IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 December 2023, accepted 3 January 2024, date of publication 11 January 2024, date of current version 22 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3352900

RESEARCH ARTICLE

Off-Chip Memory Allocation for Neural Processing Units

ANDREY KVOCHKO, EVGENII MALTSEV, ARTEM BALYSHEV, STANISLAV MALAKHOV, AND ALEXANDER EFIMOV
Advanced System Software Laboratory, Samsung Research, 127018 Moscow, Russia
Corresponding author: Evgenii Maltsev (e.maltsev@samsung.com)

ABSTRACT Many modern Systems-on-Chip (SoCs) are equipped with specialized Machine Learning (ML) accelerators that use both on-chip and off-chip memory to execute neural networks. While on-chip memory usually has a hard limit, off-chip memory is often considered large enough to hold the network's inputs, outputs, weights, and any intermediate results that may occur during model execution. This assumption may not hold for edge devices, such as smartphones, which usually have a limit on the amount of memory a process can use. In this study, we propose a novel approach for minimizing a neural network's off-chip memory usage by introducing a tile-aware allocator capable of reusing memory occupied by parts of a tensor before the entire tensor expires. We describe the necessary conditions for such an off-chip memory allocation approach and provide the results, showing that it can save up to 33% of the peak off-chip memory usage in some common network architectures.

INDEX TERMS NPU, memory allocation, neural network runtime, tiling, strip-packing problem.

I. INTRODUCTION

In recent years, significant advances have been made in Deep Learning in several areas. DL models have achieved great accuracy in many computer vision tasks, including image classification, semantic segmentation, super-resolution, object recognition, and others [1], [2], [3], [4], [5], as well as in other domains, such as Natural Language Processing (NLP), Speech Recognition [6], [7], [8] and natural language generation (NLG) [9], [10], [11]. The improved accuracy of these models comes at the cost of an increased number of parameters and size of the feature maps. Therefore, reducing the amount of memory used to execute a model has become increasingly important.

The commercialization of these DL models prompts many companies to develop specialized AI hardware, whose main purpose is to reduce inference latency or decrease energy consumption. These accelerators are often referred to as Neural Processing Units (NPUs). They can be installed on edge devices (such as mobile devices, embedded solutions, wearables, or IoT devices with microcontrollers) to locally execute the DL model. This edge-computing solution preserves data privacy and provides real-time processing [12]. The benefits of a dedicated AI accelerator include reduction of load on the CPU, GPU, and main memory, ensuring stable performance, and speeding up inference.

Effective memory management is crucial for edge devices because of their resource-constrained environments. These devices often have weaker CPUs and are equipped with flash memory more than RAM, making them incapable of efficiently processing neural networks. Addressing the task of reducing memory consumption can enhance the device's smart functionality by enabling the execution of more complex neural networks or NN ensembles on the device. Moreover, it can decrease the size and cost of the target product by reducing the memory requirements. Even high-end mobile devices have system-level mechanisms to manage memory usage, which can lead to the termination of apps that exceed memory limits, thereby affecting the complexity and performance of deployable NN models.

Reduction of the main memory load is possible because most accelerators are equipped with fast on-chip memory, which is exclusively used for computations, while the main memory only stores inputs and outputs of neural network

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

layers. Unfortunately, in many cases, on-chip memory is not large enough to fit an entire feature map, meaning that only a part of the feature map can be loaded and processed at a time, while the bulk of the data remains in the external or off-chip memory.

To achieve this, feature maps must be divided into smaller pieces, often referred to as slices, views, or tiles. Tiles are loaded from external into the on-chip memory, processed by the NPU, and the result is stored back into the external memory to free the on-chip memory for the next tile. To avoid overwriting the input feature map data in external memory, the existing DL compilers allocate a separate external memory region for the output feature map and retain the entire input in memory until it is fully processed by the NPU.

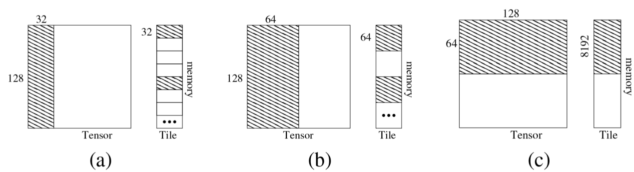


FIGURE 1. Memory layout of tiles of a 128 × 128 tensor in row-major format: a) tile size is 128 × 32 b) tile size is 128 × 64 c) tile size is 64 × 128.

This design decision is made in part because the tile of a tensor, in general, can not be presumed to occupy a contiguous memory region because it is a slice of a multidimensional tensor stored in a linear memory space. Consider a tensor with 128×128 elements allocated in the off-chip memory in row-major order. Fig. 1 shows the memory regions occupied by tiles of sizes 128×32, 128×64 and 64×128. Because of this non-contiguous nature of tensor tiles, freeing memory occupied by a tile earlier than the entire feature map is processed would result in a highly fragmented memory space, and it is presumed that it would yield little opportunity for memory consumption optimization.

The tasks of finding the optimal tiling strategy and allocating memory for tensors are performed by Deep Learning compilers such as Glow [13], TVM [14], ngraph [15], DBLP [16] or DLVM [17]. This paper proposes compiler optimization for NPUs with the main goal of reducing the required amount of external memory occupied by large intermediate feature maps. This is achieved by reusing the memory occupied by a tile of the input tensor as soon as the tile is processed, and there are no more usages of its data. In many cases, memory space is fragmented after freeing a single tile. Our method benefits from the fact that this fragmentation can demonstrate patterns in which different tiles can often occupy free non-contiguous memory regions without any overlap with the remaining tiles of the original tensor.

