shown by SGS roofline. The experiments are performed with a system with 19.2 GB/s off-chip memory bandwidth and 1.296 Tflops throughput running at 100 MHz (Reuther et al., 2022).

The latency breakdown results in Fig. 10 shows that SGS can potentially remove the off-weights access latency from the critical path, such that the individual latency of serving a stream of queries from pareto-frontiers could be reduced by [6%, 23.6%] for MobV3 and [5.7%, 7.92%] for ResNet50.

Such latency reduction essentially comes from the model boundedness shifting. The SGS pushes models towards compute-bound, which increases the utilization of the available compute resources for higher throughput and reduces latency and energy consumption. The shifting is illustrated by blue dots being pushed toward the red dots in Fig. 11.

Table 1. Bandwidth requirement of on-chip buffers

Buffer	Minimal Bandwidth Requirement
DB	$LCM($ max off-chip BW, DPE Array demanded on-chip $BW)$
SB	$LCM($ max off-chip $BW, C_P \times R \times S \times$ iActs DataWidth)
LB	DPE Array demanded on-chip BW
OB	$K_P \times$ oAct DataWidth
PB	$LCM($ max off-chip BW, DPE Array demanded on-chip $BW)$

Note: BW = bandwidth, $LCM(x_1, x_2)$: Least Common Multiple of x_1 and x_2.

5.3 SushiAccel Configuration Impact

In this subsection, we explore the impact of three main factors (i.e., bandwidth, throughput, and PB size) of **SushiAccel** on the overall end-to-end serving latency.

5.3.1 *Bandwidth - Buffers Arrangement*

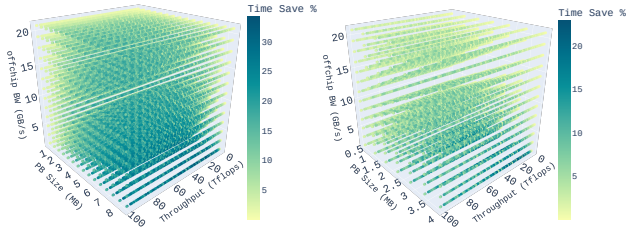
Different types of data require different bandwidths. A unified buffer for all different data types demands the controller to handle potentially all-to-all connections between buffers and all compute units. While the design of the splitting buffer only needs a direct connection between a buffer and compute units, which saves the complexity of both the datapath and the controller. The buffer is a 2D array and its size equals $width \times height$. The width refers to the bandwidth a buffer could supply every cycle. The bandwidth demand of different buffers is shown in Tab. 1, which is determined by both workloads and hardware specifications.

5.3.2 *PB Size - Sizes of Buffers*

All buffers compete on the same total storage budget so that a balance of them is preferred to achieve good performance. The addition of a persistent buffer also introduces a new factor of common weights reuse, leading to a trade-off between inter-layer data reuse and intra-layer data reuse.

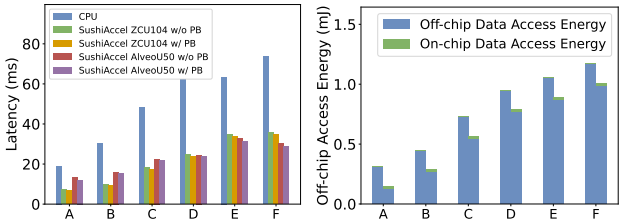
5.3.3 *Throughput - Parallelism of the Compute Array*

The parallelism of the 2D DPE Array is also a controllable knob. Within the same computation engine budget, a change in parallelism indicates a change in throughput, yielding different performances on different workloads. For example,



(a) ResNet50. (b) MobV3.

Figure 12. Latency reduction (Time Save in legend) improvement exploration on **SushiAccel** using Analytic Model.



(a) Latency comparison (b) Energy comparison

Figure 13. Real board latency and energy reduction for ResNet50. (left and right bars in (b) are **SushiAccel** w/o PB and w/ PB)

the parallelism of 16 and 32 in K and C dimensions deliver a peak throughput of 512 data per clock cycle. Therefore, we use this throughput as the factor to abstract parallelism.

