

Demand Layering for Real-Time DNN Inference with Minimized Memory Usage

Mingoo Ji^{1,2}, Saehanseul Yi³, Changjin Koo¹, Sol Ahn¹, Dongjoo Seo³, Nikil Dutt³, and Jong-Chan Kim^{1,4}

¹Graduate School of Automotive Engineering, Kookmin University, Korea

²Automotive Electronics Advanced Development TFT, Hyundai Mobis, Korea

³Department of Computer Science, University of California, Irvine, USA

⁴Department of Automobile and IT Convergence, Kookmin University, Korea

Correspondence: jongchank@kookmin.ac.kr

Abstract—When executing a deep neural network (DNN), its model parameters are loaded into GPU memory before execution, incurring a significant GPU memory burden. There are studies that reduce GPU memory usage by exploiting CPU memory as a swap device. However, this approach is not applicable in most embedded systems with integrated GPUs where CPU and GPU share a common memory. In this regard, we present Demand Layering, which employs a fast solid-state drive (SSD) as a co-running partner of a GPU and exploits the layer-by-layer execution of DNNs. In our approach, a DNN is loaded and executed in a layer-by-layer manner, minimizing the memory usage to the order of a single layer. Also, we developed a pipeline architecture that hides most additional delays caused by the interleaved parameter loadings alongside layer executions. Our implementation shows a 96.5% memory reduction with just 14.8% delay overhead on average for representative DNNs. Furthermore, by exploiting the memory-delay tradeoff, near-zero delay overhead (under 1 ms) can be achieved with a slightly increased memory usage (still an 88.4% reduction), showing the great potential of Demand Layering.

I. INTRODUCTION

To enable efficient deep neural network (DNN) inference with low-cost embedded hardware, its memory requirement should be minimized. For that, a typical approach is to apply pruning and quantization [1]–[3] that reduce the number of model parameters, however, at the cost of unavoidable accuracy loss. Once a model is fixed, all the parameters are loaded into system memory before execution. To the best of our knowledge, most state-of-the-art DNN frameworks employ this method despite its excessive memory usage. However, in the era of large-scale models [4]–[8] and concurrent DNNs [9]–[11], we argue that this naïve approach is no longer viable, and thus a new system approach is needed that can alleviate this excessive memory requirement.

Recent studies try to reduce the memory usage of DNN inference by efficiently managing activation buffers between DNN layers [12]–[14]. However, they are not applicable for storing model parameters. Besides, SwapAdvisor [15] provides a general method by utilizing inexpensive CPU memory as a swap device of scarce GPU memory. This method is promising in discrete GPU (dGPU) systems with separate CPU and GPU memory. However, most embedded systems use integrated GPUs (iGPUs), where CPU and GPU share a common memory system [16]. In such systems, reducing

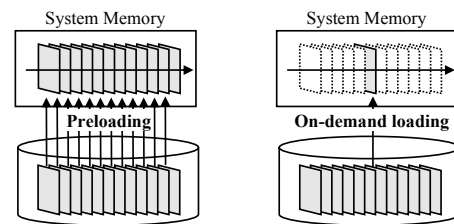


Fig. 1: Preloading vs. Demand Layering.

GPU memory at the cost of increased CPU memory does not provide any benefit.

With this motivation, this study aims to reduce the memory usage of iGPU-based DNN inference systems, explicitly targeting the memory for model parameters. Our idea is to borrow the concept of *demand paging* in conventional operating systems, where program instructions are loaded to CPU memory on demand in the granularity of *pages* (typically sized 4 KB - 16 KB). Similarly, exploiting the layer-by-layer execution of DNNs, we propose *Demand Layering* that loads model parameters on demand in the granularity of *layers* while dropping previous layers of no use. In this manner, the memory requirement is significantly reduced to the order of a single layer from the order of the entire model. Fig. 1 highlights the difference between the preloading architecture and our Demand Layering.

However, the memory reduction is not free. It comes at the cost of increased delays. Thus, we conducted a thorough delay analysis, which found out that the inference delay can be analyzed in terms of the following three operations in it:

- **Read:** Model parameters are read into CPU memory.
- **Copy:** Model parameters are copied to GPU memory.
- **Kernel:** DNN layers are executed by GPU kernels.

In the preloading architecture, all the read and copy operations are only in the *initialization phase*; thus, its inference delay is just the sum of GPU kernel executions. In contrast, Demand Layering repeatedly conducts read and copy operations during the *inference phase*, which potentially causes extra delays. For that, our baseline approach is to employ a high-performance solid-state drive (SSD). Compared with eMMC storages typically with 300 MB/s sequential read

performance, M.2 NVMe SSDs provide up to 7000 MB/s of sequential read performance [17]. Although random reads are somewhat slower, most DNN model files exhibit sequential access patterns due to the inherent sequential nature of DNN executions [10], [11].

Even with the fastest SSD, extra delays are still significant. Thus, our next approach is to hide away the delays as much as possible by pipelined execution of read, copy, and kernel operations. Fortunately, even in iGPU systems, these three operations can run in parallel, because read operations can be carried out by CPU while copy and kernel operations are being processed by GPU. Even better, Nvidia GPUs have two separate processing units: a copy engine (CE) and an execution engine (EE). The CE can process copy operations while the EE is executing GPU kernels [18], [19]. As a result, read, copy, and kernel operations can run fully in parallel. Based on this parallel hardware architecture, we developed and evaluated a number of software pipeline architectures on an Nvidia Jetson AGX Xavier platform with various DNNs. The remainder of this section introduces the case with YOLOv4 [20] in particular, whose model size is 245.8 MB and its average inference delay in the preloading architecture is 160.8 ms. Besides, the largest layer size is 18.0 MB.

Synchronous pipeline. In the 3-stage synchronous pipeline architecture, its read, copy, and kernel stages advance while synchronized with a common pipeline cycle. Since kernel operations are usually the longest among the three stages, most read and copy operations are hidden behind kernel operations. This pipeline architecture needs two inter-stage buffers: (i) a CPU memory buffer between read and copy stages and (ii) a GPU memory buffer between copy and kernel stages. Since each buffer needs to hold just the layer being processed, the required buffer size is the size of the largest layer. In addition, the buffers should be *double-buffered* because, for example, a read to the CPU buffer can happen simultaneously with a copy from the same buffer. The same applies to the GPU buffer. Our implementation provides an 85.4% memory reduction (to 72.0 MB) with 23.7% delay overhead (to 198.9 ms).

Asynchronous pipeline. If a read operation happens to be the longest in a synchronous pipeline cycle, it causes a GPU idling interval, negatively impacting the delay. To minimize such unwanted delays, our architecture is modified to an asynchronous pipeline, where pipeline stages advance at their own paces [21]. Between the pipeline stages, we introduce two *circular buffers* that can barely hold the largest layer each, instead of the two pairs of double buffers used in the synchronous architecture, cutting the memory requirement in half. Our implementation provides a 92.7% memory reduction (to 36.0 MB) with 12.7% delay overhead (to 181.2 ms).

Two-stage pipeline. Recent iGPU-based system on chips (SoCs) (e.g., Nvidia Xavier) provide a special memory management scheme so that a memory buffer can be accessed both from CPU and GPU [16]. This zero-copy memory eliminates the need of copy operations, enabling a 2-stage pipeline. By this architecture, the memory requirement is further reduced to just the order of a single layer. Our implementation provides

a 96.3% memory reduction (to 18.0 MB) with 21.5% delay overhead (to 195.3 ms).

Memory-delay tradeoff. In the asynchronous pipeline architectures, we can intentionally increase the circular buffer size to exploit the tradeoff relation between memory and delay. Thus, we can devise an iterative optimization process by gradually increasing the buffer size until there is no further delay reduction. By this optimization method, we can find the *minimal delay* configuration. As a result, near-zero (< 1.0 ms) delay overhead is achieved by a slight increase in memory usage (from 18.0 MB to 52.8 MB).

The contributions of this study can be summarized as:

- We propose Demand Layering for minimized memory usage in DNN inference systems by loading and executing layers in a layer-by-layer manner.
- Three pipeline architectures are presented that minimize the extra delay overhead of Demand Layering.
- The pipeline architectures are implemented and evaluated on Nvidia Jetson AGX Xavier, showing significant memory reductions with near-zero delay overhead.

II. PRELIMINARIES

A. Deep Neural Networks (DNNs)

In contrast to conventional programs, which are sequences of *instructions*, DNNs are sequences of *parameters*, organized by layers such as convolutional and fully connected layers. The parameters are produced in a training phase and stored in a DNN *model file*, whose file format depends on the DNN framework of your choice. For example, Darknet [22] uses .weights binary files. PyTorch [23] uses .pt or .pth files, which are serialized binary files by the Python pickle module. TensorFlow [24] uses .pb files, which are binary files by the ProtoBuf format.