We propose a method of allocating tensors in external memory that considers both the difference in the lifetimes of a tensor and its tiles and the non-continuity of the memory occupied by tiles. We demonstrate that our memory allocation algorithm can be effectively applied to reduce the amount of external memory used by a neural network model.

In Section II, we describe the problem in detail and present several previously proposed solutions. We present our Tile-Aware Allocator (TAA) design in Section III, and in Section IV we show that this approach can significantly reduce peak off-chip memory usage in many popular NN models, and compare external memory demand with the Shared Objects algorithm described in [18].

II. PROBLEM STATEMENT AND RELATED WORK

Allocating tensor data is a common task for many deep neural network (DNN) compilers. There are two main approaches to memory allocation: dynamic and static approaches. Dynamic allocation involves a runtime environment that allocates memory during a program execution. In contrast, static memory allocation reserves memory regions for tensors and data structures at the compile time before the program runs. Deep Learning compilers typically implement a memory manager that conducts a memory planning pass for static allocation to a pre-allocated memory buffer holding intermediate tensors and applies optimizations. This paper discusses an algorithm for memory management in a Deep Learning compiler that facilitates static memory allocation.

The task of allocating tensor data resembles a strip-packing problem. This problem involves a memory region with a specified width and infinite height along with a set of items (tensors) characterized by their size and lifetime. In this context, the width of the memory region corresponds to time, whereas its height represents a memory offset. The lifetime of a tensor is typically determined by the interval between its first and last use in a graph walk. The problem is to find the lower-left corner of each item such that no overlap between items occurs and the height of the packing is minimal.

A. PRIOR WORK

The strip-packing problem is known to be NP-hard, therefore, researchers have attempted to find heuristics to approximate the optimal solution [19], [20], [21], [22].

In general, DNN Compilers split the strip-packing problem into memory allocators and schedulers. Memory allocators focus on finding item offsets that produce optimal packing, and keep the tensor lifetime fixed. Schedulers vary only tensor lifetime. Both can be modified to produce more optimal packing. Pisarchyk and Lee [18] proposed a memory-optimizing allocator based on a linear scan algorithm, Barenboim et al. [23] used graph coloring, and Ahn et al. [16] implemented a memory-aware scheduler.

Other methods for memory usage reduction exist. General-purpose deep learning frameworks such as Caffe [24], PyTorch [25], and MXNet [26] incorporate several static memory reduction techniques. Common optimization methods, such as operation fusion, in-place operations, and memory sharing, can be adapted for deep learning compilers that target edge devices. For instance, operation fusion combines activation functions with the preceding operation

(e.g., convolution), optimizing both the performance and memory transfer between the host device and the accelerator. In-place operations store output values directly in the memory assigned to an input value. Memory sharing optimization repurposes the memory of intermediate results that are no longer required for further computation. Similar optimizations are discussed in [27], whereas [28] delves into memory optimization techniques aimed at reducing memory consumption and enhancing computational effectiveness by minimizing additional memory transformations for memory-bound operations. Operations such as depthwise convolution can introduce additional performance overhead.

In [29], the authors optimized the on-chip memory using a Scheduling Method with operation fusion and memory reuse. Additionally, frameworks such as the Glow Compiler [13] implement *buffer sharing* optimizations that attempt to reuse an instruction input‘s buffer for its output. Minakova and Stefanov [30] and Artemev and Roeder [31] employed the Cyclo-Static Dataflow model and MapReduce techniques, leveraging CNN properties to process model input in parts, thus reducing the size of intermediate tensors. General-purpose deep learning frameworks, such as those in [25] and [32], are designed for both training and inference of neural networks. In this study, we focus on the inference-only work mode. Consequently, certain memory optimizations are inapplicable, such as in-place operations [27] which are utilized during the training mode and require data from the forward pass to be retained for the backward pass.

B. CHALLENGES

Freeing memory occupied by a tile introduces memory fragmentation because tiles occupy non-contiguous regions of memory. To place a different tile into this non-contiguous memory space, we need to be sure that the new tile does not intersect any other live memory region between the free memory fragments.

Let us consider tile t that spans $s_i(t)$ elements in i-th dimension of tensor T, starting with element $o_i(t)$, $i \in [0, rank(T))$. The memory distance between two consecutive elements of t in dimension i is denoted as $stride_i(t)$. The memory address of a tile element at index $\mathbf{x}$ is given by

$$\begin{aligned} addr(\mathbf{x}, t, T) &= addr(T) + \sum_{i=0}^{rank(T)} (x_i + o_i(t)) \cdot stride_i(t) \\ &= addr(T) + \mathit{off}\,(t, T) + \sum_{i=0}^{rank(T)} x_i \cdot stride_i(t), \\ & 0 \le x_i < s_i(t) \text{ for } i \in [0, rank(T)), \text{ and} \\ \mathit{off}\,(t, T) &= \sum_{i=0}^{rank(T)} o_i(t) \cdot stride_i(t) \end{aligned} \tag{1}$$

The two tiles t and $\hat{t}$ of tensors T and $\hat{T}$ overlap if they share the same memory address for some elements $\mathbf{x}$ and $\hat{\mathbf{x}}$ respectively:

$$addr(\mathbf{x}, t, T) = addr(\hat{\mathbf{x}}, \hat{t}, \hat{T})$$

Alternatively, they overlap if the following equation has a solution for some $\mathbf{x}$ and $\hat{\mathbf{x}}$

$$\begin{aligned} &\sum_{i=0}^{rank(T)} x_i \cdot stride_i(t) - \sum_{j=0}^{rank(\hat{T})} \hat{x}_j \cdot stride_j(\hat{t}) \\ &\quad = addr(\hat{T}) + \mathit{off}\,(\hat{t}, \hat{T}) - addr(T) - \mathit{off}\,(t, T) \end{aligned} \tag{2}$$

where $0 \le x_i < s_i(t)$ for $i \in [0, rank(T))$ and $0 \le \hat{x}_j < s_j(\hat{t})$ for $j \in [0, rank(\hat{T}))$.