5.3.4 *Design Space Exploration*

As Fig. 12 shows with larger PB sizes, more on-chip computation, and less off-chip bandwidth, the latency is improved. However, for MobV3, due to the smaller size, having depthwise conv layers, and less reuse, the amount of improvement is lesser for MobV3 compared with the ResNet50.

5.4 SushiAccel Evaluation

In this subsection, we evaluate how **SushiAccel** will impact the latency and energy reduction. We evaluate different scales of **SushiAccel** on two real FPGAs with different budgets running the 3x3 convolution layers of ResNet50. The **SushiAccel** on Alveo U50 has off-chip bandwidth of 14.4 GB/s, PB size of 1.69 MB, and throughput of 0.9216 TFlops running at 100 MHz.

5.4.1 *Resources Allocation among Buffers*

The resource utilization of **SushiAccel** w/ PB and w/o PB under optimal configurations on both Xilinx ZCU104 and Alveo U50 are shown in Tab. 2 with a breakdown on-chip storage allocation shown in Tab. 3. Both **SushiAccel** w/ PB and **SushiAccel** w/o PB use the same amount of overall on-chip storage for a fair comparison.

5.4.2 *Latency Evaluation*

The real-board latency and energy consumption results are shown in Fig. 13a with resources shown in Tab. 2. On ZCU104, compared with CPU, **SushiAccel** w/o PB achieves $1.81X \sim 3.04X$ speedup and **SushiAccel** w/ PB achieves $1.87X \sim 3.17X$ for different *SubNets*. While on Alveo U50, compared with CPU, **SushiAccel** w/o PB achieves $1.43X \sim 2.54X$ speedup and **SushiAccel** w/ PB achieves $1.57X \sim 2.61X$ for different *SubNets*. Fig. 13a also shows that the scale-up design on Alveo U50 performs worse than the small-scale design on ZCU104 under small *SubNets* because of higher off-chip DRAM competition in data center cluster hosting Alveo U50 than simple embedded ZCU104. Thus, off-chip data access dominates latency in Alveo U50, resulting in the slow down for small *SubNets*.

5.4.3 *Energy Evaluation*

Energy in data movement has been proved to dominate the entire power consumption of neural network accelerator (Dally et al., 2020) and thus we estimate the overall energy through profiling the off-chip DRAM data access for all different platforms shown in Figure 13b.

We estimate the off-chip energy by profiling the DRAM data access and compute it as $NumberAccess \times EnergyPerAccess$. With the proposed *SubGraph* Reuse, we could save $[14\%, 52.6\%]$ off-chip data access energy saving for ResNet50 and $[43.6\%, 78.7\%]$ for MobV3 compared to **SushiAccel** w/o PB.

5.5 Comparing with DPU

We compared **SushiAccel** against Xilinx DPU using real layer-wise end-to-end inference latency of min-*SubNet* on ZCU104 as shown in Fig. 14. We consider convolution layers with 3×3 kernel sizes. **SushiAccel** w/o PB achieved $0.5 \sim 1.95\times$ faster execution time than Xilinx DPU (25.1% GeoMean speedup). This quantitative comparison lends credence to the proposal of adding a Persistent Buffer (PB) to a state-of-the-art ML accelerator design. There are also seldom cases when **SushiAccel** performs worse than Xilinx DPU, because **SushiAccel** takes less parallelism in height (X) and width (Y) dimensions (Fig. 5), leading to higher latency under workload with higher X and Y values.

5.6 SushiSched Functional Evaluation

In this section, we evaluate the performance of **SushiSched** for both ResNet50 and MobV3.

