Occamy: Memory-efficient GPU Compiler for DNN Inference

Jaeho Lee Shinnung Jeong Seungbin Song Kunwoo Kim Heelim Choi Youngsok Kim Hanjun Kim
Yonsei University
{jaeho, shin0403, seungbin, prove22, heelim, youngsok, hanjun}@yonsei.ac.kr

***Abstract*—This work proposes Occamy, a new memory-efficient DNN compiler that reduces the memory usage of a DNN model without affecting its accuracy. For each DNN operation, Occamy analyzes the dimensions of input and output tensors, and their liveness within the operation. Across all the operations, Occamy analyzes liveness of all the tensors, generates a memory pool after calculating the maximum required memory size, and schedules when and where to place each tensor in the memory pool. Compared to PyTorch, on an integrated embedded GPU for six DNNs, Occamy reduces the memory usage by 34.6% and achieves a geometric mean speedup of 1.25×.**

I. Introduction

Deep Neural Networks (DNNs) are becoming deeper and wider to achieve higher accuracy, but require larger GPU memory capacity. Demanding larger GPU memory sizes makes it difficult to execute state-of-the-art DNNs on various GPU-equipped systems, especially on embedded devices having limited GPU memory sizes. With the advance of DNN technology, each DNN model becomes deeper and wider and achieves higher accuracy. The higher accuracy of a DNN model enlarges its application to various fields such as image classification, object detection, super-resolution and natural language processing. However, the deeper and wider DNN models require additional memory space and thus limit affordable GPU devices. For example, PyTorch [1] requires 24 GB of memory space to execute an SRGAN model for the image super-resolution that commodity GPU devices can barely or hardly support. This memory starvation problem becomes severe especially for embedded devices, which have limited memory spaces like 16 GB memory of Jetson AGX Xavier [2].

To relieve the memory burden on a GPU, various optimization techniques such as memory offloading [3, 4, 5, 6], recomputation [4, 6], tensor decomposition [6] and model compression [7] have been proposed. Still, their schemes are limited to DNN training or require for programmers to change the DNN model affecting its accuracy. Since DNN training consists of two steps, a forward and backward process, the backward process requires intermediate results generated by the forward process. Memory offloading creates a memory pool by combining CPU and GPU memories and keeps the intermediate results at the CPU memory as cold data. However, unlike DNN training, DNN inference consists of only the forward process with few cold data and can be executed on an integrated GPU that shares memory with a CPU, so the memory offloading schemes are not effective. Recomputation [4, 6], tensor decomposition [6] and model compression [7] reduce the required memory size by recomputing the intermediate results of some light-weight layers instead of keeping them, by decomposing a large tensor into smaller tensors, and by reducing computation or data precision, but they change the DNN model and affect its accuracy.

Moreover, memory management operations dramatically affect the DNN inference latency. Fig. 1 shows that allocating and deallocating tensors take one-third to half of the entire latency for ResNet50 and YOLOv1 on a discrete GPU (NVIDIA Titan RTX), while their computation takes only 20 to 25 % of the entire latency. Moreover, when to allocate and deallocate tensors affects the overall latency. Fig. 2 shows that eagerly allocating all the tensors together at the beginning (Fig. 2a) reduces the overall latency of ResNet50 from 27.1 ms to 21.9 ms compared to the lazy allocation that allocates tensors when they are necessary (Fig. 2b). Thus, memory management optimization is crucial not only to reduce the required memory size but also to reduce the DNN inference latency.

Existing DNN frameworks such as PyTorch [1] and TensorFlow Lite Micro [8] reduce the memory allocation and deallocation overheads with the eager memory allocation and a memory pool, but their schemes do not optimize the required memory size. Although PyTorch allocates all the tensors used in a DNN model at the beginning and reduces the memory allocation latency, the tensors are allocated earlier than when they are used, requiring additional unnecessary memory spaces. For example, the gray area in Fig. 2a is the time period from when a tensor is allocated and when the tensor is used for the first time, showing the unnecessarily allocated tensors. Compared to the laze allocation scheme in Fig. 2b, the eager allocation scheme wastes huge memory spaces to reduce memory allocation latency. Instead of allocating and deallocating a memory space, TensorFlow Lite Micro [8] and TASO [9] reuse allocated memory spaces with a memory pool. Although the memory pool schemes reduce the memory allocation and deallocation latency while not increasing the required memory size, the memory pool suffers from fragmentation problems because the schemes allocate tensors in their memory pool within reflecting tensors used in succeeding layers. Therefore, a new memory management scheme that can reflect the entire DNN model is required.