This linear Diophantine equation with bounded variables can be solved in polynomial time [33], [34]. In a naive approach, this equation needs to be solved $n \cdot m$ times to determine whether a single tile of a tensor conflicts with any other tile, where n is the number of tensors in a model, and m is the number of tiles in a tensor. Considering that there are m tiles in a tensor, to determine if a tensor can be placed at a given offset, this equation must be solved $n \cdot m^2$ times. To find a suitable offset, additional S attempts are required, where S is proportional to the size of the on-chip memory in bytes. Overall, the complexity of this approach is $O(S \cdot n^2 \cdot m^2)$ for allocating all the tensors.

This naive approach is prohibitively expensive in terms of computational cost. Therefore, this study focuses on two main questions.

1) How to efficiently check if two tiles of a tensor intersect?
2) How to adjust the desired memory location for the current tensor when a conflict exists?

C. DEFINITIONS

- First, a tile is defined as a fragment of a tensor. Each tensor is associated with a set of tiles.
- We define tile lifetime as the time interval between the first and last use of the tile.
- The tensor lifetime is the interval between the first and last uses of a tensor as a *whole*. For large tensors that cannot fit entirely into the limited on-chip memory, the tensor lifetime spans the time from when it is constructed from the output tiles of the previous layer to when it is first divided into input tiles for the next layer. Often, this means that the tensor lifetime is effectively reduced to zero: no operation can use the entire tensor without dividing it into smaller parts.
- We define a "peer" as an item (a tile or tensor) whose lifetime intersects with the lifetime of a given item.
- Next, we place constraints on the memory addresses where a tensor can be allocated. The tiles offsets relative to the start of the tensor are fixed. The memory address of the tensor is considered invalid if any of the tiles overlap with another allocated item or its tile.

III. MEMORY ALLOCATION ALGORITHM

In this study, we propose a tile-aware memory allocator (TAA) that utilizes the difference between the lifetimes of the entire tensor and its individual tiles to reduce the off-chip memory footprint. Our approach efficiently handles a large

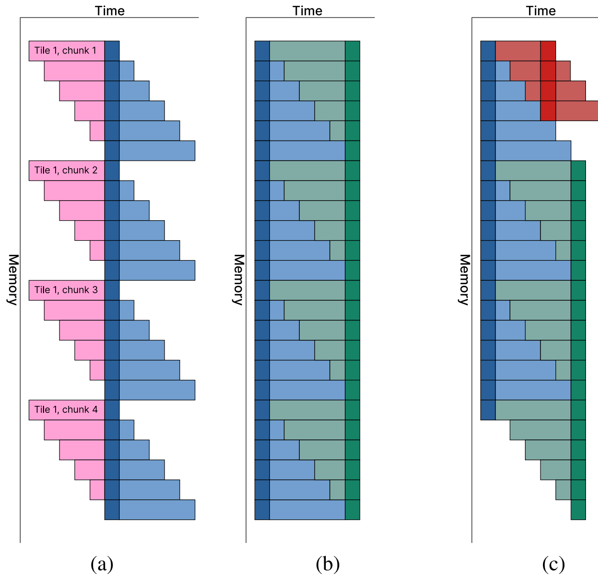


FIGURE 2. Example of the tensors allocation in time and memory. a) Allocation of one tensor; b) Fitting the output of an arbitrary neural network layer into the unoccupied space; c) Resolving a collision.

number of tiles resulting from processing tensors on a chip with limited on-board memory. The main idea is to reuse the memory occupied by a tile immediately after its last use, but before the last use of the entire tensor.

The main contributions of this paper are as follows.

- We define the conditions under which two tensors with intersecting lifetimes can share memory and reduce the off-chip memory consumption.
- These conditions are used in the proposed memory allocation algorithm, which solves a modified version of the strip-packing problem. The algorithm considers both the tensor and tile lifetimes.
- Testing the implemented algorithm with widely-used neural network models shows that our memory allocation approach reduces external memory usage by up to 33%. The results are presented in IV.

An example of tensor allocation, in terms of time and memory, is shown in Fig. 2. Let us assume that we have an intermediate tensor, as shown in Fig. (2a). The tensor is constructed from the six output tiles of the previous neural network layer (indicated by the pink-colored zones) and is divided into six tiles for the next layer (shown in light blue). Each tile consists of four chunks. The peak memory consumption for the tensor is marked by a dark blue zone and has a shorter lifetime than its individual tiles.

Owing the variations in tile lifetimes and their processing order, we can attempt to fit the output of an arbitrary neural network layer into the unoccupied space within the same time interval. An example of such a space utilization by an arbitrary tensor is shown in Fig. 2b. The initial tensor is indicated by a dark blue color, whereas the output tensor is marked in dark green.

