CHALMERS
UNIVERSITY OF TECHNOLOGY

Scratchpad Memory Management for Deep Learning Accelerators

Downloaded from: https://research.chalmers.se, 2026-04-11 21:41 UTC

Citation for the original published paper (version of record):

Zouzoula, S., Maleki, M., Azhar, M. et al (2024). Scratchpad Memory Management for Deep Learning Accelerators. ACM International Conference Proceeding Series: 629-639.
http://dx.doi.org/10.1145/3673038.3673115

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Scratchpad Memory Management for Deep Learning Accelerators

Stavroula Zouzoula
Chalmers University of Technology
Gothenburg, Sweden
zouzoula@chalmers.se

Mohammad Ali Maleki
Chalmers University of Technology
Gothenburg, Sweden
mohammad.ali.maleki@chalmers.se

Muhammad Waqar Azhar
ZeroPoint Technologies AB
Gothenburg, Sweden
waqar.azhar@zptcorp.com

Pedro Trancoso
Chalmers University of Technology
Gothenburg, Sweden
ppedro@chalmers.se

ABSTRACT

The success of Artificial Intelligence (AI) applications is driven by efficient hardware accelerators. Recent trends show a rapid increase in the application demands, which in most cases surpass the available resources in the accelerators. As such, the efficient management of these limited resources becomes a critical factor in achieving high-performance.

In this work we focus on the management of the available on-chip memory resources for Deep Learning (DL) accelerators. While most state-of-the-art accelerators have static buffer separation for different data types, we observed that the heterogeneity of recent DL models demands for more flexible solutions. In this work we propose using all on-chip scratchpad memory, including space for double buffering, in a unified way. To efficiently exploit that space, we propose a memory management technique that can apply different policies to best meet the demands of each different execution phase. For cases when the available memory is less than the requirements, the memory management can use the available space for either optimizing the data reuse or the fetching of data ahead.

Comparing our approach against a baseline accelerator shows that the flexibility in the management of the scratchpad memory leads to a considerable reduction of up to 80% of the off-chip memory accesses, or up to 56% of the latency.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; • **Computing methodologies → Modeling and simulation**.

KEYWORDS

Deep Learning Accelerators, Scratchpad, Memory Management

ACM Reference Format:
Stavroula Zouzoula, Mohammad Ali Maleki, Muhammad Waqar Azhar, and Pedro Trancoso. 2024. Scratchpad Memory Management for Deep Learning Accelerators. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673115

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPP '24, August 12–15, 2024, Gotland, Sweden
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673115

1 INTRODUCTION

With the rise of AI, there is an increased demand for the execution of complex and resource hungry Deep Learning (DL) applications across devices spanning the entire compute continuum. This puts a considerable pressure on the hardware leading to the fast development of domain-specific architectures, also known as accelerators. Recent examples include the stand-alone Google Tensor Processing Unit (TPU) [14] or the Neural Processing Units (NPU) in many general-purpose CPUs such as in the Intel processors [8]. While there are many such accelerators in the market, in order for the hardware to be able to execute efficiently the different and evolving models, the hardware is composed of simple basic generic modules. Typically, these modules include units to accelerate matrix operations (e.g. systolic array [17]) and local buffers (e.g. scratchpads [2]) to capture data reuse. More complex and dedicated accelerators (e.g. FPGA-based [26]) are required in order to achieve better performance and/or efficiency.

This work focuses on the management of the on-chip memory for DL hardware accelerators. Although DL applications have usually a very large memory footprint [28], their execution pattern exhibits a considerable amount of data reuse. Consequently, in most cases, DL accelerators come with a considerable amount of local memory. The memory accesses for DL applications are regular and deterministic [21]. As such, the local memory modules are usually simple software-managed buffers also known as scratchpads [2]. As the different data types (i.e. input, output and filters) have different access patterns and exhibit different types of reuse, the common approach in accelerators is to have separate memory modules, one dedicated to each data type. For example, the Google TPU v1 has a 24MB local buffer for the input and output and a separate buffer for the filters [14]. While this makes the hardware and control simple, it is not able to capture the variety of requirements of the different DL models and their layers.