Fig. 15 shows that the **SushiSched** is able to serve queries with strictly lesser latency and/or better accuracy where blue dots represent served queries by employing **SushiSched**. In Fig. 15a and Fig. 15c, blue dots are almost always below the line $y = x$ manifesting that the **SushiSched** can

Table 2. Resources comparison of **SushiAccel** with DPU

	SushiAccel w/o PB	**SushiAccel** w/ PB	Xilinx DPU DPUCZDX8G	**SushiAccel** w/o PB	**SushiAccel** w/ PB
Device	ZCU104	ZCU104	ZCU104	Alveo U50	Alveo U50
LUT	61180 (26.6%)	64307 (27.9%)	41640 (18.1%)	231668 (26.63%)	244969 (28.16%)
Register	107216 (23.3%)	117724 (25.5%)	69180 (15%)	435071 (24.96%)	445602 (25.56%)
BRAM	192.5 (61.7%)	198.5 (63.6%)	0	452.5 (33.67%)	452.5 (33.67%)
URAM	48 (50%)	96 (100%)	60 (62.5%)	48 (7.5%)	96 (15%)
DSP	1507 (87.2%)	1459 (87.2%)	438 (25.35%)	4739 (79.78%)	4740 (79.79%)
PeakOps/cycle	2592	2592	2304	9216	9216
GFlops (100MHz)	259.2	259.2	230.4	921.6	921.6

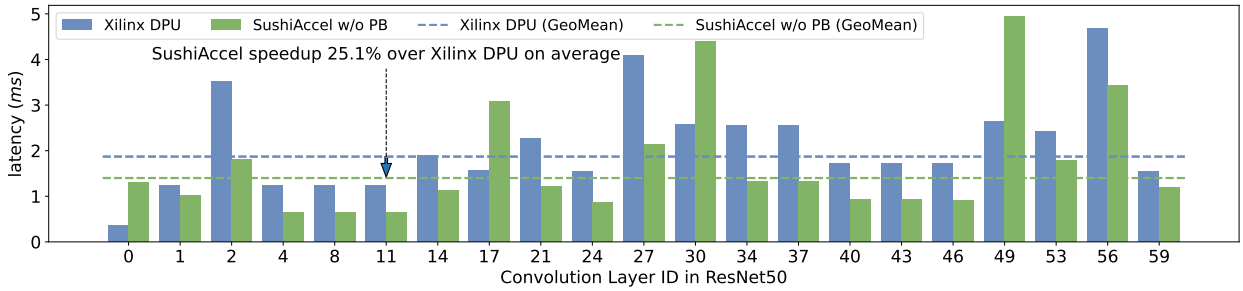


Figure 14. The latency comparison between **SushiAccel** w/o PB and Xilinx DPU for ResNet50.

Table 3. Buffer configurations of **SushiAccel** (ZCU104 board)

	SushiAccel w/o PB		**SushiAccel** w/ PB	
	BRAM (KB)	URAM (KB)	BRAM (KB)	URAM (KB)
DB-Ping	0	1152	0	576
DB-Pong	0	1152	0	576
SB	8	1152	8	576
LB	54	0	54	0
OB	327	0	327	0
ZSB	8	0	8	0
PB	0	0	0	1728
Overall	397	3456	397	3456

Table 4. Reuse comparison (prior works v.s. **SUSHI**).

Work	iActs Reuse Fig. 8a & 8b	oAct Reuse Partial Sum	Weights Reuse iAct Tiling	*SubGraph* Reuse
MAERI (Kwon et al., 2018)	✓	✗	✓	temporal ✗
NVDLA (NVIDIA, 2016)	✗	✓	✓	temporal ✗
Eyeriss (Chen et al., 2016)	✓	✗	✓	temporal ✗
Xilinx DPU (Xilinx, 2022)	✓	✓	✓	temporal ✗
SUSHI	✓	✓	✓	spatial ✓ temporal ✓

serve strictly lesser latency if the latency is a hard constraint that needs to be satisfied. Similarly, all blue dots above the line $y = x$ in Fig. 15b and Fig. 15d show that the **SushiSched** can serve strictly better accuracy if accuracy is a hard constraint that needs to be met.

5.7 End-to-End SUSHI Evaluation

In this section, we compare the latency-accuracy tradeoff results among **SUSHI** w/o PB, **SUSHI** w/ PB (state-unaware caching), and **SUSHI**. The blue dots in Fig. 16 illustrate how **SUSHI** serves random queries[4].