The main task of the memory allocator is to find a suitable location in the memory for the next tensor, given the currently allocated items, such that none of its tiles intersect in both

Algorithm 1 TensorAllocate

1 overlap ← -1
2 addr ← start of first memory region of sufficient size
3 **while** *overlap* $\neq 0$ **do**
4   overlap ← 0
5   **foreach** *tile of* T **do**
6   **foreach** *allocated peer of* T **do**
7   **if** *lifetimes of tile and peer do not overlap* **then**
8   **continue**
9   **end**
10   overlap ← `CollisionSize` (tile, addr, peer, address of peer's tensor)
11   **if** *overlap* $\neq 0$ **then**
12   **break**
13   **end**
14   **end**
15   **if** *overlap* $\neq 0$ **then**
16   **break**
17   **end**
18   **end**
19   addr ← first memory region of sufficient size, starting with addr + overlap
20 **end**

time and memory with another item. By iteratively applying this process to each tensor, a memory allocation scheme for the entire model is found. While the starting memory address for a tensor can be selected based on these constraints alone, the starting address for a tile is fixed at a specific offset relative to the beginning of the tensor.

Our strategy is to first find a suitable place for tensor T itself and then check that neither of its tiles conflicts with the already allocated items. If they do, we adjust the starting address of T according to the size of the overlapping region of the two intersecting chunks.

In case of a conflict, such as when the tensor being allocated intersects in time and space with another tensor (e.g., indicated by the red zone in Fig. 2c), which has been allocated in the available memory space between the live tiles, we shift the tensor address by the size of the overlap region. This process will eventually yield the configuration presented in Fig. 2c, where the green tensor is shifted relative to the zero-memory offset.

The conditions for reusing occupied tile memory in the allocation algorithm are as follows.

- The tensors are divided into tiles and processed in parts. The order of tile processing is fixed.
- The lifetimes of tensors can intersect.
- The lifetimes of individual tiles should not intersect.

A. TENSORS ALLOCATION

First part of allocation alorithm starts with tensor sorting. All tensors are sorted according to one of the three heuristics.

- First allocate items that take up the most amount of memory.
- Allocate items with longest lifetime first.
- Sort items by decreasing number of peers.

For each tensor T in the sorted list, we use a linear scan among the allocated tensor peers to find the first unoccupied region of memory of sufficient size to fit T.

Tensor peers are identified as tiles of other allocated tensors whose lifetimes overlap with those of any tile of the given tensor.

The task of finding peers is essentially the task of, given an interval I, finding intervals overlapping with I. The solution to this problem is trivial using an augmented Binary Search Tree, with minimal start point and maximal end point of a subtree recorded at every tree node. The insertion and query complexity of this tree is $O(log(n))$, and the space complexity is O(n).

Algorithm 1 solves the problem of determining a suitable starting address for a tensor. We employ a linear scan starting at a given memory offset to find a candidate location for a tensor and then check if any of its tiles intersect with any allocated peer items using Algorithm 2.

B. FINDING TILE INTERSECTIONS

Once a suitable memory region has been found, we check whether no tiles of T intersect with any other allocated peer tile. We note that while existing SMT solvers, such as Z3 [35], can solve problems such as Equation 2, our task is less general than the problems these solvers were designed for.

- The first simplification is that by designing, the number of contiguous regions in a tile is small. Selecting a tiling strategy that minimizes tile fragmentation is not only beneficial for the TAA algorithm but also reduces memory transfer delays owing to fewer transfer requests, as shown by Sousa et al. [36].
- The second simplification is that the boundary conditions on the variables in Equation 2 are determined by tensor sizes, which always have a lower limit of zero and a relatively small upper limit determined by the size of a model that can fit into the device memory.

Given these limitations, using a general-purpose SMT solver is not the most efficient solution because of the large number of tiles that can occur in a neural network and the prohibitively large number of equations to solve for a full allocation pass. Therefore, we employed a different approach.

In the preparation step, for each tile, we find all contiguous chunks of memory using Algorithm 3. This algorithm returns the offsets and sizes of each contiguous memory chunk of a tile. The complexity of this algorithm is $O(n)$, where n denotes the number of chunks.

Given the offsets and sizes of each chunk of every tile, Algorithm 2 checks whether two tiles of a tensor intersect and returns the size of the first overlapping region found. It relies on the fact that Algorithm 3 returns the chunks in the order of increasing offsets and uses a linear scan across the chunks of the two tiles to find the first intersection. Note that calls to the `RenderChunks` algorithm simply retrieve a memoized value in the preparation step. The complexity of Algorithm 2 is linear in terms of the number of chunks.

Algorithm 2 CollisionSize

```
Input: t1, t2, t1base, t2base /* t1 and t2 are
    tiles to check; t1base and t2base
    are addresses of the tiles'
    tensors                              */
Result: Size of the overlapping region
1  if t1 lies completely to the left of t2 then return 0
2  if t1 lies completely to the right of t2 then return 0
3  t1idx ← 0
4  t2idx ← 0
5  o1, s1 ← RenderChunks(shape(t1), strides(t1), 0,
     0)
6  o2, s2 ← RenderChunks(shape(t2), strides(t2), 0,
     0)
7  while t1idx < len(s1) and t2idx < len(s2) do
       // for definition of off(), refer
          to Equation 1
8      t1start ← off(t1) + t1base + o1[t1idx]
9      t1end ← t1start + s1[t1idx]
10     t2start ← off(t2) + t2base + o2[t2idx]
11     t2end ← t2start + s2[t2idx]
12     if t1end ≤ t2start then
13         t1idx ← t1idx + 1
14         continue
15     end
16     if t2end ≤ t1start then
17         t2idx ← t2idx + 1
18         continue
19     end
20     return t1end - t2start
21 end
22 return 0
```

C. STEP-BY-STEP EXAMPLES

1) ALGORITHM RENDERCHUNKS

In this section, we provide a step-by-step example of the `RenderChunks` algorithm for rendering contiguous chunks of a $4 \times 64 \times 128$ tile of a $4 \times 128 \times 128$ tensor in CHW format. In this example, the tile consists of four contiguous chunks of equal size with each chunk representing the top half of the tensor.