The core of a DL application is the execution of a pre-trained multi-layer deep neural network model. A model is usually executed in a layer-by-layer manner and each layer may have different data footprint, access, and reuse patterns [19]. Some recent accelerators handle these different layer requirements by offering a single local buffer for all the data types but then rely on sophisticated mapping schemes to make best use of the available space [7, 39, 41]. Nevertheless, most of these works require changes in the algorithm's dataflow or accesses patterns to match the hardware. Furthermore, frequent changes in models being executed, as well as support for multi-tenancy [20] require more efficient mappings. In addition,

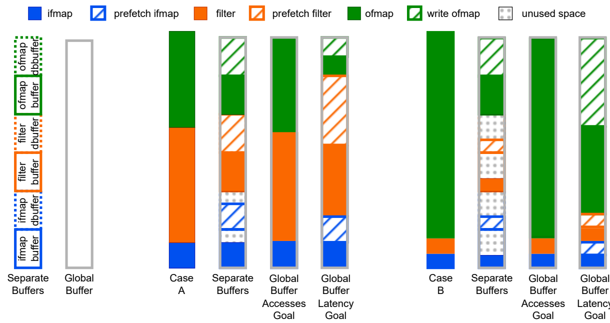


Figure 1: Case A requires large space for filters while case B requires large space for output feature maps. A global buffer with a management scheme is a flexible way to able to capture more reuse or have more space for prefetching than separate buffers.

almost all accelerators optimize the execution by providing a memory space for double buffering [27] to overlap the execution of a layer with the prefetching of the data for the next layer. This space is used exclusively for data prefetching and can not be exploited for data reuse.

It is important to note that the on-chip memory space is used for two purposes: (1) to avoid costly unnecessary off-chip memory accesses by storing data that is reused during execution; and (2) to bring data ahead of its use (prefetching), thereby reducing the wait time for costly off-chip memory accesses.

The goal of this work is to exploit the complete on-chip memory space in a more flexible way, as to adapt to the increasing demands of the ever evolving models (e.g. memory requirements, diverse access and reuse patterns). At the same time, we want to provide a technique that supports algorithm optimizations without relying on complex dedicated software and/or hardware implementations. Consequently we propose a solution that exploits a single on-chip scratchpad memory (i.e. global buffer) designed to meet the needs of various DL applications data types. The effective use of this scratchpad is achieved with a flexible software memory management technique. This technique uses lightweight estimation models to statically determine how to partition the available space among different data types and how much space to reserve for prefetching. The objective for the memory management is to identify the optimal data reuse given the constraints and goals. Our focus is on estimating the potential benefits and effectiveness of the approach.

Figure 1 shows the benefits of using a global buffer with flexible memory management compared to fixed separate buffers, for two different cases. The diagram does not represent real values but is inspired by real requirements observed for the ResNet18 model (see Figure 3). The data requirements are shown as a breakdown for each different data type: input, also known as input feature map (ifmap); output, also known as output feature map (ofmap); and weights or filters. For each case we show three mappings of data to the available on-chip memory space: (1) mapping for the separate buffer setup; (2) mapping for the global buffer setup with goal of reducing the accesses; and (3) mapping for the global buffer setup with goal of reducing the latency. The solid pattern shows data stored for reuse while the dashed shows data stored for prefetching. We clearly observe that in both case A and B the data reuse and prefetching can not be satisfied for the separate buffer even though there is space available, while for the global buffer with the management scheme it is possible to utilize the complete available space for more reuse (access goal) or prefetching (latency goal).

The main contributions of our work are the following:

- A set of different policies for the scratchpad memory management that exploit different data reuse patterns.
- A mechanism that decides which policy to use for each different layer of the model execution in order to optimize the execution for reduction of memory accesses or latency.
- The evaluation of the proposed memory management schemes for different memory sizes and different goals.

Our evaluation using six well known DL models shows that our proposed scratchpad memory management technique reduces the off-chip memory accesses up to 80% or the latency up to 56% when compared to the baseline, a SCALE-Sim simulated system representing a standard accelerator with separate buffers. We also show the trade-off in using the available on-chip memory space for achieving different goals. For example, for the smaller buffer size, we also observed that optimizing for latency, as opposed to off-chip accesses, resulted in a reduction of 23% in the latency with a cost of an increase in 33% in the off-chip accesses.