For ResNet50 in all cases, **SUSHI** w/o scheduler consistently outperforms No-**SUSHI**. For random queries, **SUSHI** is also able to decrease the latency by 21% on average given the same accuracy compared to not having **SUSHI**.

In the case of MobV3, due to its small size, a relatively larger fraction of a *SubNet* fits in PB, resulting in a higher cache-hit ratio (Appendix A.4). **SUSHI** offers better accuracy-latency tradeoff than **SUSHI** w/o scheduler, with the exception of only a few points. In the case of MobV3, **SUSHI** is also able to decrease the latency by 25% on average given the same accuracy compared to not having **SUSHI**.

Finally, **SUSHI** increases the serving accuracy by up to 0.98% for the same latency, which is significant for ML serving applications.

[4]Due to the overlap, only limited points in the figures are visible

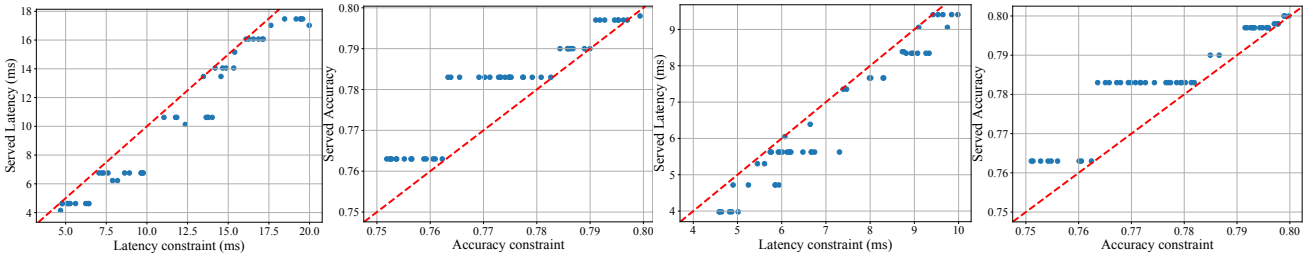
6 Related Work

Various accelerator designs such as Maeri (Kwon et al., 2018), Eyeriss (Chen et al., 2018), NVDLA (NVIDIA, 2016), and DPU (Xilinx, 2022) support different types of reuse Fig. 8. A comparison of them is shown in Tab. 4. However, all of these works achieve intra-model cross-layer reuse in contrast to the cross-query reuse we propose with **SushiAccel**.

Clipper (Crankshaw et al., 2017) serves single model queries without exposing a latency/accuracy tradeoff. Inferline (Crankshaw et al., 2018) serves multiple models but in a pipeline, there's no latency/accuracy tradeoff per model. INFaaS (Romero et al., 2021b) provides a query-time latency/accuracy tradeoff mechanism and policy but suffers from expensive model switching mechanisms. This also translates into a policy that minimizes model switching as a result. The vertically integrated inference serving stack provided by **SUSHI** naturally plugs into existing inference serving frameworks, enabling agile navigation of the latency/accuracy tradeoff at query time.

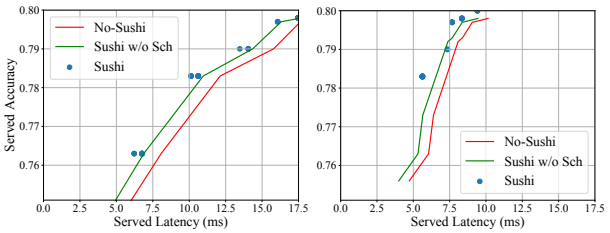
7 Conclusion

SUSHI is a vertically integrated hardware-software inference serving stack that takes advantage of the temporal locality induced by serving inference queries on the same



(a) Latency of ResNet50. (b) Accuracy of ResNet50. (c) Latency of MobV3. (d) Accuracy of MobV3.

Figure 15. Serve strictly better accuracy and lesser latency for ResNet50 and MobV3 using **SUSHI**.