We start with `dsizes`=[4, 64, 128] and `dstrides`= [16384, 128, 1].

The initial `offset` and `axis` are set to 0, and the element size is 1 byte.

The check for contiguity in line 3 fails, because `axis` 0 is not contiguous. The criterion for contiguity can be defined as the product of all the dimension sizes starting at the current axis+1 and ending at the rank of the tensor equal to the stride of the current dimension.

Algorithm 3 RenderChunks

```
Input: dsizes[N], dstrides[N], offset, axis, esize
    // dsizes and dstrides are sizes
    and strides of each tile
    dimension; esize is tensor
    element size in bytes
Result: Chunk offsets offsets[M], chunk sizes
        sizes[M]
1  offsets ← ∅
2  sizes ← ∅
3  if dimension axis and further are contiguous then
4  |  offsets ← offsets ∪ {offset}
5  |  sizes ← sizes ∪ {dsize[axis] · dstrides[axis]}
6  |  return offsets, sizes
7  end
8  forall i ∈ [0, dsizes[axis]] do
9  |  if axis=len(dsizes) - 1 then
10 |  |  offsets ← offsets ∪ {offset}
11 |  |  sizes ← sizes ∪ {esize}
12 |  else
13 |  |  noffsets, nsizes ← RenderChunks (dsizes,
   |  |    dstrides, offset, axis + 1)
14 |  |  if dimension axis + 1 is contiguous and
   |  |    noffsets[0] = offsets[−1] + sizes[−1] then
15 |  |  |  sizes[−1] ← sizes[−1] + nsizes[0]
16 |  |  else
17 |  |  |  sizes ← sizes ∪ nsizes
18 |  |  |  offsets ← offsets ∪ noffsets
19 |  |  end
20 |  end
21 |  offset ← offset + dstrides[axis]
22 end
23 return offsets, sizes
```

For `axis` 0, this product is equal to 64 * 128 = 8192, which is different from the stride of this dimension (16384).

Entering the loop in line 8, `axis` 0 is not the last axis, so we move on to line 13, where we perform a recursive call to `RenderChunks` with the `axis` increased to 1. In this case, `axis` 1 is contiguous (the size of the next dimension is equal to 128, as is the stride of dimension 1), so `RenderChunks` returns [0] for `offsets` and [8192] for `sizes`.

The following condition in line 14 checks whether the next dimension is contiguous and the returned memory chunk starts immediately after the last rendered chunk ends; if so, merges the two chunks. Otherwise, the returned chunk is appended to the result.

In line 21, we set the `offset` to 16384 and continue through loop three more times, ultimately returning the `sizes` and `offsets` of the contiguous memory chunks: [8192, 8192, 8192, 8192] and [0, 16384, 32768, 49152].

2) ALGORITHM COLLISIONSIZE

In this section, the execution of the `CollisionSize` algorithm is described. Consider a tensor of size $4 \times 128 \times 128$ allocated at memory offset 0 and its tile `t1` of size $4 \times 64 \times 128$ at offset 0 in each dimension. Consider another tensor of the same size ($4 \times 128 \times 128$) allocated at memory offset 128, and its tile `t2` of size $4 \times 64 \times 128$ at offset 0. The `CollisionSize` algorithm returns the size of the overlapping memory region for the two tiles.

We start with tiles `t1` and `t2`, and set `t1base` to 0 and `t2base` to 128. The checks in lines 1 and 2 fail because `t1` lies neither completely to the left nor to the right of `t2`. This can be verified by introducing the notion of tile span, which is the distance between the first and last elements of the tile. The span of `t1` and `t2` is 57344 bytes. We can see that

$$t2base + offset(t2) < t1base + offset(t1) + span(t1),$$

implying that `t1` does not lie completely to the left of `t2`, and

$$t1base + offset(t1) < t2base + offset(t2) + span(t2),$$

implying that `t1` does not lie completely to the right of `t2`.

Next, we initialize the indices for iterating through chunks of `t1` and `t2` (`t1idx` and `t2idx`) to 0. We then use the `RenderChunks` function to calculate the offsets and sizes of the chunks for both tiles:

For `t1`, `RenderChunks` with the shape (4, 64, 128) and strides (16384, 128, 1) results in offsets [0, 16384, 32768, 49152] and sizes [8192, 8192, 8192, 8192] bytes for each of the four chunks. For `t2`, the calculation is similar. Using these offsets and sizes, we enter the main loop of the `CollisionSize` algorithm. The loop iterates over the chunks of `t1` and `t2` to check for overlap.

We calculate the start and the end of the current chunk for `t1` (`t1start` and `t1end`): `t1start` $= 0 + 0 + 0 = 0$, `t1end` $= 0 + 8192 = 8192$. For `t2`: `t2start` $= 0 + 128 + 0 = 128$, `t2end` $= 128 + 8192 = 8320$. Then, we check if `t1end` (8192) is less than or equal to `t2start` (128), or `t2end` (8320) is less than or equal to `t1start` (0). Neither condition is true, therefore, we have an overlap. The size of the overlapping region is `t1end` - `t2start` = 8192 - 128 = 8064 bytes.