2 BACKGROUND

This section covers the background and related information relevant to our work. DL models, such as Convolutional Neural Networks (CNNs), are widely used in a variety of applications, such as computer vision [3, 12, 13], image processing [16] and speech recognition [1]. Performance and efficiency requirements have led to the development of hardware accelerators for these applications. While a multitude of DL accelerators have been proposed in the past [11], our work is somehow oblivious to the different proposed computational engines and the scope is focused on the on-chip memory subsystems of these accelerators and the approaches used to manage that available memory space.

2.1 DL accelerator memory systems

In order to satisfy the increasing memory requirement demands of modern AI applications, DL accelerators usually include a large off-chip memory which is complemented with smaller on-chip scratchpad memories (or buffers) for capturing the local reuse of the data, as to reduce the off-chip data transfers. The most common on-chip memory systems are:

- **Separate buffers for each data type** [5, 10, 29, 37]. In this system, each data type, i.e. input feature map (ifmap), filters and output feature map (ofmap), is stored on a different on-chip buffer. This setup, which can be seen as an extension of a set of dedicated registers, has the simplest hardware but no flexibility in the use of its space.
- **Activation buffer and a filter buffer** [4, 15, 23, 34]. In this system there are two separate buffers: a unified buffer for the ifmap and ofmap and a separate buffer for the filters. In this setup the sharing is limited so the hardware complexity does not increase significantly and it allows for more flexibility

than in the separate buffers. By allowing the input and the output to be on the same buffer, this opens the possibility for data reuse across layers as the results (outputs) of one layer can be reused directly from the buffer as inputs to the next executing layer (*inter-layer reuse*).

- **Global buffer (GLB)** [7, 9, 36, 38, 39, 41]. In this system, all data types are stored in a single memory unit, the GLB. This is the most flexible setup as accommodates any data types of any size as long as they fit in the available space. In terms of hardware there is an increasing complexity in the access to the shared memory space but not enough to overshadow the potential benefits of the extra captured data reuse.

In the context of this paper, we focus on accelerators with a GLB. The main reason is that GLB-based systems give the opportunity to exploit inter-layer reuse, which leads to reduced off-chip data transfers. Also, due to the data types heterogeneity (see Section 3.3), a single buffer can lead to better utilization of the available memory space and improve the data reuse. In addition, instead of keeping a separate space for the working and prefetching data as in most accelerators, we exploit the use of a scratchpad memory that combines all on-chip memory resources into a single pool that can be used in a flexible manner. A software memory management scheme is then proposed to use that space in the best way according to a particular objective such as latency reduction.

2.2 Reuse patterns in model layers

As mentioned before, modern accelerators employ on-chip scratchpad memories for local reuse of the data types. The data types of a fully connected or convolutional layer (ifmap, filter, ofmap) are described by the hyperparameters listed in Table 1. The main reuse types are *global, inter-* and *intra-layer reuse*:

Global reuse: the network filters (also known as weights) are stored on-chip, so they are used every time a new input is fed into the model for processing.

Inter-layer reuse: the output of a layer is used as input for the next layer.

Intra-layer reuse: there are two main data reuse patters in effect:

- *ifmap reuse*: Each ifmap is reused by multiple filters to generate the ofmap channels. In addition, each ifmap element is reused from the same filter approximately $F_H * F_W$ times, while sliding over the ifmap to generate the ofmap elements.
- *filter reuse*: Each filter is reused $O_H * O_W$ times on the same ifmap to generate one ofmap channel. Filters are also reused by multiple inputs.

2.3 Scheduling

Scheduling of the data on the scratchpad memory and later on the accelerator's processing unit affects the efficiency of the execution. Scheduling is affected by two factors, the available on-chip memory size and the underlying dataflow. In most cases, the on-chip memory size is not large enough to store all the data types (input, filter, output) of even a single layer of a neural network. To mitigate the impact of limited on-chip memory, tiling emerges as a valuable technique for enabling the execution of neural network computations within the available limited memory budget.

Table 1: Hyperparameters of a model layer.