This work proposes Occamy, a new memory-efficient GPU compiler for DNN inference. Occamy analyzes dimensions of input and output tensors for each DNN operation, and their liveness within the operation. If the liveness results of two tensors are not overlapped within the same operation, Occamy makes the two tensors share the same memory space (tensor coalescing). Occamy fuses two DNN operations into another operation if possible, and reduces memory operations between the operations (layer fusion). Then, Occamy analyzes memory access patterns of all the tensors based on the liveness analysis results, calculates the maximum required memory size for the entire DNN model, and generates a memory pool with the maximum size (memory access pattern analysis). Finally, Occamy schedules when and where to place each tensor in the memory pool, and inserts memory management instructions, so each DNN model can efficiently use the memory pool without suffering from the fragmentation problem.

This work implements Occamy on the top of the MLIR compiler framework [10] by extending the ONNX-MLIR compiler project [11]. This work extends the CPU-based original ONNX-MLIR compiler to support GPUs and evaluates Occamy with the six DNN inference models such as ResNet50 [12], MobileNet [13], SDD-ResNet50 [14], BERT [15], YOLOv1 [16], and SRGAN [17] using an integrated GPU system (Jeston AGX Xavier [2]) and a discrete GPU system (NVIDIA GeForce RTX 3090 with 24GB memory and Inter® Core™ i7-8700). Compared to PyTorch with JetPack SDK and PyTorch, Occamy reduces memory usage by 31.2% and 37.7% and achieves geomean speedups of 1.29 and 1.21 times on the integrated GPU and the discrete GPU systems.

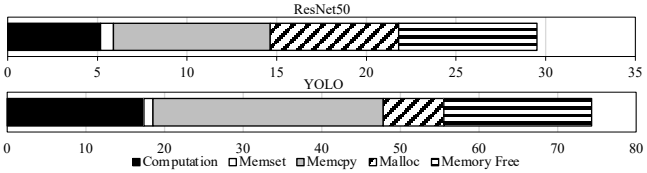
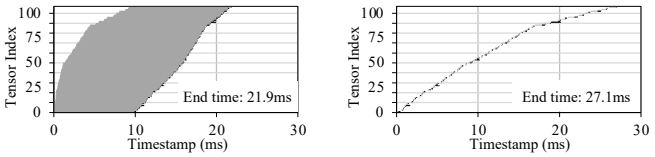


Fig. 1: The execution time of memory and computation operations of ResNet50 and YOLO.



(a) Eager memory allocation (b) Lazy memory allocation

Fig. 2: Liveness of tensors in ResNet50 for the eager and lazy memory allocation schemes. The gray area means the period when a tensor is allocated but not yet used, and the black area means the period when the tensor is actually used.

II. Motivation

Dynamic memory allocation on a GPU easily becomes a bottleneck of a SIMD/SIMT application [18, 19, 20]. To allocate virtual memory and to map the virtual page to its corresponding physical memory page on a GPU, the dynamic memory allocation requires OS system calls. Moreover, to provide consistency of the memory (de)allocation, the OS protects the memory (de)allocation with synchronization primitives (e.g., mutex). To enable concurrent allocations and deallocations of multiple applications, synchronization primitives (e.g., mutex) protect memory (de)allocation calls and provide consistency of the memory (de)allocations. Because the (de)allocations are atomic, concurrent invocations on memory (de)allocations can cause contention and synchronization overheads [18].

This work analyzes the dynamic memory (de)allocation overheads in a DNN inference model by measuring the execution time of memory management and computation operations for ResNet50 and YOLO on a discrete GPU (NVIDIA Titan RTX). As Fig. 1 illustrates, the memory allocation and deallocation take one-third to half of the entire latency, while their computation takes only 20% to 25% of the entire latency. Unlike the memory copy overhead that can be hidden by the asynchronous copy and unified memory schemes [21, 22, 23, 24], the memory (de)allocation cannot be hidden because of the synchronization primitives.

To relieve the memory allocation overheads, widely-used DNN frameworks such as PyTorch [1] eagerly allocate all the tensors to the GPU memory at the beginning of a DNN model. For example, PyTorch allocates the GPU memory spaces for each tensor, copies all required tensors to the memory, and executes layers using the cached tensors. This eager memory allocation can reduce the inference time from 27.1 ms to 21.9 ms (Fig. 2) because there is no OS intervention or a synchronization point caused by memory (de)allocation during the layer execution. However, eagerly allocating all the tensors requires a large amount of the GPU memory space to hold a whole DNN model and sometimes fails to run the model in a GPU. For example, PyTorch requires 24 GB of GPU memory to execute SRGAN [17], but Jetson AGX Javier has only 16 GB of the unified memory, which is not enough to run SRGAN model.

On the other hand, the on-demand (lazy) memory allocation can reduce the memory footprints of DNN models as shown in Fig. 2b. The lazy memory allocation scheme allocates only required tensors to the GPU memory before executing each layer and frees them after the layer. Since only the tensors used in each layer occupy the GPU memory, the scheme minimizes the GPU memory usage. However, the lazy memory allocation degrades the inference time because of frequent system calls and synchronization caused by (de)allocations.