3) ALGORITHM TENSORALLOCATE

In this section, we provide a line-by-line execution example of the `TensorAllocate` algorithm. For the sake of example, let us consider a neural network consisting of one or several unary element-wise layers. The output size of such a network is always equal to its input size. Let us assume that input tensor `I` of size 4×128x128 is processed in four tiles of size 4×64x128, and is allocated at a memory offset of 0. The lifetime of tile `i1` is [0, 1], that of tile `i2` is [0, 2], `i3` is [0, 3], and that of `i4` is [0, 4]. The lifetime of tensor `I` is set to [0, 0].

Tile `i1` is loaded onto the DLA and processed, and the result is returned to the off-chip memory. This result is the first tile of the output tensor `O`, `o1`, for which we want to find an offset and its lifetime is [2, 5]. We used the `TensorAllocate` algorithm to find the offset of tensor `O`.

The lifetime of tensor `O` is [5, 5], which does not overlap with the lifetime of `I`. This implies that we can choose offset 0 as the first suitable address for tensor `O`. Therefore, we set `addr` to 0. The algorithm enters the while loop in line 3. After setting the `overlap` to zero in line 4, we enter the loop over all tiles of `O`, starting with `o1`, and the loop over all tiles of `I`, starting with `i1` immediately after. The lifetime of `o1` [2, 5] does not overlap with the lifetime of `i1` [0, 1], therefore we continue to `i2`. Tile `i2` has an offset of 16384, which means that it lies completely to the right of `o1`, so even though the lifetimes of `o1` and `i2` overlap, the `CollisionSize` algorithm returns 0 for tile `i2`. The same argument applies to tiles `i3` and `i4`. The process is repeated for all tiles of `O`, confirming that there is no overlap between tensors `I` and `O`, and returns an offset of 0 for tensor `O`.

TABLE 1. Peak external memory usage by intermediate tensors.

Model	Peak memory usage (bytes)		
	TAA	Shared Objects	Change
Inception v3	901600	901600	0.00%
Inception v4	**1894560**	2235616	-15.26%
MobileNet v1	**278272**	301056	-7.57%
MobileNet v2	100352	100352	0.00%
ResnetV2 101	**2791232**	3240000	-13.85%
SqueezeNet	**410528**	484182	-15.2%
MNASNet 1.3	200704	200704	0.00%
FSRCNN	**16777216**	25165824	-33.33%

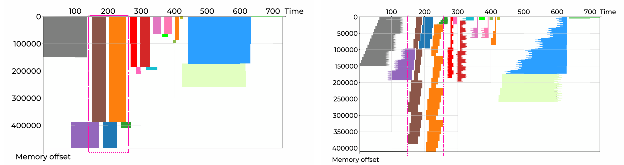
TABLE 2. Phase duration of the intermediate tensors allocation.

Model	Allocation time (μs)	
	TAA	Shared Objects
Inception v3	79852	50566
Inception v4	632723	267550
MobileNet v1	24974	23724
MobileNet v2	12936	12511
ResnetV2 101	914560	320050
SqueezeNet	32015	6274
MNASNet 1.3	50980	48221
FSRCNN	399186	173413

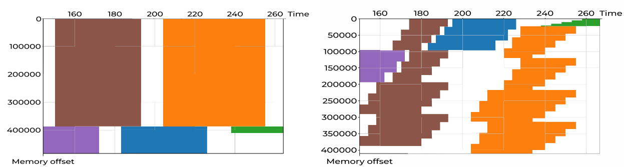
IV. EXPERIMENTAL RESULTS

In this section, we present a comparison between the TAA approach and the algorithms described in [18]. In [18], the authors used shared memory buffers and attempted to allocate sorted tensors to these shared buffers to minimize the peak of memory consumption. Three main strategies were used in this Shared Object algorithm: Greedy by Breadth, Greedy by Size, Greedy by Size Improved. The Greedy by Breadth strategy takes into account the total size of all peers for each tensor. The Greedy by Size strategy sorts tensors based on their memory size. The Greedy by Size Improved strategy sorts tensors according to their memory sizes by splitting them into levels. For each model, we used all three Shared Objects approaches from [18] (Greedy by Breadth, Greedy by Size, Greedy by Size Improved) and selected the one that produces the packing with the least peak off-chip memory. To evaluate the TAA allocator, we used five CNN image classification models: Inception V3 [1], Inception V4 [37], MobileNet V1 [5], MobileNetV2 [4], ResNet V2 101 [38], SqueezeNet [39], MNASNet 1.3 [40] and FSRCNN [3]. In Table 1 we reported the peak memory usage by the intermediate tensors, as well as the total time (Table 2) taken by Algorithm 1. All models were quantized to eight bits using the open-source neural network compiler ONE [41]. For model compilation, we used a single workstation with an AMD Ryzen Threadripper 2950X 16-Core processor. The results are presented in Tables 1 and 2.

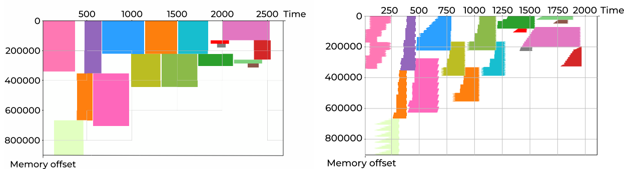
In the remainder of this section, we analyzed the packing produced by TAA for some of the models, identified peak memory usage points, and showed in detail how TAA is able to reduce the peak memory usage.