(a) ResNet50. (b) MobV3.

Figure 16. Comparing delivering latency-vs-accuracy of No-**SUSHI** and **SUSHI** w/o scheduler and baselines for ResNet50 and MobV3.

weight-shared supernetwork structure. To the best of our knowledge, the concept of SubGraph Stationary (SGS) optimization across queries is novel. We demonstrate that, to achieve the best temporal locality benefit, the proposed hardware implementation **SushiAccel** must work in tandem with the software scheduler **SushiSched** to control what SubNets to serve for each query and how to update the accelerator state. We further ensure generalizability of **SushiSched** by abstracting the effect of hardware state on the latency (and energy) of served SubNets with a black box SubGraph latency table. This decouples **SushiSched** from any accelerator implementation, while maintaining its state-awareness implicitly. **SUSHI** can be naturally integrated in state-of-the-art ML inference serving frameworks and enables better latency/accuracy tradeoffs for a stream of queries with latency/accuracy constraints. For a stream of queries, our results show 0.98% improvement in the served accuracy, and up to 25% latency reduction.

8 Acknowledgment

This material is based upon work partially supported by the National Science Foundation under Grant Number CCF-2029004. Additional support was provided by a sponsored research award by Cisco Research. We would like to further acknowledge the insightful comments of the review panel as well as the skillful guidance of our shepherd, Dr. Qijing Jenny Huang, which greatly contributed to the quality of this paper. We thank the anonymous reviewers of MLSys, and the SAIL Research Group members for valuable feedback and the stimulating intellectual environment they provide. We also thank Taekyung Heo from Synergy lab for his feedback on the initial version of the paper. **Disclaimer:** Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Abdelaziz, H., shafiee, a., Shin, J. H., Pedram, A., and Hassoun, J. Rethinking floating point overheads for mixed precision dnn accelerators. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 223–239, 2021. URL https://proceedings.mlsys.org/paper/2021/file/5f93f983524def3dca464469d2cf9f3e-Paper.pdf.

Ali, W., Abdelkarim, S., Zidan, M., Zahran, M., and El Sallab, A. Yolo3d: End-to-end real-time 3d oriented object bounding box detection from lidar point cloud. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pp. 0–0, 2018.

Bai, Y., Wang, Y. X., and Liberty, E. Proxquant: Quantized neural networks via proximal operators. In *ICLR'19*, arXiv (2018), 2018.

Cai, H., Zhu, L., and Han, S. Proxylessnas: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332, 2018. URL http://arxiv.org/abs/1812.00332.

Cai, H., Gan, C., and Han, S. Once for all: Train one network and specialize it for efficient deployment. *CoRR*, abs/1908.09791, 2019. URL http://arxiv.org/abs/1908.09791.

Chen, A., Demmel, J., Dinh, G., Haberle, M., and Holtz, O. Communication bounds for convolutional neural networks. *arXiv preprint arXiv:2204.08279*, 2022.

Chen, L.-C., Papandreou, G., Schroff, F., and Adam, H. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017a.

Chen, Y.-H., Krishna, T., Emer, J. S., and Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep

Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2016.

Chen, Y.-H., Emer, J., and Sze, V. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro*, 37(3):12–21, 2017b.

Chen, Y.-H., Yang, T.-J., Emer, J., and Sze, V. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *arXiv preprint arXiv:1807.07928*, 2018.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 613–627, 2017.

Crankshaw, D., Sela, G.-E., Zumar, C., Mo, X., Gonzalez, J. E., Stoica, I., and Tumanov, A. Inferline: Ml prediction pipeline provisioning and management for tight latency objectives. *arXiv preprint arXiv:1812.01776*, 2018.

Dally, W. J., Turakhia, Y., and Han, S. Domain-specific hardware accelerators. *Communications of the ACM*, 63: 48 – 57, 2020.

Datta, T., Mishra, S., and Swain, S. Real-time tracking and lane line detection technique for an autonomous ground vehicle system. In *International Conference on Intelligent Computing and Smart Communication 2019*, pp. 1609–1625. Springer, 2020.