Hyperparameter	Description
I_H / I_W	ifmap height / width
F_H / F_W	filter height / width
C_I	number of ifmap / filter channels
$F_\#$	number of 3D filters
O_H / O_W	ofmap height / width
C_O	number of ofmap channels
S	stride
P	padding

Tiling: Tiling involves partitioning the neural network computation graph into smaller tiles that fit within the available memory space. Due to *intra-layer reuse*, the same data is used multiple times. As the network is partitioned in tiles, some data might need to be loaded multiple times. For example, in a convolution layer, an ifmap tile is needed by all filters. If the filters are tiled as well, then the ifmap or filter tile will have to be loaded from the off-chip memory multiple times. These multiple loads increase the number of off-chip memory accesses. As the off-chip data transfers are the most energy costly operations [18], approximately 10-100x of the energy for a local computation, their increase leads to higher energy consumption. In addition, the limited bandwidth of the off-chip memories in combination with the increased need of transferring tiles can lead to latency overheads due to contention.

Different tiling methods that aim to minimize the off-chip memory accesses have been proposed before. In [28] the authors present some memory schemes for reading each element from the off-chip memory only once and one scheme with working sets of filters and activations. However, the memory requirements, even for the least memory demanding scheme, can be in the order of megabytes.

The main goal of [6, 22, 33, 40] is to reduce the energy consumption of the system by reducing the off-chip accesses. In [33], the authors try to identify tile sizes that will lead to reduced off-chip accesses, while considering the architectural parameters. Their approach is design space exploration (DSE), where they check all possible combinations, leading to a time consuming process. In [22] the authors use loop transformations (interchange and tiling) as to maximize reuse under memory constraint. In [40] the authors decide on the tile sizes by checking the off-chip accesses, aiming to keep as many outputs on-chip as possible. However, the ofmap buffer size remains static for the entire model execution.

In [27] the authors try to find the optimal tiling parameters for input and output when using batching, but focus on the fully connected (FC) layers only. In [35] the authors work with the architectural-dataflow co-design problem for CNN accelerators and find tile loop permutation and tile sizes that minimize the total energy for the execution through DSE.

Dataflow: Dataflow exploits reuse of specific data for reducing the off-chip communication. The existing dataflows are: (1) *weight stationary (WS)*, where the filter weights remain stationary in the register file (RF); (2) *input stationary (IS)*, where the input pixels remain stationary in the RF; (3) *output stationary (OS)*, where the output pixels remain stationary in the RF (the partial sums are stored in the same RF); (4) *row stationary (RS)* [7], where one row

of filter weights and one row of input pixels stay stationary in the processing element (PE); and (5) *no local reuse (NLR)* where there is no data reuse at the RF level.

Most common accelerators use one of the above described dataflows, but there are works such as [19] that exploit adaptive dataflows and tiling sizes for each layer of a network, as to minimize the off-chip communication and therefore the energy consumption.

In our work we propose a software memory management scheme to use the scratchpad memory space in the best way for the particular model execution phase, compute engine configuration, memory buffer size constraint, and target goal, which in our case is reduction of off-chip accesses or latency. We explore this using policies that estimate the key metrics for the decisions in a lightweight way allowing for the memory management to change dynamically even as the requirements change during runtime. In particular, our policies use different tiling techniques to exploit the data reuse types mentioned above and a combination of dataflows (OS, NLR).

3 METHODOLOGY

The main objective of this work is to manage the available on-chip memory space for exploiting data reuse and prefetching. A critical constraint is the size of the on-chip buffer (Section 3.1). Due to the memory constraint, we identify some memory management policies (i.e. tiling and reuse pattern) that keep the off-chip data transfers close to minimum (Section 3.2). Finally, we formulate an algorithm that matches the per layer requirements of a model with a policy considering the execution target (Section 3.3).

3.1 Optimization problem formulation

In this work we consider two optimization objectives.

Objective 1: Reduce off-chip data transfers for a network under memory constraint.

Objective 2: Reduce latency for a network under memory constraint.

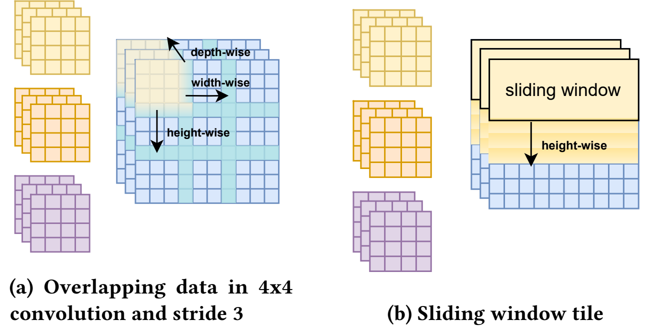
The constraint of the objectives is the size of the on-chip GLB, as it limits the amount of data that can be stored. For deciding on the amount of data stored we take two approaches: (1) use the entire space for loading data for the current execution and (2) reserve space for prefetching of the data types for the next execution. The second approach is examined for reducing latency by hiding transfer time with compute time. The constraints for the two approaches are described in Equation (1) and Equation (2), respectively. The tile sizes of the data types can be either of the ones described in Section 3.2.