Recent works [3, 4, 25] propose runtime systems that trace tensor memory access patterns to prefetch and evict the tensors on the training phase efficiently. However, tracking memory access patterns at runtime causes the profiling overhead. While the profiling is effective for DNN training which repeatedly executes DNN models in forward and backward ways, it is not suitable for DNN inference which executes DNN models once.

To reduce the required memory size and the DNN inference time, a new smart memory management scheme is necessary. The eager memory allocation scheme reduces the inference time at the expense of the additional GPU memory space. On the other hand, the lazy memory allocation scheme minimizes the required memory size but is slow due to frequent system calls and synchronization. To exploit the strengths of the both schemes, it is necessary to allocate the minimal memory space at the beginning and reuse them throughout the entire inference, thus avoiding the system call and synchronization overheads. Moreover, since the size of tensors is fixed at each DNN model, the lifetime of each tensor and the minimal required memory space can be statically calculated at the compile time without causing runtime overhead.

III. Design of the Occamy Compiler

This work proposes Occamy, a new memory-efficient DNN compiler. Fig. 3 illustrates the overall compilation process of Occamy. Occamy transforms an input DNN model described in the ONNX model into its corresponding DNN library invocations with GPU memory management instructions written in LLVM IR. Occamy consists of three important optimization steps. First, Occamy analyzes the liveness of tensors and converts the DNN model in ONNX IR into DNN IR code with memory management instructions (§III-A). Second, Occamy optimizes the DNN IR code at the operation level, such as kernel fusion that eliminates inefficient tensor reloads between layers, and tensor coalescing that reuses memory spaces for input and output tensors by coalescing them (§III-B). Third, Occamy analyzes the memory access patterns of tensors, calculates the maximum memory usage, and generates a memory pool with memory management instructions (§III-C). With the static memory pattern analysis and the memory pool, Occamy can reduce the maximum memory usage like the lazy memory allocation without suffering from system call and synchronization overheads from (de)allocation like the eager memory allocation. After finishing three optimization steps, Occamy converts the DNN IR code to the LLVM IR code, and generates an executable DNN binary.

A. Liveness-aware Memory Operation Insertion

Occamy first analyzes the liveness and dimensions of all the tensors in a DNN model, and inserts memory management operations such as allocation, deallocation and copy to allow succeeding optimization steps to explicitly manipulate the operations. Here, Occamy adopts the lazy memory allocation scheme for further optimization.

Tensor Liveness Analyzer. Occamy analyzes the liveness of each tensor and stores the result as a table. The liveness table consists of the first-use and last-use indices of each tensor, and their memory sizes. Occamy will use the memory size to analyze the detailed memory usage patterns and to calculate the size of a memory pool.

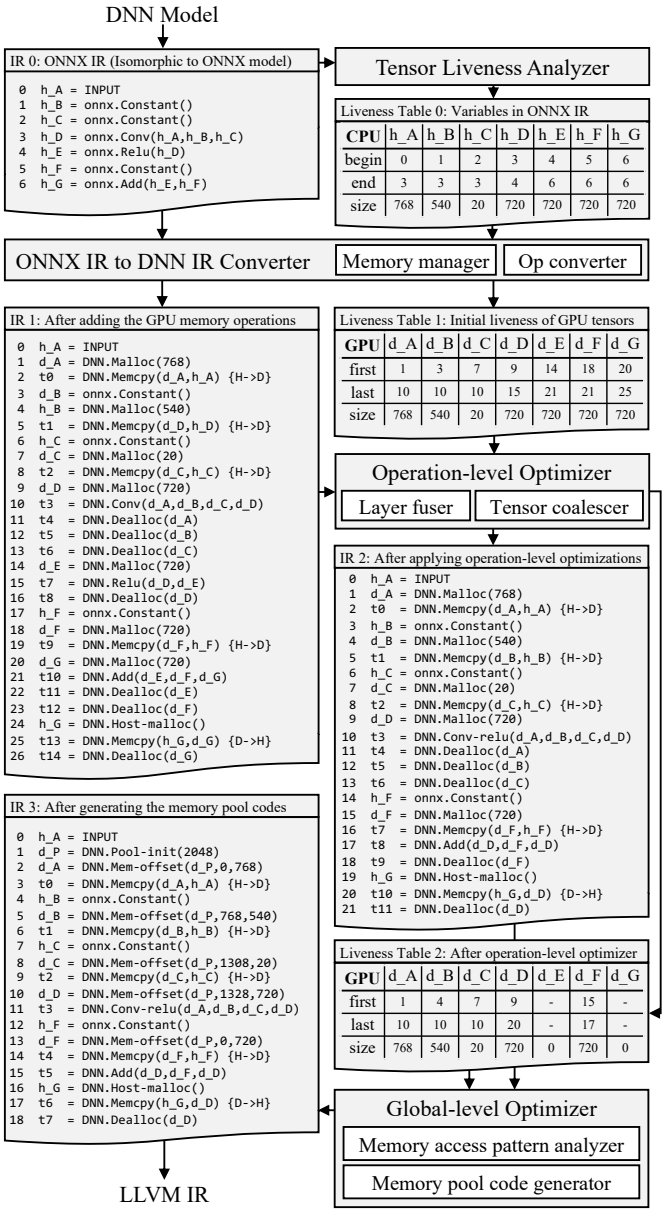


Fig. 3: Overall compilation flow of the Occamy compiler.