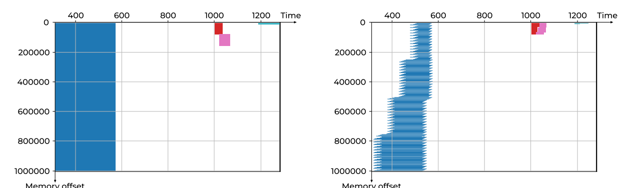
(a) Model packing produced by the baseline Shared Objects algorithm (left) and the TAA algorithm (right) for SqueezeNet



(b) Shared Objects algorithm peak memory graph (left) and TAA peak memory graph (right) for SqueezeNet



(c) Model packing produced by the baseline Shared Objects algorithm (left) and the TAA algorithm (right) for Inception v3



(d) Model packing produced by the baseline Shared Objects algorithm (left) and the TAA algorithm (right) for MobileNet v2

FIGURE 3. Model baseline packing and TAA peak memory graph for different models.

In Fig. 3a (left) we plotted the packing produced by the baseline Shared Objects algorithm for SqueezeNet (the peak memory is marked by the purple dotted line and zoomed in Fig. 3b).

Peak memory usage occured when tensors `fire3/concat` (brown), `fire4/squeeze/relu` (blue), and `fire4/concat` (orange) were allocated at the same time.

Table 3 listed the lifetimes, shapes, and strides of the tensors. Because of the intersecting lifetimes of these tensors, the Shared Objects algorithm allocated a separate object for each of them.

TABLE 3. Analysis of SqueezeNet peak memory usage.

Tensor	Lifetime	Shape	Strides
`fire3/concat`	150 - 192	4x55x55x32	96800, 1760, 32, 1
`fire4/squeeze/relu`	183 - 225	1x55x55x32	96800, 1760, 32, 1
`fire4/concat`	204 - 254	4x55x55x32	96800, 1760, 32, 1

The TAA assigned the same offset of zero to all of these tensors. Fig. 3b (right) shows that, even though these tensors have overlapping lifetimes, decomposing these lifetimes into separate lifetimes for each tile makes it possible to allocate these tensors at the same offset with no data corruption.

Our algorithm may be ineffective for several reasons. When peak memory usage occurs owing to tensors with incompatible tile shapes, tile intersections may occur at any base tensor offset. An example of this is Inception V3, whose packing and peak memory usage, as determined by the TAA algorithm, are shown in Fig. 3c. Another reason for the ineffectiveness of TAA could be that peak memory usage occurs because of a single large feature map that dominates the other intermediate tensors. This is the case with MobileNet V2, whose input is much larger than of the other tensors in the off-chip memory, as shown in Fig. 3d.

V. CONCLUSION

In this paper, we presented a novel approach to tensor allocation that utilizes the restrictions of the Deep Learning model execution environment with a limited amount of on-chip memory. We evaluate this approach on various popular model architectures and show that, by reducing the lifetime of a tensor in external memory and considering tiles as constraints on memory locations for this tensor, it is possible to significantly reduce the peak external memory usage of these models. We note that, while this approach increases the model compilation time, this increase is not significant for the models considered.

REFERENCES

[1] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.

[2] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. 36th Int. Conf. Mach. Learn.*, vol. 97, K. Chaudhuri and R. Salakhutdinov, Eds., Jun. 2019, pp. 6105–6114.

[3] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," in *Proc. ECCV*, 2016, pp. 391–407.

[4] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[6] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang, "Conformer: Convolution-augmented transformer for speech recognition," 2020, *arXiv:2005.08100*.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[8] T. B. Brown et al., "Language models are few-shot learners," 2020, *arXiv:2005.14165*.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017, *arXiv:1706.03762*.

[10] N. Fatima, A. S. Imran, Z. Kastrati, S. M. Daudpota, and A. Soomro, "A systematic literature review on text generation using deep neural network models," *IEEE Access*, vol. 10, pp. 53490–53503, 2022, doi: 10.1109/ACCESS.2022.3174108.

[11] E. N. Crothers, N. Japkowicz, and H. L. Viktor, "Machine-generated text: A comprehensive survey of threat models and detection methods," *IEEE Access*, vol. 11, pp. 70977–71002, 2023, doi: 10.1109/ACCESS.2023.3294090.

[12] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85714–85728, 2020, doi: 10.1109/ACCESS.2020.2991734.

[13] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2018, *arXiv:1805.00907*.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, Carlsbad, CA, USA, Oct. 2018, pp. 578–594.

[15] S. Cyphers et al., "Intel nGraph: An intermediate representation, compiler, and executor for deep learning," 2018, *arXiv:1801.08058*.

[16] B. Hoon Ahn, J. Lee, J. Menjay Lin, H.-P. Cheng, J. Hou, and H. Esmaeilzadeh, "Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices," 2020, *arXiv:2003.02369*.

[17] R. Wei, L. Schwartz, and V. Adve, "DLVM: A modern compiler infrastructure for deep learning systems," 2017, *arXiv:1711.03016*.

[18] Y. Pisarchyk and J. Lee, "Efficient memory management for deep neural net inference," 2020, *arXiv:2001.03288*.

[19] R. Harren, K. Jansen, L. Prädel, and R. van Stee, "A $(5/3+\epsilon)$-approximation for strip packing," *Comput. Geometry*, vol. 47, no. 2, pp. 248–267, 2014.