Regardless of the file format, the model files must be loaded to GPU memory in the initialization phase. Then, in the inference phase, the preloaded parameters are interpreted and executed by a DNN inference framework in a *layer-by-layer* manner [9], [10]. This preloading architecture inherently imposes a significant GPU memory burden for storing the entire model parameters, especially serious in multi-DNN systems.

B. Integrated CPU-GPU Systems

When designing embedded systems for DNN applications, iGPUs are highly preferred to dGPUs due to the advantage in its size, weight, and power (SWaP) properties [16]. In contrast to dGPUs, iGPUs share the same physical memory space with CPU. In such systems, GPU memory optimization at the expense of CPU memory cannot make a beneficial deal. Instead, a holistic CPU-GPU memory optimization method is required.

A typical example of integrated CPU-GPU systems is Nvidia Jetson AGX Xavier, which is our experimental platform. Fig. 2 shows its internal architecture with 16 GB shared DRAM, an 8-core 64-bit ARM CPU, and a 512-core integrated Volta GPU connected through a system bus. Additionally, it is

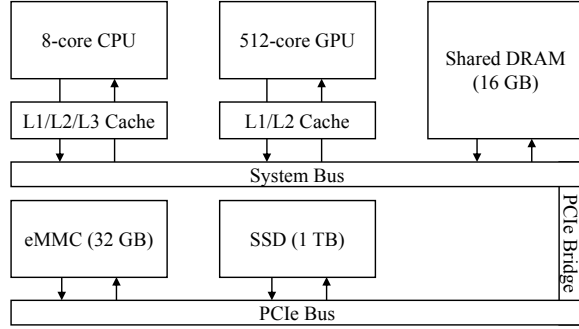


Fig. 2: Nvidia Jetson AGX Xavier hardware architecture.

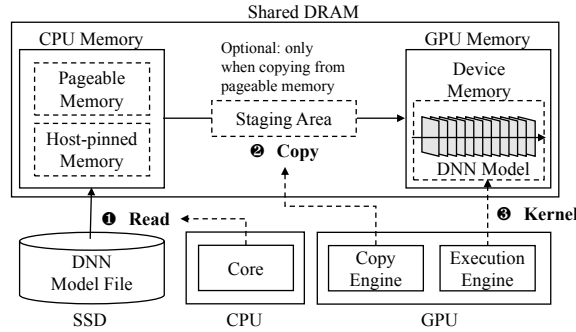


Fig. 3: Data flow of model parameters.

equipped with an M.2 NVMe interface through a PCI express (PCIe) bus that can host an optional SSD besides its built-in 32 GB eMMC storage.

C. Solid-State Drives (SSDs)

For many years, eMMC storages have dominated most embedded systems since conventional embedded applications did not require either TB-scale capacity or GB/s-level bandwidth. However, in recent data-intensive applications like autonomous driving, a vast amount of data should be stored and retrieved in real time, requiring huge storage capacities and high bandwidth. Since neither of them can be achieved by eMMC devices, a viable alternative is to employ SSDs in such data-centric embedded systems. Recent commercial off-the-shelf (COTS) SSDs can satisfy such excessive requirements with their ever-growing capacity and bandwidth.

Our experimental platform is also equipped with a Samsung 980 PRO NVMe M.2 SSD with 1 TB capacity and its officially announced 7000 MB/s sequential read performance. The SSD is connected to both CPU and GPU via a PCIe Gen4 interface. In our target application (i.e., DNN inference), the SSD is used to store DNN model files, which are usually above hundreds of megabytes. Furthermore, in multi-DNN systems, the storage requirement is far more significant, making SSDs an ideal choice for storing DNN model files.

D. Data Flow of DNN Model Parameters

To begin an inference (i.e., forward propagation) on a DNN model, the entire parameters should be in GPU memory such that GPU kernels can directly access them. For that, a three-step approach is usually used, which is depicted in Fig. 3. The model file is first read from disk to a CPU memory buffer (❶). When allocating CPU buffers, there are several choices provided by the Nvidia CUDA runtime, which will be detailed in Section IV-B. Then the parameters are copied to a GPU memory buffer (❷). When the source CPU buffer happens to be a pageable memory by the usual `malloc()` function that is not under the control of the CUDA runtime, the copy is done via a hidden staging area, incurring possible blockings and delays in case of a staging area shortage. After the copy operation, GPU kernels can access and execute the DNN layers in the GPU memory buffer (❸). As explained in Section II-B, CPU and GPU memory buffers are from the same shared DRAM space. Thus, both buffers should be accounted for when estimating the memory usage of a DNN inference system. The read operation is processed by CPU, while the copy and kernel operations are executed by GPU. Since GPUs have two separate processing units for them (i.e., *copy engine* and *execution engine*), read and copy operations can run simultaneously [25]. As a result, read, copy, and kernel operations can run fully in parallel, providing a great chance for optimizing the DNN execution architecture.

E. Observations and Our Motivation

Meanwhile, we have the following observations during the investigation on various DNN inference frameworks in GPU-based embedded systems:

(i) **Memory burden in DNN inference.** To the best of our knowledge, most DNN inference frameworks preload the whole model parameters from disk to memory in the initialization phase to avoid disk operations in the inference phase. However, this *preloading* architecture permanently occupies a significant amount of system memory for storing model parameters, which is not acceptable in resource-constrained embedded systems.

(ii) **Layer-by-layer DNN execution:** DNN models have a layered structure, where there are strict data dependencies between layers. Model files are also organized following the layered architecture. Most notably, when a certain layer is executing, only that layer's parameters are accessed, and the rest of the parameters are irrelevant to the current layer execution.

(iii) **High-performance SSDs.** Most recent SSDs are fast enough, reaching the speed of 7000 MB/s for sequential reads. Certainly, the speed of random reads is far slower than sequential reads. Fortunately, however, what we need for the model file is only sequential reads that can best extract the peak performance of SSDs.

Motivation. With the above observations, our intuition is that the memory usage can be drastically reduced by loading and unloading model parameters by a layer's granularity in the inference phase without preloading them in the initialization

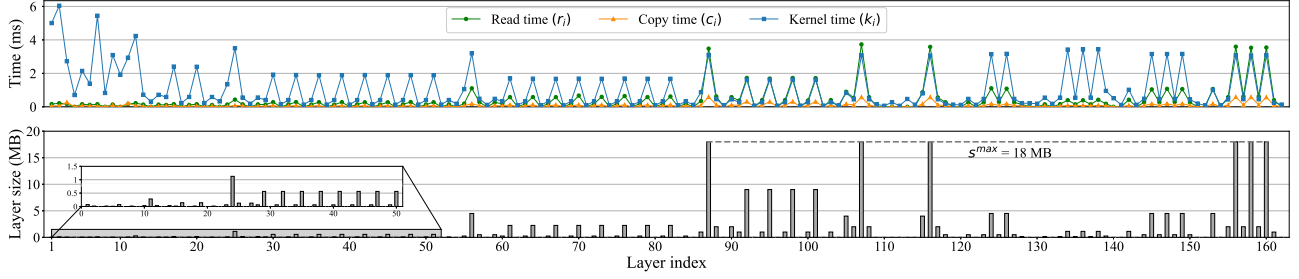


Fig. 4: Layer-wise measurements of the YOLOv4 object detection network (input resolution = 608×608).

phase. By that, the maximum memory usage will be reduced from the order of the entire model to the order of a single layer. However, read and copy operations are additionally performed in the inference phase, potentially adding extra delays if they are not adequately overlapped with kernel executions using pipeline architectures.

Initial Profiling. To pre-inspect the optimization opportunity, we measured the layer execution time of the popular YOLOv4 object detection model using the Darknet framework on an Nvidia Jetson AGX Xavier platform. The upper graph in Fig. 4 shows each layer’s average read, copy, and kernel execution times for 100 inference iterations, while the lower graph shows the size of each corresponding layer. Comparing the two graphs gives a hint of the strong correlation between timings and layer sizes. Fig. 5 shows the measured distributions, indicating that the read and copy times are significantly shorter than the kernel times in most layers, showing the potential for overlapping read and copy operations behind kernel executions.

III. SYSTEM MODEL AND PROBLEM DESCRIPTION

A. System Model

We assume an integrated CPU-GPU system equipped with an SSD, where a DNN inference engine runs a given DNN with N layers, denoted by $\{l_1, l_2, \dots, l_N\}$. The whole model parameters are stored in a model file in the SSD, which is sequentially organized by layers. Each i -th layer l_i ’s size is denoted by s_i , while the largest layer size is denoted by s^{max} . In the timing perspective, each layer is characterized by a tuple $l_i = (r_i, c_i, k_i)$, where r_i is the time for reading, c_i is the time for copying, and k_i is the time for kernel execution. Note that r_i , c_i , k_i , and s_i are random variables, not fixed values like worst-case or average execution times. For ease of presentation, we rather use *read time*, *copy time*, and *kernel time* when referring to the above layer-wise timing properties. Also, when specifically referring to the read, copy, and kernel operations themselves, we use capital letters (R_i , C_i , K_i) instead.