$$\text{GLB} \geq I_{\text{Tile}} + F_{\text{Tile}} + O_{\text{Tile}} \quad (1)$$

$$\text{GLB} \geq 2 * I_{\text{Tile}} + 2 * F_{\text{Tile}} + 2 * O_{\text{Tile}} \quad (2)$$

3.2 On-chip memory policies

In Section 2.2 we described the main reuse types in convolutional layers. In ***intra-layer reuse*** the off-chip transfers are minimized, i.e. each element is transferred only once. To take advantage of *intra-layer reuse* the on-chip storage should be large enough to fit the entire layer, which can be some hundreds of kB. However, based on the *intra-layer reuse* patterns, we identify some memory



(a) Overlapping data in 4x4 convolution and stride 3 **(b) Sliding window tile**

Figure 2: Ifmap access directions and elements re-load.

management policies to reduce the per layer memory requirements while keeping the number of off-chip data transfers to the minimum.

Policy 1. ifmap reuse: For taking advantage of *ifmap reuse*, all the filters of a layer need to be stored on-chip. The ifmap can be loaded in tiles of size $F_H * I_W * C_I$ in a height-wise manner. The ofmap size should be equal to $1 * O_W * C_O$ to hold the results from the inner product of the F_H ifmap rows with all the filters of the layer. By sizing the ifmap tile to $F_H * I_W * C_I$, i.e. a *sliding window*, and load the tiles height-wise, we completely exploit *ifmap reuse*, as the ifmap elements need to be loaded only once from off-chip (Figure 2b). If the ifmap tile's dimensions are smaller than I_H, I_W, C_I, then depending on the access direction, a number of elements needs to be re-loaded. Figure 2a shows the three access directions, height-wise, width-wise and depth-wise, and denotes with turquoise colour the elements that need to be re-loaded.

Policy 2. filter reuse: For making the most of *filter reuse*, the entire ifmap of a layer needs to be stored on-chip. The filters can be loaded one-by-one and the ofmap size should be equal to $O_H * O_W$ to hold the results from the inner product of one filter with the ifmap.

Policy 3. per channel reuse: An important detail about convolutions, is that we have reuse only per channels. That means that one channel of the ifmap is reused only by one channel of each filter. Considering the reuse only per channels, we can benefit from *ifmap reuse* by storing on-chip only one channel of all the filters of the layer, $F_H * F_W * F_\#$. The ifmap can be loaded in tiles of size $F_H * I_W$ in a height-wise manner. In this case, the ofmap should be sized to hold the output results for the entire layer, $O_H * O_W * C_O$.

In the above described policies each element is transferred only once from/to off-chip memory. However, most of these techniques require high memory storage for some layers of the network, e.g., EfficientNetB0 requires 1491.9kB for intra-layer reuse and 1252.3kB for policy 3. To overcome the high memory demands, we modify policies 1 and 3 so that the filters and ofmap elements are transferred only once, while the transfers for the ifmap elements may increase.

Policy 4. partial ifmap reuse: In this policy, similar to policy 1, ifmap is loaded in sliding window tiles of size $F_H * I_W * C_I$. The filters are loaded in blocks of n filters each, $F_H * F_W * C_I * n$, where $n = [1, F_\#)$. The ifmap will be loaded from off-chip memory $x = \lceil \frac{F_\#}{n} \rceil$ times. The ofmap is sized to store the results from the inner product of the ifmap tile with the n filters of the layer, $1 * O_W * n$.

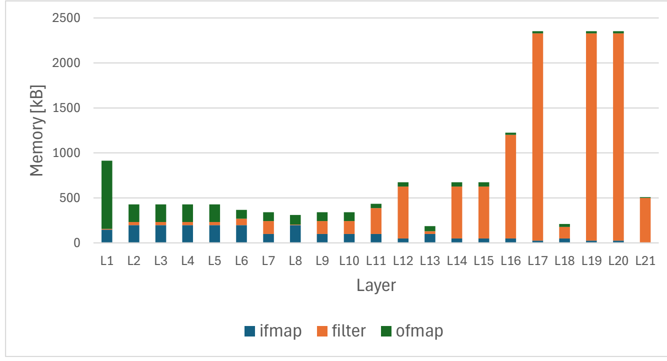


Figure 3: Memory breakdown into the different data types for each layer of the ResNet18 model.