Deo, N. and Trivedi, M. M. Convolutional social pooling for vehicle trajectory prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1468–1476, 2018.

Eriksson, D., Chuang, P. I., Daulton, S., Xia, P., Shrivastava, A., Babu, A., Zhao, S., Aly, A., Venkatesh, G., and Balandat, M. Latency-aware neural architecture search with multi-objective bayesian optimization. *CoRR*, abs/2106.11890, 2021. URL https://arxiv.org/abs/2106.11890.

Fang, J., Shafiee, A., Abdel-Aziz, H., Thorsley, D., Georgiadis, G., and Hassoun, J. Near-lossless post-training quantization of deep neural networks via a piecewise linear approximation. *CoRR*, abs/2002.00104, 2020. URL https://arxiv.org/abs/2002.00104.

Fleischer, B., Shukla, S., Ziegler, M., Silberman, J., Oh, J., Srinivasan, V., Choi, J., Mueller, S., Agrawal, A., Babinsky, T., et al. A scalable multi-teraops deep learning processor core for ai trainina and inference. In *2018 IEEE Symposium on VLSI Circuits*, pp. 35–36. IEEE, 2018.

Gog, I., Kalra, S., Schafhalter, P., Gonzalez, J. E., and Stoica, I. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 453–471, 2022.

Halpern, M., Boroujerdian, B., Mummert, T., Duesterwald, E., and Reddi, V. J. One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 34–47, Los Alamitos, CA, USA, mar 2019. IEEE Computer Society. doi: 10.1109/ISPASS.2019.00012. URL https://doi.ieeecomputersociety.org/10.1109/ISPASS.2019.00012.

Hong, S., Xu, Y., Khare, A., Priambada, S., Maher, K., Aljiffry, A., Sun, J., and Tumanov, A. HOLMES: Health OnLine Model Ensemble Serving for Deep Learning Models in Intensive Care Units. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1614–1624, 2020.

Hsieh, K., Ananthanarayanan, G., Bodik, P., Venkataraman, S., Bahl, P., Philipose, M., Gibbons, P. B., and Mutlu, O. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 269–286, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/hsieh.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. URL http://arxiv.org/abs/1602.07360.

Jiang, H., He, P., Chen, W., Liu, X., Gao, J., and Zhao, T. SMART: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2177–2190, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.197. URL https://aclanthology.org/2020.acl-main.197.

Jokic, P., Emery, S., and Benini, L. Improving memory utilization in convolutional neural network accelerators. *IEEE Embedded Systems Letters*, 13(3):77–80, 2020.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor

processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Kao, S.-C., Subramanian, S., Agrawal, G., Yazdanbakhsh, A., and Krishna, T. Flat: An optimized dataflow for mitigating attention bottlenecks, 2022.

Kwon, H., Samajdar, A., and Krishna, T. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

Liu, W., Liao, S., Ren, W., Hu, W., and Yu, Y. High-level semantic feature detection: A new perspective for pedestrian detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5187–5196, 2019.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the value of network pruning. *CoRR*, abs/1810.05270, 2018. URL http://arxiv.org/abs/1810.05270.

NVIDIA. NVIDIA Deep Learning Accelerator (NVDLA), 2016. URL http://nvdla.org/primer.html.

Ovtcharov, K., Ruwase, O., Kim, J.-Y., Fowers, J., Strauss, K., and Chung, E. S. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

Pouransari, H., Tu, Z., and Tuzel, O. Least squares binary quantization of neural networks. In *CVPRW'20*, 2020.

Qin, E., Samajdar, A., Kwon, H., Nadella, V., Srinivasan, S., Das, D., Kaul, B., and Krishna, T. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, 2020. doi: 10.1109/HPCA47549.2020.00015.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459. IEEE, 2020.

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., and Kepner, J. Ai and ml accelerator survey and trends, 2022. URL https://arxiv.org/abs/2210.04055.

Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411. USENIX Association, July 2021a. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/romero.

Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411, 2021b.