In YOLOv4 in Fig. 4, we have $N = 162$ layers with various layer types (i.e., 110 convolutional, 21 route, 23 shortcut, 3 maxpool, 3 yolo, and 2 upsample layers). The total size of the layers is 245.8 MB. As shown in the figure, Most r_i , c_i , and k_i have strong correlations to s_i , because large layers naturally

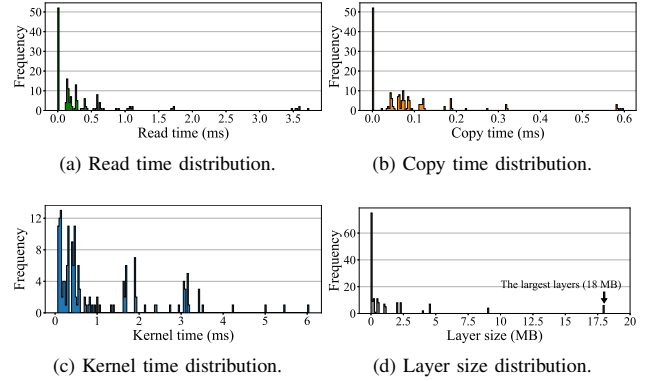


Fig. 5: Measured distributions.

involve more time to read, copy, and execute. One interesting observation is that the foremost layers exhibit relatively large kernel times, which do not properly reflect their small layer sizes. The reason is that the foremost layers extract features from the image, so their kernel times tend to depend strongly on the input image size rather than the layer size.

By looking at Fig. 5, all the four distributions are long-tailed. In particular, regarding the layer size, 75 layers (46%) are under 0.1 MB, while we have six layers with the same largest layer size $s^{max} = 18.0$ MB. Another interesting question is how the input image resolution affects the timings. Fig. 6 compares the total read, copy, and kernel times with varying input resolutions. For example, 608 in the x-axis represents a 608×608 input resolution to the DNN. The figure indicates that the input resolution has no noticeable impact on the read and copy times because they only depend on the layer size. However, the kernel time is heavily dependent on the input resolution. Thus, if the input resolution gets too small (e.g., the 224×224 case), the total kernel time becomes even smaller

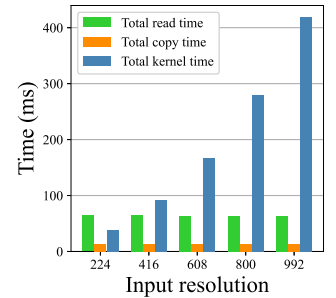


Fig. 6: How the input resolution affects the timings (YOLOv4).

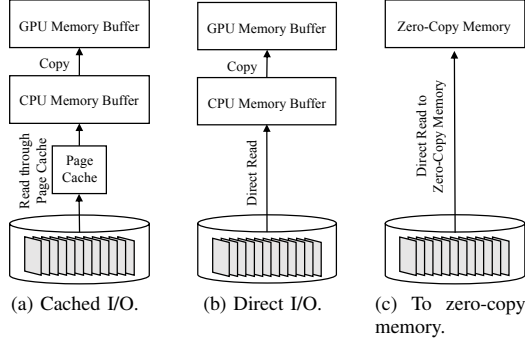


Fig. 7: Comparison of I/O methods for read operations.

than the total read time, possibly losing the optimization opportunity for overlapping read and copy operations behind kernel executions. However, such low input resolutions are practically not used due to their inferior detection accuracy. We use 608×608 as the default input resolution, which is from the YOLO DNN's default configuration.

B. Problem Description

In the preloading architecture, all the read and copy operations are finished in the initialization phase. Thus, only the kernel operations are executed in the inference phase. Thus, its delay is given by

$$\sum_{i=1}^N k_i, \quad (1)$$

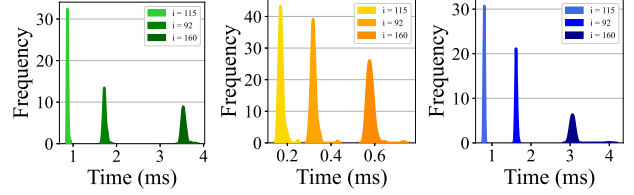
which is the *optimal delay* that cannot be reduced any further. In the memory perspective, we need a CPU buffer and a GPU buffer, both of which should be able to accommodate the entire model parameters. Thus, the memory consumption is two times the total layer size, given by

$$2 \times \sum_{i=1}^N s_i. \quad (2)$$

Beginning from the above preloading architecture, this study has the following objectives:

- To design and implement a layer-by-layer loading and execution (Demand Layering) framework (Section IV).
- To minimize the delay overhead caused by the additional read and copy operations (Section V and Section VI).
- To evaluate our implementation in terms of memory and delay with state-of-the-art DNN workloads (Section VII).

This study limits our problem to systems running a single DNN in isolation. However, once solved, the solution can be easily applied to systems concurrently running multiple DNNs. In fact, our solution is more valuable in such multi-DNN systems that require significantly larger memory to encompass multiple DNNs [9].



(a) Read distributions. (b) Copy distributions. (c) Kernel distributions.

Fig. 8: Profiling results of selected convolutional layers ($s_{115}=4$ MB, $s_{92}=9$ MB, and $s_{160}=18$ MB).

IV. DEMAND LAYERING

A. Read Operations

A read operation R_i is to read the l_i 's portion (layer parameters) of the model file to CPU memory. We assume that the model file is sequential such that there need no random offset changes between reads to fully exploit the SSD's sequential read performance. For the sake of real-time performance, the variance of r_i must be kept within a predictable range. However, assuming the stock Linux scheduler, it is not possible to provide hard guarantees.

Nonetheless, to minimize the variance as much as possible, our choice is to use *direct I/O* (Fig. 7b) instead of *cached I/O* (Fig. 7a), expecting to suppress unpredictable cache behaviors. Fig. 8a shows the profiling results of read times by direct I/O with selected YOLOv4 layers, showing that the variances are predictable within ranges. Also, by using direct I/O, we have another significant benefit of saving CPU memory for the page cache. Due to the aggressive page cache policy of Linux, the page cache will eventually duplicate most contents of the model file in CPU memory. Because we intend to minimize the memory usage by model parameters, it is desirable not to have a duplicate in the page cache.

In addition, our read operations are designed as asynchronous, using the POSIX asynchronous I/O method (AIO). With AIO, the calling thread does not block while a background read thread is processing pending read requests in the AIO queue. Since AIO has a limitation of supporting only `O_DIRECT` files, there is no choice but to use direct I/O.

B. Copy Operations

A copy operation C_i is to transfer a layer's parameters in CPU memory to GPU memory, making them accessible by GPU kernels. The CUDA runtime provides synchronous (`cudaMemcpy()`) and asynchronous (`cudaMemcpyAsync()`) copy functions. Here, our choice is, again, to make our copy operations fully asynchronous such that the completion of an asynchronous copy request can be later checked by CUDA synchronization primitives like the CUDA event API.

However, the asynchronous copy functions may sometimes block depending on the CPU memory buffer types. The CUDA runtime provides two basic memory types for CPU memory

buffers: (i) *pageable* and (ii) *host-pinned*. A pageable buffer is allocated by `malloc()`, while a host-pinned buffer is allocated by `cudaHostAlloc()`. Invoking an asynchronous copy with its source location at a pageable memory buffer is unsafe. The copy will fail if the source buffer is paged out during the copy operation. In this regard, the CUDA runtime provides a hidden staging area, as shortly explained with Fig. 3, which is in host-pinned memory that is never paged out. Although the internal organization of the staging area is not officially released, we found out that, by reverse engineering, it is made of a limited number of fixed-length buffers. If the buffers are all in use, the copy request will block. Also, if a copy size is larger than the largest individual staging buffer, it will block. With experiments, we observed many such unexpected blocking scenarios when copying large layers with asynchronous copy functions. With the above observation, our choice is to use host-pinned memory for CPU buffer allocations. Fig. 8b shows the profiling results of such copy times of selected YOLOv4 layers, which are much smaller than read and kernel times.

C. Kernel Operations

A kernel operation K_i is to execute CUDA kernel functions that implement the corresponding layer, accessing only GPU memory buffers allocated by `cudaMalloc()`. For example, if the layer is a convolutional layer, it may call CUDA-based GeMM (General Matrix Multiplication) functions with input feature maps and layer parameters (i.e., convolution filters) to produce output feature maps. If a higher-level DNN library (e.g., cuDNN) is used, it may call the library's built-in implementation of a convolutional layer. Fig. 8c shows the profiling results of kernel times of selected YOLOv4 layers, which are significantly larger than the read and copy times. Kernel requests are non-blocking in its nature. A kernel request just places the request in a CUDA stream (i.e., a queue for pending kernel calls) and returns. Then the kernel request eventually goes through the queue to be executed by the GPU execution engine. To check the completion of a kernel operation, we put a CUDA event, a synchronization primitive, behind each kernel request in the same CUDA stream. Later, the completion of each kernel operation can be confirmed by querying the inserted CUDA event.