Note that this policy can lead to high memory requirements when C_I is large.

Policy 5. partial per channel reuse: In this policy, similar to policy 3 and 4, ifmap is loaded in sliding window tiles of one channel only in height-wise manner, $F_H * I_W$. The filters are loaded in blocks of n filters each, one channel per filter, $F_H * F_W * n$, leading in loading the ifmap from off-chip memory $x = \lceil \frac{F_\#}{n} \rceil$ times. The ofmap is sized to hold the results deriving from the n filters, $O_H * O_W * n$.

3.3 Memory management technique

CNNs are highly heterogeneous, requiring different memory sizes for each layer and for each data type. Figure 3 shows the memory breakdown of the different data types for each layer of the ResNet18 model. As we observe, the first layers require more memory for the ifmap and ofmap, while the last layers require more memory for the filters. As it was shown already in Figure 1, a fixed memory partitioning would lead to under-utilized memory for some data types, while that empty space could be used for another data type. This inherent heterogeneity in layers strengthens the decision for heterogeneous on-chip memory management, that can be adapted to the layer's needs.

Motivated by the above fact, we formulate an algorithm to manage the scratchpad memory per layer in a neural network, aiming to reduce the off-chip accesses or latency. The operational flow of our memory management technique, based on our previously developed RAINBOW tool [42], is depicted in Figure 4. The inputs to the algorithm are a CNN model description and accelerator specifications (operation per cycle, data width, GLB size, off-chip memory bandwidth). The CNN model description is generated through code that translates TensorFlow or Pytorch models to the input format of the system. The accelerator specifications are user defined, based on the desired system to examine.

The memory management algorithm integrates the *intra-layer reuse* pattern and the reuse policies outlined in Section 3.2. For each policy the memory requirements and the number of off-chip accesses are estimated based on the sizes and accesses described in Section 3.2 and the latency based on the number of operations, bandwidth and tile sizes for each layer of the model. The simplest way to execute a model is to apply the same policy to each layer

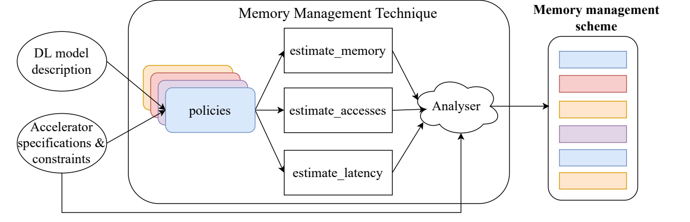


Figure 4: Operational flow of our proposed memory management technique.

of the model. This is what we call a *homogeneous execution plan* or *homogeneous management scheme*. For each policy we can thus get a different homogeneous management scheme.

The estimation results are fed to an analyser, that based on the GLB size constraint and optimization objective, decides on which policy should be applied to each layer. The output of the analyser is a *heterogeneous execution plan* or *heterogeneous management scheme*, i.e. a plan where each layer of the model may use a different management policy, one that satisfies better the problem objective. Algorithm 1 describes how the memory management technique works, when the objective is the reduction of off-chip accesses. Line 1 defines the policies integrated in the system. In lines 7-9 the memory requirements, accesses and latency of each layer for each policy are estimated. The analyser is described in lines 10-16. If the condition in line 10 for one layer is not true for any of the policies, then we have to search for appropriate tile sizes that will satisfy the condition. This may lead to an increased off-chip accesses.

Algorithm 1 Finding per layer policy when targeting minimum accesses

Input: CNN, GLB_size
Output: per_layer_policy

```
1:  policies = {intra-layer reuse, intra-layer reuse with prefetching,
    policy 1-5, policy 1-5 with prefetching}
2:  for layer ∈ CNN do
3:      min_accesses ← ∞
4:      min_latency ← ∞
5:      layer_policy ← NULL
6:      for policy ∈ policies do
7:          memory = estimate_memory(policy)
8:          accesses = estimate_accesses(policy)
9:          latency = estimate_latency(policy)
10:         if memory ≤ GLB_size then
11:             if accesses < min_accesses then
12:                 per_layer_policy[layer] ← policy
13:             else if accesses = min_accesses then
14:                 if latency < min_latency then
15:                     per_layer_policy[layer] ← policy
16:                 end if
17:             end if
18:         end if
19:     end for
20: end for
```

Table 2: Characteristics of the DL models studied. Types of layer can be: convolution (CV), depth-wise convolution (DW), point-wise convolution (PW), fully-connected (FC), or projection layer (PL).