Sahni, M., Varshini, S., Khare, A., and Tumanov, A. CompOFA – compound once-for-all networks for faster multi-platform deployment. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=IgIk8RRT-Z.

Siu, K., Stuart, D. M., Mahmoud, M., and Moshovos, A. Memory requirements for convolutional neural network hardware accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 111–121. IEEE, 2018.

Sundermeyer, M., Schlüter, R., and Ney, H. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

Tabernik, D. and Skočaj, D. Deep learning for large-scale traffic-sign detection and recognition. *IEEE transactions on intelligent transportation systems*, 21(4):1427–1440, 2019.

Venkatesan, R., Shao, Y. S., Wang, M., Clemons, J., Dai, S., Fojtik, M., Keller, B., Klinefelter, A., Pinckney, N., Raina, P., et al. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2019.

Wang, C., Liu, Y., Zuo, K., Tong, J., Ding, Y., and Ren, P. ac 2 slam: Fpga accelerated high-accuracy slam with heapsort and parallel keypoint extractor. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–9. IEEE, 2021.

Wang, Y. E., Wei, G.-Y., and Brooks, D. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019.

Wei, X., Liang, Y., and Cong, J. Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2019.

Xilinx. Xilinx Deep Learning Unit (DPU), 2022. URL https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Deep-Learning-Processor-Unit.

Yu, J., Jin, P., Liu, H., Bender, G., Kindermans, P.-J., Tan, M., Huang, T., Song, X., Pang, R., and Le, Q. Bignas: Scaling up neural architecture search with big single-stage models. In *European Conference on Computer Vision*, pp. 702–717. Springer, 2020.

Yuan, G., Behnam, P., Li, Z., Shafiee, A., Lin, S., Ma, X., Liu, H., Qian, X., Bojnordi, M. N., Wang, Y., et al. Forms: Fine-grained polarized reram-based in-situ computation for mixed-signal dnn accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 265–278. IEEE, 2021.

Zhang, D., Yang, J., Ye, D., and Hua, G. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *European Conference on Computer Vision (ECCV)*, 2018.

A APPENDIX - ABLATION STUDIES

A.1 Temporal Analysis of Subgraph Caching

In this section, we explore the impact of a number of vectorized *SubGraphs* employed in the running average results as well as the size of $Latency - Table$ on the accuracy-latency results. Making cache update decisions after each query improves both latency and accuracy results (Fig. 17), but is prohibitively expensive as the new *SubGraph* must be fetched from off-chip memory.

For ResNet50, increasing the number of queries to two, the results worsen. Increasing the number of queries to 4 and 8 yields better results. Eventually, there's a point when the performance starts to get worse (e.g., at 10+ queries) as the benefit of temporal locality will be reduced. So there's a tradeoff between the staleness of query history over which the cached *SubGraph* is computed and the cost of updating cache frequently.

Following the same methodology for MobV3 (18), we observe that averaging over 10 queries gives us the best trade-off, leading to better accuracy-latency results.

A.2 Impact of $Latency - Table$ size

The results in Tab. 5 show the average latency improvement by increasing the size of $Latency - Table$ compared with **SUSHI** w/o scheduler. As the results for ResNet50 show, increasing the size of the table improves performance, but is quickly saturated. This is consistent with the important property of **SushiSched** table (rapid lookups on the critical query path).

For MobV3, we see almost no improvement in latency with increased table size, which shows that if the PB is large enough to hold a large portion of the *SubNet* (and, with other on-chip buffers—the whole *SubNet*), the small table size can capture most of the required information by the scheduler. Thus, for smaller models, we keep the horizontal size of $Latency - Table$ minimal.

A.3 Lookup Latency

We used the lookup table as a fast-search data structure. For the largest model, (ResNet-50) the latency in microseconds is shown in table Tab. 6. These results show that the lookup table time is less than $\frac{1}{1000}$ of the inference time and, thus, doesn't significantly interfere with the query's critical path.