D. Scheduling Read, Copy, and Kernel Operations

Fig. 9 shows the scheduling architecture of Demand Layering, where a scheduling thread (on CPU) *releases* asynchronous read, copy, and kernel requests. Once requested, the pending read requests in the AIO queue are sequentially processed by the AIO thread; the pending copy requests in the *copy stream* are processed by the GPU copy engine; the pending kernel requests in the *kernel stream* are executed by the GPU execution engine. Since all the read, copy, and kernel requests are asynchronous, the scheduling thread can freely release a request at any time without any blocking.

Even with the freedom in scheduling, there are strict scheduling constraints to maintain the system's reliability. We

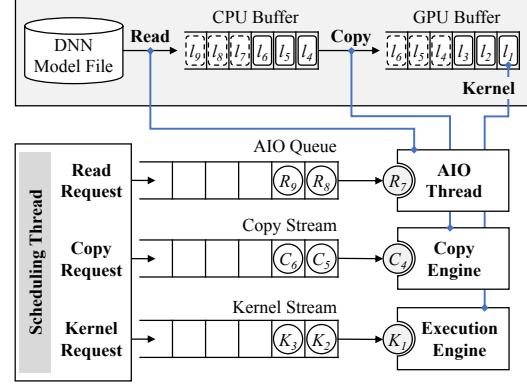


Fig. 9: Scheduling of read, copy, and kernel operations.

define three kinds of such constraints: (i) *release constraints*, (ii) *completion constraints*, and (iii) *target buffer constraints*. A release constraint is denoted by $\mathcal{R}_{prev} \rightarrow \mathcal{R}_{next}$, where an arbitrary request \mathcal{R}_{next} cannot be released before the release of \mathcal{R}_{prev} . A completion constraint is denoted by $\mathcal{R}_{prev} \mapsto \mathcal{R}_{next}$, where a request \mathcal{R}_{next} cannot be released before the completion of \mathcal{R}_{prev} . A target buffer constraint is denoted by $\mathcal{R} \rightsquigarrow CPU(s)$ or $\mathcal{R} \rightsquigarrow GPU(s)$, where a request \mathcal{R} can be released only when a contiguous memory area with the size of s in the target CPU memory or GPU memory is available. Then our scheduling constraints can be formally defined as follows:

$$\begin{aligned} R_i &\rightarrow R_{i+1}, C_i \rightarrow C_{i+1}, K_i \rightarrow K_{i+1}, \\ R_i &\mapsto C_i, C_i \mapsto K_i, \\ R_i &\rightsquigarrow CPU(s_i), C_i \rightsquigarrow GPU(s_i), \end{aligned} \quad (3)$$

meaning that every release of a request in the same type must be in the strict order of layers; a copy request can only be released after that layer's read operation is completed; a kernel request can only be released after its corresponding copy operation is completed; a read request cannot be released until the available CPU memory is enough for the layer size; a copy request cannot be released until the available GPU memory is enough for the layer size.

V. PIPELINE SCHEDULE OPTIMIZATION

A. Sequential Architecture

The most naïve scheduling method for Demand Layering is to execute read, copy, and kernel operations sequentially without overlapping their executions, as in Fig 10a. In this method, we just need a CPU buffer and a GPU buffer both with the size of s^{max} , making its memory requirement fall from the preloading architecture's $2 \times \sum_{i=1}^N s_i$ in Eq.(2) to

$$2 \times s^{max}, \quad (4)$$

which is a significant drop from the order of the entire model to the order of a layer. For example, YOLOv4 has a 92.7% reduction (from 491.6 MB to 36.0 MB). However, the sequential architecture causes a significant amount of delay

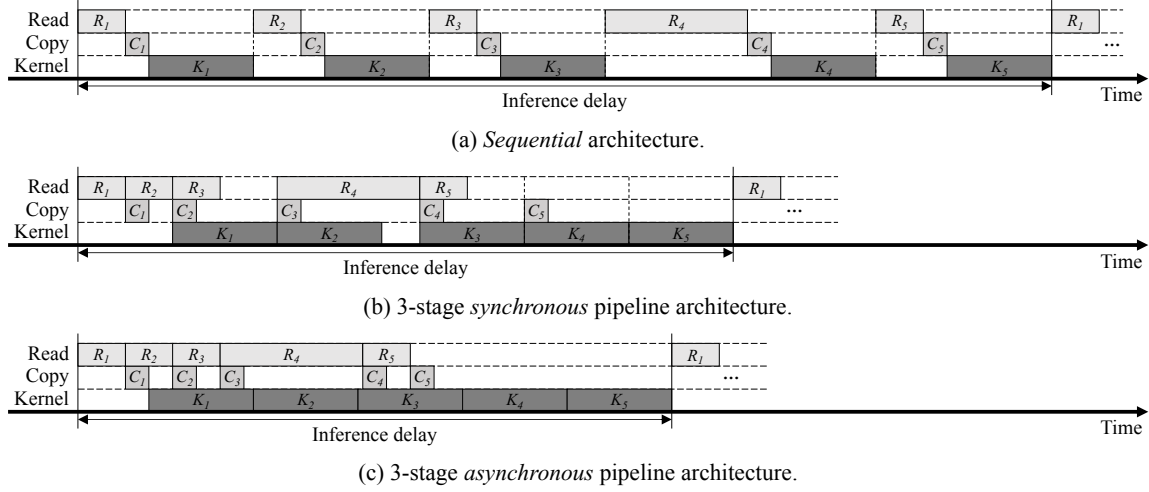


Fig. 10: Pipeline scheduling algorithms, illustrated by a simplified 5-layer DNN.

overhead, as illustrated in Fig 10a by the GPU idling intervals during read and copy operations, increasing the inference delay to

$$\sum_{i=1}^N (r_i + c_i + k_i) \quad (5)$$

from the preloading architecture's $\sum_{i=1}^N k_i$ in Eq.(1).

B. Synchronous Pipeline

To minimize the delay overhead of the sequential architecture, our next approach is to employ a pipeline architecture as illustrated in Fig. 10b. This pipeline architecture is possible due to the parallelizable nature of read, copy, and kernel executions. In this 3-stage *synchronous* pipeline architecture, the three pipeline stages (i.e., read, copy, and kernel stages) advance with a single synchronized pipeline cycle. To implement such architecture, we need to employ the *double-buffering* scheme because R_i must be writing to a CPU buffer while C_{i-1} is moving data from the same CPU buffer to a GPU buffer simultaneously. The same applies to GPU memory because C_{i-1} must be writing to a GPU buffer while K_{i-2} is executing its layer in the same GPU buffer. Due to this double-buffering constraint, its memory requirement is increased to

$$2 \times 2 \times s^{max}, \quad (6)$$

which is double the sequential architecture's requirement.

Despite the increased memory requirement, due to the pipeline architecture, most read and copy operations are hidden behind kernel operations since kernel operations are mostly the longest. However, certain long read operations, such as R_4 in Fig. 10b, can dominate certain pipeline cycles. Considering such scenarios, the inference delay can be generally calculated as

$$r_1 + \max(\{r_2, c_1\}) + \sum_{i=1}^{N-2} \max(\{r_{i+2}, c_{i+1}, k_i\}) + \max(\{c_N, k_{N-1}\}) + k_N, \quad (7)$$

by incorporating all the pipeline cycles ranging from R_1 to K_N . As illustrated in Fig. 10b, this pipelined execution significantly reduces the delay overhead from the sequential architecture.

C. Asynchronous Pipeline

To minimize the possible GPU idling intervals in the synchronous pipeline architecture, our next approach is to employ the asynchronous pipeline [21], where pipeline stages advance at their own paces without being synchronized by a single pipeline cycle. Fig 10c shows how the asynchronous pipeline differs from the synchronous one in Fig 10b. As illustrated in the figure, R_4 can be released right after the completion of R_3 , which in turn eliminates the GPU idling interval between K_2 and K_3 that existed in the synchronous architecture. In the asynchronous pipeline architecture, we can reduce the memory requirement to

$$2 \times s^{max}, \quad (8)$$

since we no longer need the double-buffering scheme. Only a single pair of CPU and GPU buffers, both with the size of s^{max} , suffices to execute the entire DNN. In contrast to the buffers in the synchronous architecture, which store only one layer at a time, the buffers in the asynchronous architecture are designed as *circular queues* that can hold multiple pending layers, as illustrated in the top of Fig. 9.