[20] K. Jansen and M. Rau, "Improved approximation for two dimensional strip packing with polynomial bounded width," 2016, *arXiv:1610.04430*.

[21] Z. Chen and J. Chen, "An effective corner increment-based algorithm for the two-dimensional strip packing problem," *IEEE Access*, vol. 6, pp. 72906–72924, 2018, doi: 10.1109/ACCESS.2018.2882823.

[22] M. Chen, K. Li, D. Zhang, L. Zheng, and X. Fu, "Hierarchical search-embedded hybrid heuristic algorithm for two-dimensional strip packing problem," *IEEE Access*, vol. 7, pp. 179086–179103, 2019, doi: 10.1109/ACCESS.2019.2953531.

[23] L. Barenboim, R. Drucker, O. Zatulovsky, and E. Levi, "Memory allocation for neural networks using graph coloring," in *Proc. 23rd Int. Conf. Distrib. Comput. Netw.*, Jan. 2022, pp. 232–233.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014, *arXiv:1408.5093*.

[25] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 8024–8035.

[26] 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," 2015, *arXiv:1512.01274*.

[27] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016, *arXiv:1604.06174*.

[28] C.-J. Wu et al., "Machine learning at Facebook: Understanding inference at the edge," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 331–344.

[29] Y. Zhuang, S. Peng, X. Chen, S. Zhou, T. Zhi, W. Li, and S. Liu, "Deep Fusion: A software scheduling method for memory access optimization," in *Network and Parallel Computing*. Cham, Switzerland: Springer, 2019, pp. 277–288.

[30] S. Minakova and T. Stefanov, "Buffer sizes reduction for memory-efficient CNN inference on mobile and embedded devices," in *Proc. 23rd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2020, pp. 133–140.

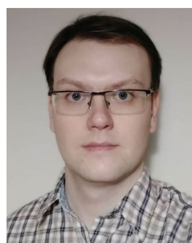
[31] A. Artemev, T. Roeder, and M. van der Wilk, "Memory safe computations with XLA compiler," 2022, *arXiv:2206.14148*.
[32] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*.
[33] P. Ramachandran, "Use of extended Euclidean algorithm in solving a system of linear diophantine equations with bounded variables," in *Proc. Int. Algorithmic Number Theory Symp.*, 2006, pp. 182–192.
[34] K. Aardal, C. A. J. Hurkens, and A. K. Lenstra, "Solving a system of linear diophantine equations with lower and upper bounds on the variables," *Math. Oper. Res.*, vol. 25, no. 3, pp. 427–442, Aug. 2000.
[35] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. 14th Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer-Verlag, 2008, pp. 337–340.
[36] R. Sousa, B. Jung, J. Kwak, M. Frank, and G. Araujo, "Efficient tensor slicing for multicore NPUs using memory burst modeling," in *Proc. IEEE 33rd Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Oct. 2021, pp. 84–93.
[37] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 4278–4284.
[38] K. He, S. Ren, J. Sun, and X. Zhang, "Identity mappings in deep residual networks," in *Proc. ECCV*, 2016, pp. 630–645.
[39] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5 MB model size," 2016, *arXiv:1602.07360*.
[40] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 2815–2823.
[41] Samsung. (2022). *Samsung/One: On-Device Neural Engine*. [Online]. Available: https://github.com/Samsung/ONE.git

ANDREY KVOCHKO received the B.S. degree in mathematics and computer science from Moscow Engineering Physics University, in 2010, and the M.S. degree in computer science from The University of Arizona, in 2013. From 2014 to 2022, he was an Expert Software Engineer with Samsung Research Russia, Moscow. Since 2022, he has been a Research Engineer with Lightricks, Israel. His primary research interests include image processing, compiler development, and deep learning accelerators.

EVGENII MALTSEV received the Ph.D. degree in computer science from Siberian Federal University, Krasnoyarsk, in 2017. From 2013 to 2018, he was an Associate Professor with Siberian Federal University. From 2018 to 2022, he was a Research Scientist with the Skolkovo Institute of Science and Technology, Moscow. Since 2022, he has been the Project Leader with the Advanced System Software Laboratory, Samsung Research Russia, Moscow. He is the author and coauthor of more than 20 articles on machine learning, computer-aided design, 3D printing, and remote sensing. His current research interests include software development, machine learning for edge devices, and neural processors.

ARTEM BALYSHEV received the B.S. and M.S. degrees in applied mathematics and physics from the Moscow Institute of Physics and Technology (MIPT), Moscow, in 2023. Since 2022, he has been a Software Engineer with the NN Compiler and Runtime Project, Samsung Research Russia, Moscow. His main research interests include system software development, machine learning, and learning on devices.

STANISLAV MALAKHOV received the B.S. and M.S. degrees in mechanics and applied mathematics from Lomonosov Moscow State University, Moscow, in 2005. Since 2022, he has been a Software Engineer with the Advanced System Software Laboratory, Samsung Research Russia, Moscow. His main research interest includes the compilation of deep neural networks for edge devices.



ALEXANDER EFIMOV received the Specialist degree in applied mathematics and computer science from Lomonosov Moscow State University, Moscow, in 2013. From 2018 to 2023, he was the Software Engineer/Tech Leader with the Advanced System Software Laboratory, Samsung Research Russia, Moscow. Since 2023, he has been a Senior Software Engineer with the AI/ML GEMS Team, Luxoft Serbia, Belgrade. His main research interests include system software development, ML frameworks, and runtimes.