Algorithm 1 shows how Occamy analyzes the liveness of tensors in the input operator list. For each operator, the liveness analyzer updates the last use of each operand (line 3 to line 5). Then, the analyzer generates a new table entry for the operator and update member of entries such as first-use, last-use, and size (line 6 to line 8). Liveness Table 0 of Fig. 3 shows an example of a liveness table generated by the tensor liveness analyzer. For example, variable `h_A` is defined in line 0 of the ONNX IR, and the last use of variable `h_A` is depicted in line 3. Therefore, the liveness of variable `h_A` begins in line 0 and ends in line 3, shown in the result table.

ONNX IR to DNN IR Converter. Based on the liveness analysis result, the ONNX IR to DNN IR converter inserts memory management operators, and changes ONNX operations to DNN operations that explicitly denote memory spaces for input and output tensors. Occamy inserts GPU memory allocation and deallocation operators for each tensor after the definition and the last use, respectively. To

Algorithm 1: Tensor Liveness Analyzer

Input: *opList*: Operator lists of DNN inference model
Output: *LA*: Liveness table recording begin, end, and size of variables

```
1  Function LivenessAnalysis (LA, opList) :
2      for op ∈ opList do
3          for operand ∈ op.operands do
4              LA[operand].end ← op
5          end
6          LA[op].begin ← op      // op returns a new tensor
7          LA[op].end ← op
8          LA[op].size ← op.size  // computed from dimension
9      end
10     return LA
11 end
```

copy the predefined input, weight and bias values from CPU to GPU, Occamy inserts memory copy operators after their memory allocation operators like line 2 of IR 1 of Fig. 3. Occamy also inserts memory copy operators to copy the final output data from GPU to CPU like line 25 of IR 1 of Fig. 3. Occamy transfers only the final output data to CPU instead of every intermediate result, thus reducing communication overheads between CPU and GPU. Finally, the converter transforms DNN operators of ONNX IR such as ONNX.conv to corresponding DNN IR operators such as DNN.conv for further optimizations.

B. Operation-level Optimizer

Occamy adopts two operation-level optimizations to reduce memory usage based on the domain specific knowledge on DNN operations.

Layer Fusion. Occamy fuses layers to eliminate inefficient memory reloads between the layers. Occamy investigates consecutive DNN operators, and then substitutes the operators into one fused operator if the pattern is matched. For example, IR 2 in Fig. 3 shows the `DNN.Conv-relu` operator in line 10 that is substituted from `DNN.Conv` and `DNN.relu` in IR 1. Note that some intermediate tensors can be used in other operators, so Occamy applies this optimization when the intermediate tensors are not live-out at the fused operator.

Tensor Coalescer. Occamy coalesces input and output tensors to share their memory spaces in the elementwise operations such as arithmetic operators. The elementwise operators can use the same memory space for input and output because each thread independently accesses each element. For example, Occamy converts the `DNN.add` operator using the same input and output memory space `d_D` in line 17 of IR 2. Here, Occamy applies this optimization when its input tensor is not used later (not live-out at the operator).

C. Global-level Optimizer

Global-level optimization generates a memory-efficient code using a memory pool technique. Occamy analyzes the memory access patterns as a perspective of the entire DNN model, not confined by independent operators. Occamy analyzes the liveness and size of the entire memory object, then decides offsets of each memory object at the memory pool. Since Occamy can analyze end-to-end memory operation in contrast to the existing work using runtime memory adaptor, Occamy can adjust better memory schedule than the runtime memory adaptor.

Memory Access Pattern Analyzer. To efficiently support different access patterns, Occamy adopts multiple memory scheduling algorithms, and finds the optimal memory pool size. This memory access pattern analyzer uses four different memory scheduling algorithms such as first-fit, best-fit, longer-first-fit, and bigger-first-fit. For example,

Algorithm 2: Bigger-first-fit memory allocation algorithm

```
   Input  : liveTable : Tensor set of liveness
   Output : MemLoc : Memory location set of tensors
            PSize : Size of memory pool
1  Function BiggerFirstScheduler (liveTable, MemLoc, PSize) :
2      schedule ← {}
3      desendingOrderSort (liveTable, "size")
4      for var ∈ liveTable do
5          loc ← getStartLoc (schedule, var, "first-fit")
6          region ← {loc, loc+ var.size}
7          schedule.insertRegion (region, var.begin, var.end)
8          MemLoc.insert({var, loc})
9      end
10 end
```

ResNet50 and SSD-ResNet50 show the smallest memory pool size with first-fit and bigger-first-fit, respectively. Occamy performs four scheduling algorithms iteratively and finds the best scheduling policy for the DNN model.