However, due to its complexity, the scheduling thread for the asynchronous pipeline architecture should be carefully designed considering all the scheduling constraints in Section IV-D. Algorithm 1 describes our scheduling algorithm. The while loop is a busy loop deciding when to release requests asynchronously until there remains no more operation to schedule. The algorithm handles four indices, where i , j , and k denote the read, copy, and kernel requests next to be scheduled. In addition, K_h denotes the foremost kernel operation that is executing in GPU. **Lines 3-6** take care of

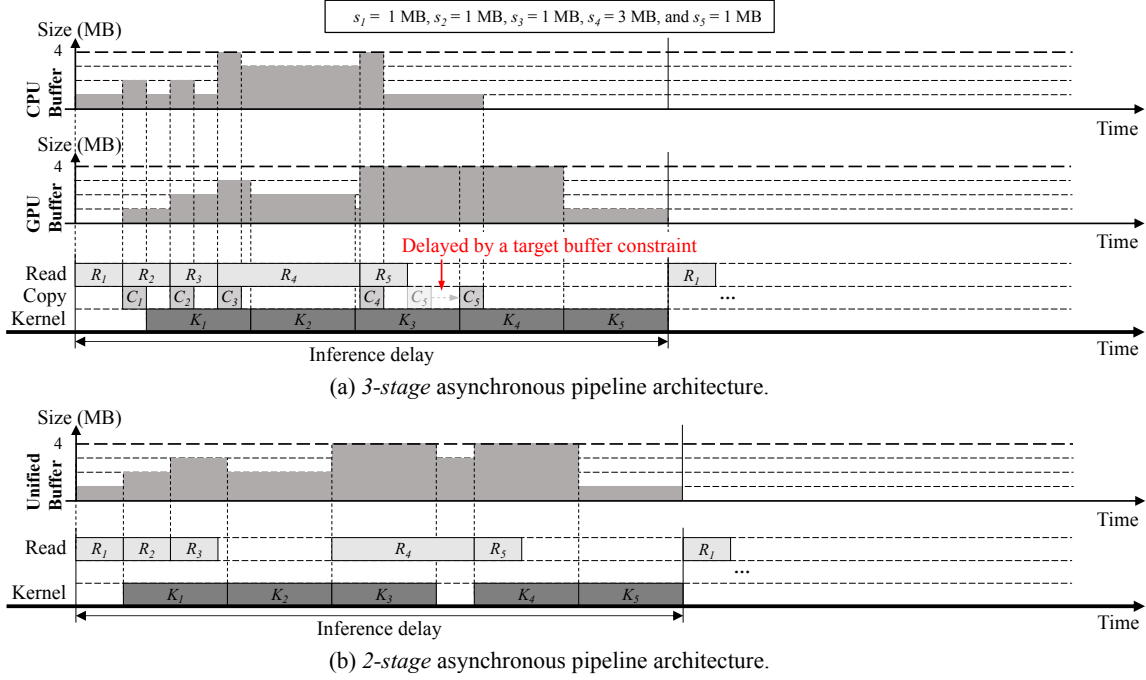


Fig. 11: 3-stage vs. 2-stage pipeline architecture.

the completion of the currently executing kernel operation by freeing the allocated amount of GPU buffer and advancing the index h . The constraint $R_i \rightsquigarrow CPU(s_i)$ is checked in **Lines 7-12** for submitting a read request. **Lines 13-18** check the constraints $R_j \mapsto C_j$ and $C_j \rightsquigarrow GPU(s_j)$ for a copy request. **Lines 19-24** also consider the constraint $C_k \mapsto K_k$ for a kernel request.

D. Iterative Optimization by Memory-Delay Tradeoff

In the asynchronous pipeline architecture, we have the freedom to increase buffer sizes beyond the minimum requirement to exploit the memory-delay tradeoff. For example, if the buffer sizes are large enough such that the entire model can be brought into memory, the inference delay can be very close to the optimal delay of the preloading architecture. With this insight, we additionally propose an iterative delay minimization method. In this method, the optimization process begins from the asynchronous pipeline's minimal memory requirement ($2 \times s^{max}$) and iteratively increases the buffer sizes with a step size (denoted by δ) until the delay does not decrease any more. We call this a *minimal delay* configuration.

To be precise, the iterative optimization method has two phases for finding the minimal delay point. The first phase records every resulting average delay by gradually increasing the buffer sizes until the maximum (i.e., the model size), by which the minimal delay point can be found. In some cases, multiple points can have the same minimal delay since the delay is not monotonically decreasing. Then, in the second phase, the recorded results are revisited from the beginning, which stops when we first encounter the minimal delay.

VI. BLEEDING-EDGE OPTIMIZATION FOR XAVIER SOCs

Note: This section further optimizes the memory usage, assuming some features of the most brand new Nvidia Xavier SoCs. Thus, this section's optimization method can possibly cause unknown stability issues in other hardware platforms.

Zero-copy memory. The CUDA runtime provides *unified memory* that provides a single address space across CPU and GPU memory, eliminating the need for copies in the programmer's perspective. However, in dGPU systems, copy operations are unavoidable. Thus, they are executed in the background by the CUDA runtime. In contrast, in iGPU systems, *zero-copy* memory management can be implemented since CPU and GPU share a common memory device, as in Fig. 7c. Our experimental platform provides the following zero-copy memory types: (i) host-pinned zero-copy memory by `cudaHostAlloc()` with a `cudaHostAllocMapped` option and (ii) managed memory by `cudaMallocManaged()` with a `cudaMemAttachHost` option. They are almost identical, except that managed memory reportedly does not allow simultaneous accesses from CPU and GPU [16]. However, our experiment found that it works just as we wanted, even with slightly better performance than host-pinned zero-copy memory. Thus, we decided to use managed memory.

Two-stage pipeline. With zero-copy memory, we no longer need the copy stage, enabling a 2-stage pipeline. Fig. 11 compares the difference between the 3-stage and 2-stage asynchronous pipelines. The figure also illustrates how the buffers are utilized. Note that the schedule in Fig. 11a is slightly different from Fig. 10c, because Fig. 11a additionally

Algorithm 1 Asynchronous pipeline scheduling

Require: $\mathbb{R} = \{R_1, R_2, \dots, R_N\}$
Require: $\mathbb{C} = \{C_1, C_2, \dots, C_N\}$
Require: $\mathbb{K} = \{K_1, K_2, \dots, K_N\}$
Require: b^c \triangleright Free CPU buffer (queue) size
Require: b^g \triangleright Free GPU buffer (queue) size

```

1:  $h, i, j, k \leftarrow 1$ 
2: while  $\mathbb{R} \neq \emptyset$  or  $\mathbb{C} \neq \emptyset$  or  $\mathbb{K} \neq \emptyset$  do
3:   if ISFINISHED( $K_h$ ) then
4:      $b^g \leftarrow b^g + s_h$ 
5:      $h \leftarrow h + 1$   $\triangleright h$ : foremost kernel index
6:   end if
7:   if  $b^c \geq s_i$  then
8:     REQUESTASYNC( $R_i$ )
9:      $\mathbb{R} \leftarrow \mathbb{R} \setminus \{R_i\}$ 
10:     $b^c \leftarrow b^c - s_i$ 
11:     $i \leftarrow i + 1$   $\triangleright i$ : next read index to be requested
12:  end if
13:  if ISFINISHED( $R_j$ ) and  $b^g \geq s_j$  then
14:    REQUESTASYNC( $C_j$ )
15:     $\mathbb{C} \leftarrow \mathbb{C} \setminus \{C_j\}$ 
16:     $b^g \leftarrow b^g - s_j$ 
17:     $j \leftarrow j + 1$   $\triangleright j$ : next copy index to be requested
18:  end if
19:  if ISFINISHED( $C_k$ ) then
20:    REQUESTASYNC( $K_k$ )
21:     $\mathbb{K} \leftarrow \mathbb{K} \setminus \{K_k\}$ 
22:     $b^c \leftarrow b^c + s_k$ 
23:     $k \leftarrow k + 1$   $\triangleright k$ : next kernel index to be requested
24:  end if
25: end while
  
```

considers the target buffer constraints ($R_i \rightsquigarrow CPU(s_i)$ and $C_i \rightsquigarrow GPU(s_i)$), assuming 4 MB buffer sizes, which were not considered in Fig. 10c. More specifically, in Fig. 11a, C_5 cannot be released right after R_5 , because between the completion of R_5 and the completion of K_3 , the GPU buffer is full. After the completion of K_3 , C_5 can be released after reserving the just freed 1 MB space. In this method, the memory requirement is reduced to

$$s^{max}, \quad (9)$$

which is *memory-optimal* with just a single layer size. There might be a misunderstanding that the 2-stage architecture will provide shorter delays than the 3-stage counterpart. By comparing Fig. 11a and Fig. 11b, however, the delay of the 2-stage architecture is even larger than the 3-stage architecture, which is due to the reduced total buffer size that makes the read operations feel more challenging to run due to the deteriorated target buffer constraint. For example, R_4 is heavily delayed by its target buffer constraint, leading to a GPU idling interval between K_3 and K_4 as a consequence. Thus, the primary objective of the 2-stage pipeline is to make the *minimal memory* configuration, not to reduce the inference delay.