A.4 Cache Hit Ratio

SUSHI leverages temporal locality across queries as the SubNets they induce share some weights that are common to these SubNets. The benefit of *SubGraph* Reuse thus fundamentally is a function of the workload. For instance, if all queries used the exact same SubNet and we cache the largest SubGraph of that SubNet, then the probability of its reuse is 1. To generalize this intuition, **SUSHI** makes

Table 5. Average latency improvement with respect to the size of $Latency - Table$ normalized to **SUSHI** w/o scheduler

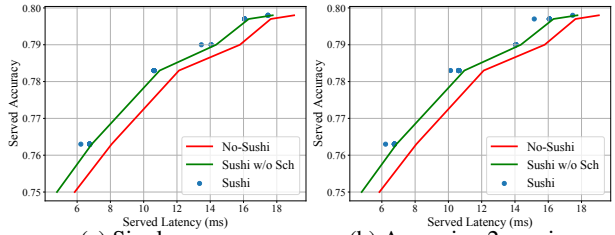
	10-cols	40-cols	80-cols	100-cols	500-cols
ResNet50	4%	7%	8%	9%	9%
MobV3	1%	1%	1%	1%	1%

Table 6. Look up time (us)

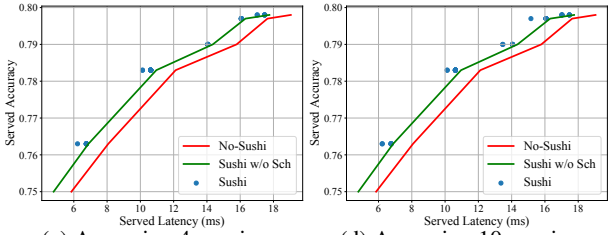
	100-cols	200-cols	500-cols	1000-cols	2000-cols
ResNet50	2	4	6	10	17

caching decisions based on the intersection of SubNets used by the last Q queries. Thus, we define the cache hit ratio as the fraction of the cached SubGraph that was "hit" or present in the SubNet served, because those weights don't need to be fetched.

For a given query trace, we log (SN_t, G_t) series of tuples where SN_t is the *SubNet* that the scheduler decided to serve at time t, and G_t is a *SubGraph* cached in PB at time t. We find the overlap between SN_t and G_t using $\frac{\|(SN_t \cap G_t)\|_2}{\|(SN_t)\|_2}$ where SN_t is already vectorized using C and K. We average this over t to get the average cache hit ratio. $||\cdot||_2$ is used as a proxy to calculate vector overlap. Thus defined, **`SUSHI`** reaches a hit ratio of 66% (78%) for ResNet50 (MobV3). It is instructive that the cache hit ratio is higher for smaller models, as the intersection of common weights used by *SubNets* over a past window of Q queries is a larger fraction of the served *SubNet*.

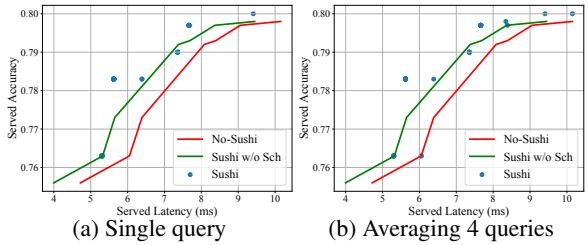


(a) Single query (b) Averaging 2 queries

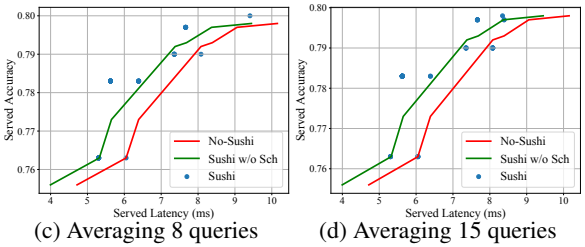


(c) Averaging 4 queries (d) Averaging 10 queries

Figure 17. Temporal analysis of subgraph caching for ResNet50



(a) Single query (b) Averaging 4 queries



(c) Averaging 8 queries (d) Averaging 15 queries

Figure 18. Temporal analysis of subgraph caching for MobV3