- **First-fit** allocates the target tensor to the first available memory hole with space equal to or larger than the target tensor while traveling memory space from low index to high index.
- **Best-fit** allocates the target tensor to the smallest memory hole that is big enough to assign the target tensor.
- **Longer-first-fit** allocates tensors in order of longer liveness. This policy performs first-fit with a sorted liveness table.
- **Bigger-first fit** allocates tensors in order of memory size of tensors with first-fit policy.

Algorithm 2 shows the bigger-first-fit scheduling algorithm as an example. The scheduler receives the liveness table as an input, and decides the size of the memory pool and a set of offsets in the memory pool for every tensor. The Bigger-first-fit scheduler sorts the liveness table in descending order by the tensor size (line 3). For each tensor variable, the scheduler gets an available start location for every timestamp based on the schedule information that has been decided so far (line 5) and reserves the memory region for each timestamp (line 7). Then, the scheduler collects the memory pool offset of the variable as the output (line 8). Occamy can adjust scheduling policies by changing the sorting algorithm.

Memory Pool Code Generator. Occamy substitutes memory operators into memory pool operators with memory pool offsets generated by the memory access pattern analyzer. Occamy first inserts a memory pool init operator, which create a memory pool with the predetermined memory pool size. Occamy changes memory allocation operators to memory pool offset operators, assigning the variable to a specific location in the memory pool. For example, `DNN.Malloc` operators of IR 2 are substituted into `DNN.Mem-offset` of IR 3. The argument of `DNN.Mem-offset` denotes the base address of the memory pool, offset, and size of each variable. The memory offset operator not only reduces memory usage but also improves inference time because the memory offset operator eliminates the memory allocation, which causes the synchronization overheads of GPUs. In addition, Occamy eliminates deallocation operators and improves inference time.

IV. Evaluation

This section evaluates the inference time and the memory usage of Occamy with six DNN inference models and two GPU hardware. The first hardware environment uses an integrated GPU system, Jeston AGX Xavier with 16GB memory and ARM v8 Rev 2 processor. The second environment uses a discrete GPU system,

DNN Model	# Op	# Var	Eager		Lazy		Occamy	
			#MMG	#Alloc	#MMG	#Alloc	#MMG	#Alloc
ResNet50	176	284	1	284	123	284	1	1
MobileNet	85	142	1	142	58	142	1	1
SSD-ResNet50	208	351	1	351	144	351	1	1
BERT	747	1120	1	1120	648	1120	1	1
YOLOv1	83	134	1	134	58	134	1	1
SRGAN	117	216	1	216	81	216	1	1

TABLE I: Number of memory management groups (MMG) and memory allocations (Alloc) of six benchmarks. MMG means continuous memory allocation operators which are not punctuated by other computation operators.

NVIDIA GeForce RTX 3090 with 24GB memory and Intel® Core™ i7-8700. The evaluation uses six DNN inference models such as ResNet50, MobileNet, SDD-ResNet50, BERT, YOLOv1, and SRGAN. Benchmarks target different applications, so each benchmark has different layer compositions and memory access patterns. ResNet50 and MobileNet are designed for image classification, and especially MobileNet is designed for embedded systems. Both SSD-ResNet50 and YOLOv1 target object detection but use different single-shot detection (SSD) models. BERT performs natural language processing (NLP) with transformers, and SRGAN performs super-resolution generative adversarial network.

The evaluation compares three different compiler implementations to evaluate this work. To implement Occamy, this work builds three implementations on top of the open-source compiler, ONNX-MLIR [11], which supports generating ONNX IR isomorphic to ONNX [26] model. Since ONNX-MLIR does not support the GPU back-end, this work implements the GPU back-end using CUDA Runtime API [27] and tests three implementations.

- **Eager** processes inference using the eagar memory allocation
- **Lazy** processes inference using the lazy memory allocation
- **Occamy** processes inference on Occamy using the memory pool

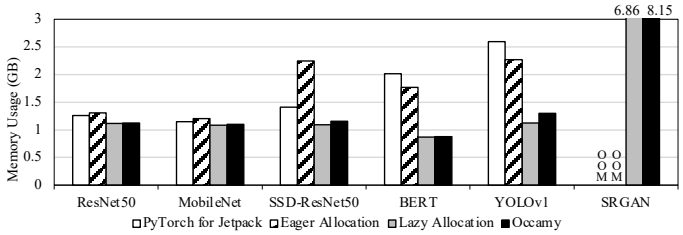
Moreover, this work compares Occamy with the state-of-the-art PyTorch [1] using the same set of benchmarks.