TABLE I: DNN models for the evaluation.

Model	Model size	Number of layers	Maximum layer size	Default resolution
YOLOv3 [27]	236.5 MB	107	18.0 MB	608 x 608
YOLOv4 [20]	245.8 MB	162	18.0 MB	608 x 608
YOLOv4-P6 [28]	487.2 MB	305	36.0 MB	640 x 640
ResNet152 [29]	219.6 MB	206	9.0 MB	608 x 608
DenseNet201 [30]	65.6 MB	306	5.9 MB	608 x 608

VII. EXPERIMENTS

A. Experimental Setup

We implemented Demand Layering for Nvidia iGPU platforms.¹ Our experimental platform is Nvidia Jetson AGX Xavier with 16 GB *shared* DRAM, an 8-core ARM CPU, and a 512-core integrated Volta GPU. Also, it is equipped with a Samsung 980 PRO NVMe M.2 SSD (1 TB) with its official sequential read performance of 7000 MB/s. As our software platform, we use Nvidia Ubuntu 18.04.6 LTS with CUDA 10.2 and JetPack 4.6.1. As our baseline implementation, we use Darknet [22], currently available at [26], which is one of the most famous state-of-the-art DNN frameworks. Darknet is written in C and CUDA, making it portable across various hardware platforms. In Darknet, a DNN model is stored in two separate files. One is a .cfg (text) file that describes the DNN's layer-by-layer structure, including the order and types of layers. The other is a .weights (binary) file, a sequence of parameters that follows the order of layers in the .cfg file. When referring to a DNN model file in this study, it specifically means the .weights file. We use a minutely customized .weights file format to eliminate unnecessary computations in read operations, which is detailed in our GitHub repository.

Our current implementation does not support concurrent multi-DNN executions. Thus, all the results are from single-DNN experiments (i.e., one DNN at a time) but with various DNN models. Since real-time scheduling is only meaningful in multitasking environments, we use the stock Linux scheduler rather than its real-time variants. This limitation will be addressed in our future work.

We use the five DNN models as detailed in Table I. Three of them are from the YOLO object detector family. The remaining two are ResNet152 and DenseNet201, which will be referred to as ResNet and DenseNet in short, respectively. YOLOv4-P6 has the largest model size (487.2 MB) and the largest single layer size (36.0 MB). DenseNet is the smallest (65.6 MB) with the largest number of layers (306). In contrast, YOLOv3 has the smallest number of layers (107). The largest layer size (s^{max}) varies from 5.9 MB (DenseNet) to 36.0 MB (YOLOv4-P6). Unless otherwise stated, the default input resolution is 608×608 except YOLOv4-P6. As an exception, YOLOv4-P6 has a different default input resolution of 640×640 due to its architectural limitation.

¹The source code is at <https://github.com/aveeslab/demand-layering>.

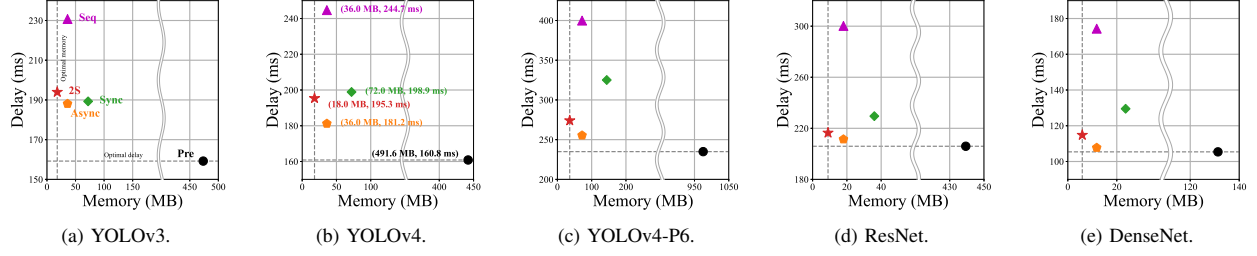


Fig. 12: Memory and delay optimization results.

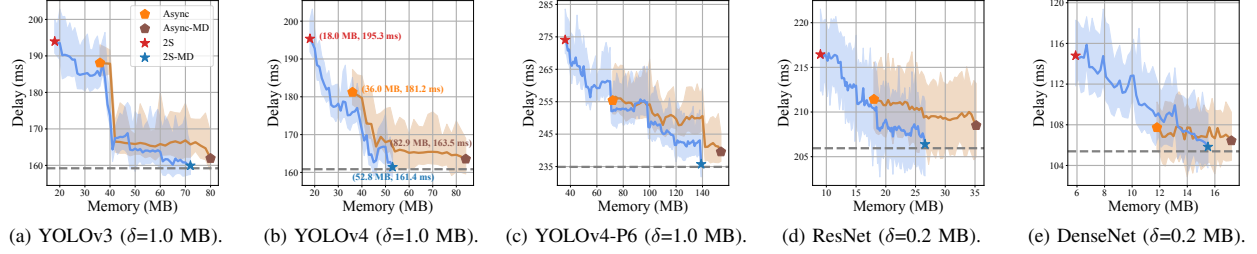


Fig. 13: Iterative delay optimization to minimal delay configurations.

TABLE II: Summary of normalized memory and delay optimization results.

Model	Seq		Sync		Async		Async-MD		2S		2S-MD	
	Memory	Delay	Memory	Delay	Memory	Delay	Memory	Delay	Memory	Delay	Memory	Delay
YOLOv3	(-92.4%, +44.9%)		(-84.8%, +18.9%)		(-92.4%, +18.1%)		(-83.1%, +1.7%)		(-96.2%, +21.8%)		(-84.8%, +0.5%)	
YOLOv4	(-92.7%, +52.1%)		(-85.4%, +23.7%)		(-92.7%, +12.7%)		(-83.1%, +1.7%)		(-96.3%, +21.5%)		(-89.3%, +0.4%)	
YOLOv4-P6	(-92.6%, +70.2%)		(-85.2%, +38.4%)		(-92.6%, +8.7%)		(-84.2%, +2.0%)		(-96.3%, +16.7%)		(-85.7%, +0.3%)	
ResNet	(-95.9%, +45.8%)		(-91.8%, +11.4%)		(-95.9%, +2.7%)		(-92.0%, +1.2%)		(-98.0%, +5.1%)		(-93.9%, +0.2%)	
DenseNet	(-91.0%, +65.4%)		(-82.0%, +22.9%)		(-91.0%, +2.0%)		(-86.9%, +1.0%)		(-95.5%, +8.9%)		(-88.2%, +0.4%)	

B. Evaluation Results

We compare the following pipeline architectures:

- **Pre:** *Preloading* architecture.
- **Seq:** *Sequential* architecture (Section V-A).
- **Sync:** 3-stage *synchronous* pipeline (Section V-B).
- **Async:** 3-stage *asynchronous* pipeline (Section V-C).
- **Async-MD:** *Minimal delay* configuration from **Async**.
- **2S:** 2-stage *asynchronous* pipeline (Section VI).
- **2S-MD:** *Minimal delay* configuration from **2S**.

For the iterative optimization for finding minimal delay configurations, refer to Section V-D.

Fig. 12 shows the memory requirement (x-axis) and inference delay (y-axis) of the considered pipeline architectures. Every data point in the figure is an average of 1000 iterations. In particular, Fig. 12b shows the results for YOLOv4, where **Pre** requires a huge amount of memory (491.6 MB) with YOLOv4’s optimal delay (160.8 ms). With **Seq**, the memory requirement is significantly dropped (to 36.0 MB) at the cost of a delay increase (to 244.7 ms). By applying **Sync**, the delay is somewhat reduced (to 198.9 ms), however, at the cost of a memory increase (to 72.0 MB) due to the double-buffering

scheme. With **Async**, the memory requirement goes back to 36.0 MB) and the delay is preferably reduced (to 181.2 ms). By applying **2S**, the memory requirement is finally optimal, which is a single layer size ($s^{max} = 18.0$ MB), with a slight delay increase (to 195.3 ms). The other DNNs show similar patterns.

Fig. 13 shows the iterative delay optimization process and the resulting minimal delay configurations. This optimization method is only applicable to asynchronous pipeline architectures (i.e., **Async** and **2S**), where we can freely adjust the buffer size above the minimum requirement. In the figure, the solid lines depict average delays as gradually increasing buffer sizes, while the colored areas illustrate min-max delay ranges. In particular, Fig. 13b shows the results of YOLOv4. During the optimization, **Async** begins at (36.0 MB, 181.2 ms) and optimized to (82.9 MB, 163.5 ms). In comparison, **2S** begins at (18.0 MB, 195.3 ms) and optimized to (52.8 MB, 161.4 ms). In both cases, the delay overhead is significantly minimized, closely approaching the optimal delay (160.8 ms) depicted by a dashed line.

Table II summarizes the normalized memory reduction and delay overhead from the baseline architecture **Pre** that has

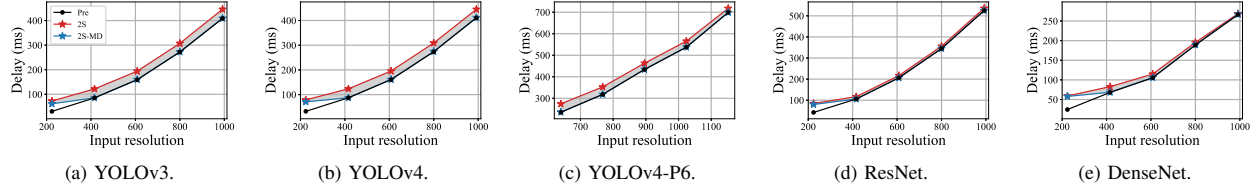


Fig. 14: Delay optimization results with varying input image resolutions.

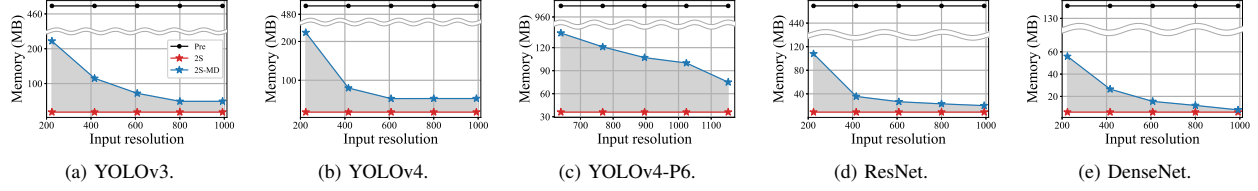


Fig. 15: Memory optimization results with varying input image resolutions.

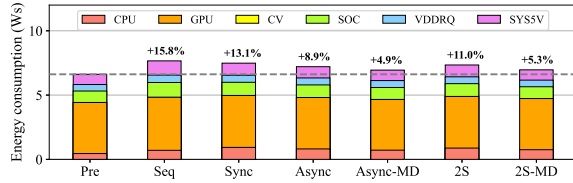


Fig. 16: Comparison of per-inference energy consumption of YOLOv4.

the maximal memory requirement with the optimal delay. In general, **2S** shows the most significant memory reductions (-96.5% on average), while **2S-MD** shows minimal delays close to the zero-delay overhead with still significant memory reductions (-88.4% on average). However, **Async** and **Async-MD** also show comparable optimization results for minimal memory and minimal delay configurations, respectively. Thus, in systems without the support of zero-copy memory, our guide is to use one of the optimization configurations between **Async** and **Async-MD** in Fig. 13. In systems supporting zero-copy memory, using the results between **2S** and **2S-MD** in Fig. 13 provides even better results. However, beware that it is still based on a bleeding-edge technology.

The input image resolution significantly impacts kernel times and the overall optimization results. Fig. 14 shows its impact on delays, where the higher input resolution generally incurs the longer delay. By comparing the five DNNs, **2S-MD** in ResNet and DenseNet particularly show near-optimal delays close to **Pre**. Since ResNet and DenseNet are well balanced with similarly sized layers, they are favorable for pipelining. Even so, when the input resolution gets too small, the gap between **2S-MD** and **Pre** visibly arises due to the reduced optimization opportunity by too short kernel times. By looking at how much delay is reduced from **2S** to **2S-**

MD (the grey area) for each DNN, all the DNNs except YOLOv4-P6 fail to reach the near-optimal delay when the input resolution gets too small (i.e., 224×224). Even for the YOLOv4-P6 case, it is not a complete experiment because we cannot lower the input resolution under a certain level due to its technical limitation. Besides, Fig. 15 shows how much memory is incremented during the iterative optimization from **2S** to **2S-MD**, depicted by the grey area. The decreasing trend indicates that the memory cost spent for reducing the delay is significantly lower with higher input resolutions. In other words, the trading cost is dependent on the input resolution. With higher input resolutions, we can easily reduce delays by increasing a small amount of memory.

Fig. 16 compares the per-inference energy consumption for various architectures, broken down by the hardware components of interest. For that, our target platform has two 3-channel INA3221 power monitors so that we can extract the energy consumption of the six hardware components through the /proc interface. In the graph, SOC denotes the Xavier SoC portion, excluding the CPU, GPU, and vision accelerator (CV) portions; VDDRQ denotes the external DRAM portion; SYS5V represents the remaining portion including attached I/O devices. It is certain that **Pre** exhibits the minimal energy consumption. By using Demand Layering, the energy consumption is somewhat increased due to the additional read and copy operations during the inference phase. Also, we need a scheduling thread that did not exist in **Pre**, making a CPU core busy. Although the total amount of work (i.e., read, copy, and kernel operations) is the same across the remaining six Demand Layering architectures, there is a slight variance among them that has a strong correlation with their inference delays, because longer delays incur more baseline energy consumption during an inference. However, the additional energy consumption by Demand Layering is not significant.

VIII. RELATED WORK

DNN frameworks. There are various DNN frameworks [22]–[24], [31]–[33], supporting both training and inference tasks. Besides, there are frameworks specialized in inference such as TensorRT [34], TensorFlow Lite [35], TensorFlow Lite Micro [36], and TinyEngine [13]. To the best of our knowledge, the above frameworks commonly employ the preloading architecture, incurring significant memory overhead that is our concern. Among them, we use Darknet as our baseline implementation.

Model optimization. Most DNN frameworks have their own model file formats, making it challenging to exchange models across frameworks. There are standardization efforts to alleviate this problem, such as ONNX [37]. Before deploying models, it is a usual practice to optimize the model such that it can run faster with less memory. For that, various model compression techniques are used, such as pruning and quantization [1]–[3]. Additionally, there are compiler frameworks [38]–[41] that optimize given models to target hardware platforms, including FPGAs and custom ASICs. Our approach does not compete with the above optimization techniques. In contrast, our method accepts such optimized models and can still significantly reduce memory usage without modifying the model itself.

DNN memory optimization. To minimize the memory usage of DNN inference systems, TensorFlow Lite reuses activation buffers that store intermediate results between layers [12]. [13] and [14] optimize memory usage for MCU hardware by combining neural architecture optimizations and cross-layer patch-based computations. [42] reduces peak memory usage by reordering layer executions. [43] reduces off-chip memory usage by fusing multiple CNN layers to utilize on-chip memory efficiently. [44] reduces memory usage by swapping out tensors to external flash storages. [45] reduces on-chip memory usage and I/O bandwidth by doing calculations across layers. Most of the above studies reduce memory usage by activation buffers. In contrast, we minimize memory usage by model parameters. Also, they do not consider GPU-based systems.

GPU memory optimization. Due to the scarcity of GPU memory, there have been many studies [46]–[51] that minimize GPU memory usage when training large DNNs, which is not this study's concern. In contrast, we focus on minimizing a DNN inference system's GPU memory usage to deploy large DNNs on embedded systems with limited memory. With the same motivation, SwapAdvisor [15] provides a general method that utilizes inexpensive CPU memory as a swap device of scarce GPU memory. In dGPU systems, it is promising. However, reducing GPU memory at the cost of increased CPU memory is not meaningful in iGPU systems. Refer to Section II-B for more information. In [16], a memory/performance co-optimization framework for multitasking GPU applications is proposed, where three different GPU memory types are empirically analyzed. Although it does not utilize the layer-by-layer execution of DNNs, the analysis results gave us great insight.

Layer-by-layer DNN execution. There are studies exploiting the layer-by-layer execution of DNNs. LaLaRAND [10] improves the schedulability of real-time DNN tasks by optimally allocating and scheduling individual layers to CPU and GPU. PipeSwitch [9] employs an idea of layer-by-layer DNN parameter loading and execution, which is similar to ours. However, PipeSwitch does not consider the read stage, and its goal is to enable fast context switching for multi-DNN systems, which is not related to the memory usage issue. MASA [52], [53] also uses a similar idea that resembles our layer-by-layer loading and execution. However, MASA assumes only CPUs without considering GPUs. TASO [54] is also based on a similar idea. However, it only supports convolutional layers by optimally selecting layer implementations (e.g., im2col and Winograd). In contrast, our approach generally supports any DNN layers, including fully connected layers as well as any unseen custom layers.