A. Overall Performance

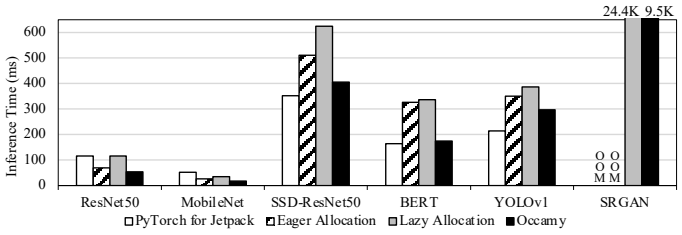
Fig. 4 shows that Occamy reduces memory usage with comparable inference time for most benchmarks in both discrete and integrated GPU systems. Occamy reduces 34.6% memory usage and 1.25× speedups on the geomean average compared to PyTorch.

On the Jetson AGX Xavier, Occamy reduces 31.2% memory usage compared to PyTorch as shown in Fig. 4a while achieving 1.29× speedups. Note that, this evaluation uses PyTorch for Jetson [28] which was released for Jetson embedded system to reduce memory usage by using memory-optimized CUDA, cuDNN, and cuBLAS libraries. Even though PyTorch adjusts specialized optimization, PyTorch shows more memory usage due to the eager memory allocation. Especially for BERT and YOLOv1, Occamy shows 56.3% and 50.1% memory reduction. These two benchmarks have one big layer (embedding layer for BERT and fully-connected layer for YOLOv1) that bounds memory pool size and other layers can reuse that memory pool efficiently. However, Occamy shows 3.7% memory saving for MobileNet. Since MobileNet is a small network, the benefit of the memory pool is concealed by the CUDA context that is used by the CUDA runtime library in both of Occamy and PyTorch. PyTorch cannot execute the SRGAN benchmark due to out-of-memory (OOM), but Occamy successfully generates a memory pool and executes SRGAN on the Jetson AGX Xavier.

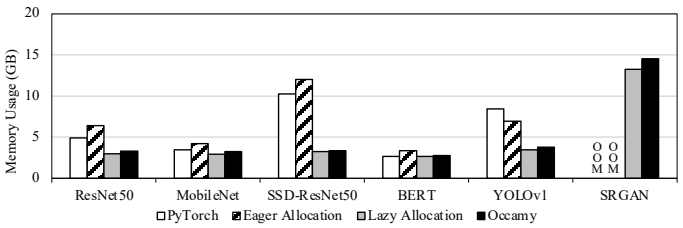
On the RTX 3090, Occamy reduces 37.7% memory usage compared to PyTorch while achieving 1.21× speedups as shown in Fig. 4c. For



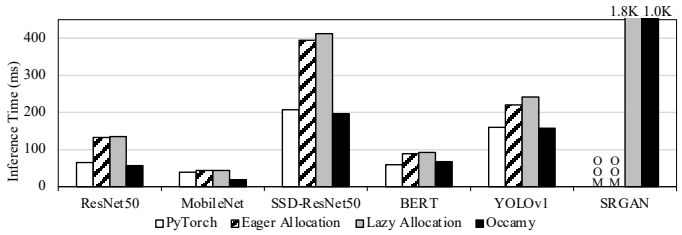
(a) Memory usage of DNN models on Jetson Xavier



(b) DNN model end to end inference time on Jetson Xavier

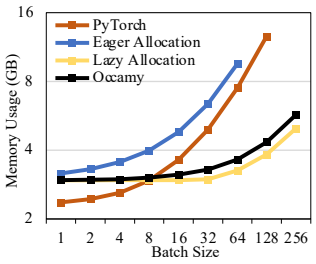


(c) Memory usage of DNN models on RTX3090 GPU

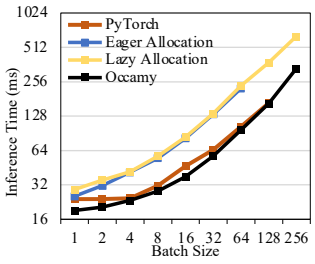


(d) DNN model end to end inference time on RTX3090 GPU

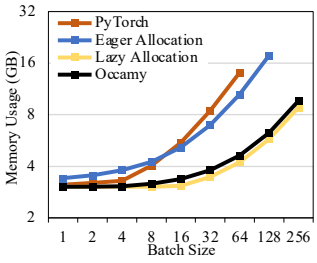
Fig. 4: Inference time and memory usage of the 6 benchmarks. The batch size is 32 for all the models on RTX3090 GPU except for SRGAN which batch size is 2 due to the out of memory error. OOM is noted for the models unable to inference due to the out of memory error.