PREM architecture. Our method has some technical similarities with the PRedictable Execution Model (PREM) architecture [55], where each task is split into a number of I/O phases, memory phases, and execution phases, and multiple such tasks are coscheduled to avoid the shared resource (e.g., cache, memory, and bus) contention. In [56]–[59], the local memory (e.g., scratchpad memory) is partitioned into two regions such that a memory phase (by DMA controller) and an execution phase (by CPU) can run in a pipelined parallel manner. They are conceptually similar to our read, copy, and kernel pipeline stages and the double-buffering scheme in the synchronous pipeline architecture. It is interesting to note that the techniques developed for the conventional real-time task model can be revisited for emerging applications (e.g., real-time DNN inference). PREM's primary objective is to eliminate the shared resource contention for predictable systems. Thus, they do not use the asynchronous pipeline technique, which may break the predictability. In contrast, we primarily focus on the memory usage, where the asynchronous pipeline is generally a better solution.

IX. CONCLUSION

This study presents Demand Layering that minimizes the DNN inference system's memory usage for model parameters by loading and executing DNN layers in a layer-by-layer manner. To minimize the delay overhead, we designed a pipeline architecture where the read, copy, and kernel operations run in parallel. It is further enhanced by employing the asynchronous pipeline and zero-copy memory. As a result, we can reduce the memory usage by 96.5% with just 14.8% delay overhead. Also, we can achieve near-zero delay overhead with a still significant 88.4% memory reduction by exploiting the memory-delay tradeoff.

In the future, we plan to develop a deterministic delay analysis method such that a given model's inference delay can be safely bounded from the model configuration. Also, we plan to develop a scheduling algorithm for concurrent DNNs. Ultimately, our final goal is to make a deterministic multi-DNN inference architecture with minimized memory usage.

ACKNOWLEDGMENT

This work was supported partially by the BK21 Four Program (5199990814084) of the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Korea, partially by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (2022R1A2C1013197), and partially by the NSF grant CCF-1704859. Mingoo Ji's work was done while at Kookmin University. J.-C. Kim is the corresponding author of this paper.

REFERENCES

- [1] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *Proc. 4th International Conference on Learning Representations (ICLR)*, 2016.
- [2] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. 15th European conference on computer vision (ECCV)*, 2018, pp. 784–800.
- [3] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. 16th IEEE international conference on computer vision (ICCV)*, 2017, pp. 2736–2744.
- [4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.
- [5] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "GPipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, pp. 103–112, 2019.
- [6] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proc. 14th European Conference on Computer Systems (EuroSys)*, 2019, pp. 1–17.
- [7] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *Proc. USENIX Annual Technical Conference (ATC)*, 2020, pp. 307–321.
- [8] L. Bernstein, A. Sludis, R. Hamerly, V. Sze, J. Emer, and D. Englund, "Freely scalable and reconfigurable optical hardware for deep learning," *Scientific reports*, vol. 11, no. 1, pp. 1–12, 2021.
- [9] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "PipeSwitch: Fast pipelined context switching for deep learning applications," in *Proc. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 499–514.
- [10] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "LaLaRAND: Flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks," in *Proc. 42nd IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 329–341.
- [11] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *Proc. 40th IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 392–405.
- [12] Y. Pisarchyk and J. Lee, "Efficient memory management for deep neural net inference," *arXiv preprint arXiv:2001.03288*, 2020.
- [13] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han *et al.*, "MCUNet: Tiny deep learning on IoT devices," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 11 711–11 722, 2020.
- [14] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Memory-efficient patch-based inference for tiny deep learning," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, pp. 2346–2358, 2021.
- [15] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *Proc. 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1341–1355.
- [16] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform," in *Proc. 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 310–323.
- [17] K. Huang, D. Imai, T. Wang, and D. Xie, "SSDs Striking Back: The storage jungle and its implications on persistent indexes," in *Proc. 12th Annual Conference on Innovative Data Systems Research (CIDR)*, 2022, pp. 9–12.
- [18] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the Nvidia TX2: Hidden details revealed," in *Proc. 38th IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 104–115.
- [19] I. S. Olmedo, N. Capodiceci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective," in *Proc. 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 213–225.
- [20] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.
- [21] S. M. Nowick and M. Singh, "High-performance asynchronous pipelines: An overview," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 8–22, 2011.
- [22] J. Redmon, "Darknet: Open source neural networks in C," <http://pjreddie.com/darknet/>, accessed: 2022-05-22.
- [23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.
- [24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.
- [25] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding pitfalls when using Nvidia GPUs for real-time tasks in autonomous systems," in *Proc. 30th Euromicro Conference on Real-Time Systems (ECRTS)*, 2018, pp. 20:1–20:21.
- [26] A. Bochkovskiy, "Darknet: Open source neural networks in C," <https://github.com/AlexeyAB/darknet>, accessed: 2022-05-22.
- [27] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.
- [28] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-YOLOv4: Scaling cross stage partial network," in *Proc. 34th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 13 029–13 038.
- [29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
- [30] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. 30th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 4700–4708.
- [31] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM International Conference on Multimedia (MM)*, 2014, pp. 675–678.
- [32] Facebook Open Source, "Caffe2: A new lightweight, modular, and scalable deep learning framework," <https://caffe2.ai>, 2017.
- [33] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.
- [34] "Nvidia TensorRT," <https://developer.nvidia.com/tensorrt>, accessed: 2022-05-22.
- [35] P. Warden and D. Situnayake, *TinyML: Machine learning with TensorFlow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.
- [36] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "TensorFlow Lite Micro: Embedded machine learning for tinyML systems," in *Proc. 4th Machine Learning and Systems (MLSys)*, vol. 3, pp. 800–811, 2021.
- [37] "ONNX," <https://onnx.ai/>, accessed: 2022-05-22.
- [38] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, no. 3, pp. 708–727, 2020.
- [39] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.
- [40] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end opti-

- mizing compiler for deep learning,” in *Proc. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 578–594.
- [41] “XLA,” <https://www.tensorflow.org/xla>, accessed: 2022-05-22.
- [42] E. Liberis and N. D. Lane, “Neural networks on microcontrollers: saving memory at inference via operator reordering,” *arXiv preprint arXiv:1910.05110*, 2019.
- [43] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer CNN accelerators,” in *Proc. 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.
- [44] H. Miao and F. X. Lin, “Enabling large neural networks on tiny micro-controllers with swapping,” *arXiv preprint arXiv:2101.08744*, 2021.
- [45] K. Goetschalckx and M. Verhelst, “Breaking high-resolution CNN bandwidth barriers with enhanced depth-first execution,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, vol. 9, no. 2, pp. 323–331, 2019.
- [46] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, “vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *Proc. 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.
- [47] T. Sekiyama, T. Imamichi, H. Imai, and R. Raymond, “Profile-guided memory optimization for deep neural networks,” *arXiv preprint arXiv:1804.10001*, 2018.
- [48] D. Yang and D. Cheng, “Efficient GPU memory management for nonlinear DNNs,” in *Proc. 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2020, pp. 185–196.
- [49] A. Shah, C.-Y. Wu, J. Mohan, V. Chidambaram, and P. Krähenbühl, “Memory optimization for deep networks,” *arXiv preprint arXiv:2010.14501*, 2020.
- [50] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, “Capuchin: Tensor-based GPU memory management for deep learning,” in *Proc. 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 891–905.
- [51] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, “Memory-efficient pipeline-parallel DNN training,” in *Proc. 38th International Conference on Machine Learning (ICML)*. PMLR, 2021, pp. 7937–7947.
- [52] B. Cox, J. Galjaard, A. Ghiassi, R. Birke, and L. Y. Chen, “Masa: Responsive multi-DNN inference on the edge,” in *Proc. 19th IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021, pp. 1–10.
- [53] B. Cox, R. Birke, and L. Chen, “Memory-aware and context-aware multi-DNN inference on the edge,” *Pervasive and Mobile Computing (PMC)*, vol. 8, no. 101594, 2022.
- [54] Y. Wen, A. Anderson, V. Radu, M. F. O’Boyle, and D. Gregg, “Taso: Time and space optimization for memory-constrained DNN inference,” in *Proc. 32nd IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 199–208.
- [55] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, “A predictable execution model for cots-based embedded systems,” in *Proc. 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 269–279.
- [56] S. Wasly and R. Pellizzoni, “Hiding memory latency using fixed priority scheduling,” in *Proc. 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 75–86.
- [57] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, “A real-time scratchpad-centric os for multi-core embedded systems,” in *Proc. 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11.
- [58] R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo, “A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems,” *Real-Time Systems*, vol. 55, no. 4, pp. 850–888, 2019.
- [59] D. Casini, P. Pazzaglia, A. Biondi, M. Di Natale, and G. Buttazzo, “Predictable memory-cpu co-scheduling with support for latency-sensitive tasks,” in *Proc. 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.