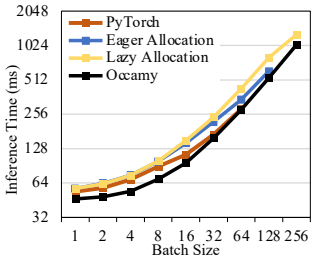
(a) GPU memory usage of ResNet50



(b) Inference time of ResNet50



(c) GPU memory usage of YOLO



(d) Inference time of YOLO

Fig. 5: Inference time and memory usage on RTX3090 GPU. Missing data point at the specific batch size means the out of memory error.

the discrete GPU system, this work evaluates Occamy and other systems with batch size 32, except for the SRGAN model which batch size is 2. Occamy shows similar inference time but reduces memory usage on ResNet50, SSD-ResNet50, and YOLOv1 compared to PyTorch. PyTorch allocates all the tensors at the beginning to speed up inference, but Occamy efficiently reuses a memory pool and reduces allocation overheads. For BERT, Occamy does not show significant memory saving because PyTorch does not support embedding operations on a GPU, so PyTorch runs one big embedding table on the CPU. However, Occamy runs the embedding table on the GPU, and allocates a memory pool of the same size as the embedding table, so the memory pool size is slightly larger than PyTorch's memory usage. If PyTorch supports running the embedding table on a GPU, the PyTorch will use more memory than Occamy.

Fig. 4 shows the efficiency of the memory pool optimization compared to the Eager and the Lazy memory allocation. Occamy uses only 7.3% more memory compared to the Lazy allocation and shows 1.59× speedups compared to the Eager allocation. Table I shows the number of memory management groups (MMG) and the numbers of memory allocation (Alloc) of six benchmarks. The Eager allocation has one MMG because it allocates and deallocates every tensor at the beginning and end of inference, respectively. In contrast, the Lazy allocation causes more MMG because it allocates and deallocates tensors just before their uses and after finishing their liveness. As a result, the Lazy allocation has smaller maximum memory usage than the others but shows slower inference time because of interference of memory operations while running inference. However, Occamy has only one MMG and one Alloc with the similar memory usage to the Lazy allocation; this result implies that the memory pool optimization has the advantage of both allocation schemes. In addition, Occamy shows additional performance improvement compared to the Eager allocations scheme by reducing the number of Alloc that causes synchronization overheads.

B. Case Study: Batch Size Impact

Fig. 5 shows the memory usages and inference times of ResNet50 and YOLO with different batch sizes on RTX3090 GPU. Fig. 5a and Fig. 5c show how memory usage differs with the different batch sizes. In the case of small batch size, in which the chance of memory reuse in the memory pool is small, PyTorch often uses less memory space than Occamy. This is because loading CUDA API functions in an imperative manner in PyTorch uses less memory than loading a whole kernel function generated by Occamy. However, as the batch size increases, the chances for Occamy to reuse the memory space increase, and Occamy can use less memory space than PyTorch or the Eager allocation by using a memory pool.

Fig. 5b and Fig. 5d show the impacts of batch size to the inference time. The inference time of Occamy is slightly lower than PyTorch in the case of the small batch size because Occamy eliminates memory allocations that have negligible effects on performance. As the batch size gets bigger, the memory copy overheads dominate the inference

time, so the inference time of Occamy becomes similar to the inference time of PyTorch.

V. Related Work

CPU-GPU unified memory pool for DNN. Previous work [3, 4, 25] proposes CPU-GPU unified memory pools for DNN training. vDNN [3] virtualizes CPU and GPU memory, seamlessly prefetches tensors to GPU memory and offloads tensors to CPU memory. Superneurons [4] characterizes computation- and memory-intensive layers, and offloads tensors in computation-intensive layers to the CPU memory, so overlaps the CPU-GPU communication overheads with the computation time. Capuchin [25] finds tensor access patterns at runtime, and decides memory prefetch/eviction in the tensor granularity. Compared to the previous work, Occamy analyzes the DNN inference model at static time, so Occamy does not cause runtime overheads to find the memory access patterns. Using the liveness analysis results, Occamy successfully schedules tensor allocation and deallocation timings at compile time.

Memory management schemes that reduce memory allocation overheads. Winter et al. [18] point out that dynamic memory allocation becomes a bottleneck of SIMD/SIMT applications. Gelado and Garland [19] propose a dynamic GPU memory allocation scheme that minimizes lock steps of (de)allocations with two-stage processes, and thus enable concurrent allocations among multiple threads without losing performance. Occamy allocates one big memory pool, and replaces all the memory allocations to memory request operations from the memory pool. Since Occamy eliminates all the (de)allocation operations, Occamy successfully reduces system calls and synchronization overheads caused by the (de)allocations.

Memory management schemes that reuse preallocated memory spaces. [1, 8, 29, 30, 31] reuse memory spaces allocated in the previous layers, and reduce the inference and training times and the memory usage. PyTorch [1] allocates all the tensors at the beginning of the program, and reduces the (de)allocation costs during the execution. TensorFlow Lite Micro [8] implements a memory pool, but their target devices are limited to embedded CPUs. Ji et al. [31] reuses memory spaces for tensors in convolution layers. PyTorch [1], TensorFlow XLA [29], MXNet [30] implements in-place operations that reuse the same memory space for input and output tensors, and reduces memory usages for inference and training. Like the schemes, Occamy coalesces input and output tensors of elementwise operators to share their memory space, and reuses memory spaces in the memory pool that is allocated at the beginning.

VI. Conclusion

This work proposes a new DNN inference compiler, Occamy, which reduces the memory usage and management overheads of a DNN model. Occamy analyzes the liveness of input and output tensors for DNN operations. Then, Occamy reduces redundant tensors by fusing layers and coalescing input and output tensors. Finally, Occamy analyzes the memory access patterns throughout the whole DNN model based on the liveness analysis result, finds the best memory scheduling algorithm, and generates the memory pool. Moreover, Occamy schedules when and where to place each tensor in the memory pool, thus reducing the memory management overheads at runtime. Occamy reduces the memory usage by 34.5% and achieves a geomean speedup of 1.25× compared to PyTorch on Jeston AGX Xavier for six DNN inference models.

Acknowledgment

We thank the CoreLab members for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by IITP-2020-0-01847, IITP-2020-0-01361, IITP-2021-0-00853 and IITP-2022-11-2032 funded by the Ministry of Science and ICT. This work is also supported by Samsung Electronics. *(Corresponding author: Hanjun Kim)*

References

[1] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". *NeurIPS*. 2019.
[2] NVIDIA. *Jetson AGX Xavier Module*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.
[3] Minsoo Rhu et al. "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design". *MICRO*. 2016.
[4] Linnan Wang et al. "Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks". *PPoPP*. 2018.
[5] Donglin Yang and Dazhao Cheng. "Efficient GPU Memory Management for Nonlinear DNNs". *HPDC*. 2020.
[6] Xiaonan Nie et al. "TSplit: Fine-grained GPU Memory Management for Efficient DNN Training via Tensor Splitting". *ICDE*. 2022.
[7] Kartikeya Bhardwaj et al. "Memory- and Communication-Aware Model Compression for Distributed Deep Learning Inference on IoT". *ACM Trans. Embed. Comput. Syst.* (2019).
[8] Robert David et al. "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems". *MLSys*. 2021.
[9] Yuan Wen et al. "TASO: Time and Space Optimization for Memory-Constrained DNN Inference". *SBAC-PAD*. 2020.
[10] Chris Lattner et al. "MLIR: Scaling compiler infrastructure for domain specific computation". *CGO*. 2021.
[11] Tian Jin et al. "Compiling ONNX neural network models using MLIR". *arXiv preprint* (2020).
[12] Kaiming He et al. "Deep Residual Learning for Image Recognition". *CVPR*. 2016.
[13] Andrew G Howard et al. "MobileNets: Efficient convolutional neural networks for mobile vision applications". *arXiv preprint* (2017).
[14] Wei Liu et al. "SSD: Single Shot MultiBox Detector". *ECCV*. 2016.
[15] Jacob Devlin et al. "BERT: Pre-training of deep bidirectional transformers for language understanding". *arXiv preprint* (2018).
[16] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". *CVPR*. 2016.
[17] Christian Ledig et al. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". *CVPR*. 2017.
[18] Martin Winter et al. "Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks". *PPoPP*. 2021.
[19] Isaac Gelado and Michael Garland. "Throughput-Oriented GPU Memory Allocation". *PPoPP*. 2019.
[20] Seunghee Shin et al. "Scheduling Page Table Walks for Irregular GPU Applications". *ISCA*. 2018.
[21] Isaac Gelado et al. "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems". *ASPLOS*. 2010.
[22] Jaehoon Jung et al. "Overlapping Host-to-Device Copy and Computation Using Hidden Unified Memory". *PPoPP*. 2020.
[23] Ignacio Sañudo Olmedo et al. "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective". *RTAS*. 2020.
[24] Ammar Ahmad Awan et al. "OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training". *HiPC*. 2018.
[25] Xuan Peng et al. "Capuchin: Tensor-Based GPU Memory Management for Deep Learning". *ASPLOS*. 2020.
[26] Junjie Bai et al. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx.
[27] NVIDIA. *CUDA Runtime API Documentation*. https://docs.nvidia.com/cuda/cuda-runtime-api/.
[28] NVIDIA. *Installing PyTorch for Jetson Platform*. https://docs.nvidia.com/deeplearning/frameworks/install-pytorch-jetson-platform/index.html.
[29] Alex Suhan et al. "LazyTensor: combining eager execution with domain-specific compilers". *arXiv preprint* (2021).
[30] Tianqi Chen et al. "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems". *arXiv preprint* (2015).
[31] Cheng Ji et al. "Memory-efficient deep learning inference with incremental weight loading and data layout reorganization on edge systems". *Journal of Systems Architecture* (2021).