Network	Number of Layers	Types of Layers
EfficientNetB0 [32]	82	CV, DW, PW, FC
GoogLeNet [30]	64	CV, PW, FC
MnasNet [31]	53	CV, DW, PW, FC
MobileNet [13]	28	CV, DW, PW, FC
MobileNetV2 [25]	53	CV, DW, PW, FC
ResNet18 [12]	21	CV, PW, FC, PL

Table 3: Maximum memory requirements in kB for policies where each element is transferred only once.

Network	Policies			
	intra-layer reuse	Policy 1	Policy 2	Policy 3
EfficientNetB0	1491.9	1176.2	1201	1252.3
GoogLeNet	2051	788.6	199.7	2051
MnasNet	1252.3	588.2	591.5	1252.3
MobileNet	1178	784.2	801.7	1038
MobileNetV2	1491.9	1176.2	1201	1252.3
ResNet18	2353	788.6	199.7	2318

4 EXPERIMENTAL SETUP

The DL models used in this work are described in Table 2. We have selected a set of well known standard DL models used in computer vision applications, with focus on image classification. These models have different characteristics (Table 2) and various memory requirements for the different policies. In Table 3 we present the memory requirements for *intra-layer reuse* and policies 1-3. Policies 4 and 5 are memory-dependent, which means that their requirements are constrained by the GLB size.

We have implemented our proposed methodology according to the description in Section 3.3. The execution of the models is assumed to be done layer-by-layer, in accordance to the baseline [24], which means that the residual connections present in some CNNs (e.g. ResNet18) are serialized. As mentioned before, the accelerator specifications and constraints are the number of operations per cycle (OPs), the data width, the GLB size and the bandwidth to off-chip memory. The number of multiply and accumulate (MAC) operations is half the number of OPs, as it requires two cycles to complete. The simulated architecture consists of 16×16 processing elements (PEs), a small size that results in higher PE utilization for the smaller tiles used. For that reason, the OPs are set to 512. The data width is 8-bits and the off-chip memory bandwidth is set to 16 elements per cycle, matching the maximum average bandwidth of the baseline. The on-chip memory bandwidth is assumed to be enough to match the demands of the PEs for both our setup and the baseline. The GLB sizes tested are 64kB, 128kB, 256kB, 512kB and 1024kB, a range from small to large on-chip memory capacities for the particular PE array. The estimations are for inference with a batch size of 1, as it is the most appropriate for latency constrained applications.

The baseline used is a systolic array implemented on the SCALE-Sim simulator [24]. We chose SCALE-Sim as it is a well known simulator and a representative of systolic array accelerators [4, 15]. The architecture selected for the baseline consists of 16×16 PEs to match our implementation with output stationary dataflow and the range of buffer sizes from the mentioned set. Since we used output stationary dataflow, we allocated a small ofmap buffer size of 4kB for all configurations. The remaining memory was distributed between the ifmap and filter buffers in ratios of 25-75%, 50-50% and 75-25%. It should be noted that the buffers in SCALE-Sim are double-buffered. However, instead of requiring additional space, the assigned buffer size is divided in half. One partition holds the active data, while the other is used for prefetching. We created those three baselines for exploring different points in the design space that could result in benefit for different layers of the CNNs.

It is relevant to note that it took approximately one minute to generate the management schemes for all the tested models on an Intel Core i7-1185G7 CPU (@3.00GHz), while for the SCALE-Sim baseline it took more than 5 hours. The difference in the runtime of our approach and the baseline comes from the implementation. Our approach is based on heuristics (i.e. memory management policies) that are integrated in the implementation leading to fast generation of the estimations. On the other hand, the baseline is a full simulator of a systolic array accelerator that produces also the traces to the off-chip memory, leading to high runtime. Results from our methodology have been validated against [28].

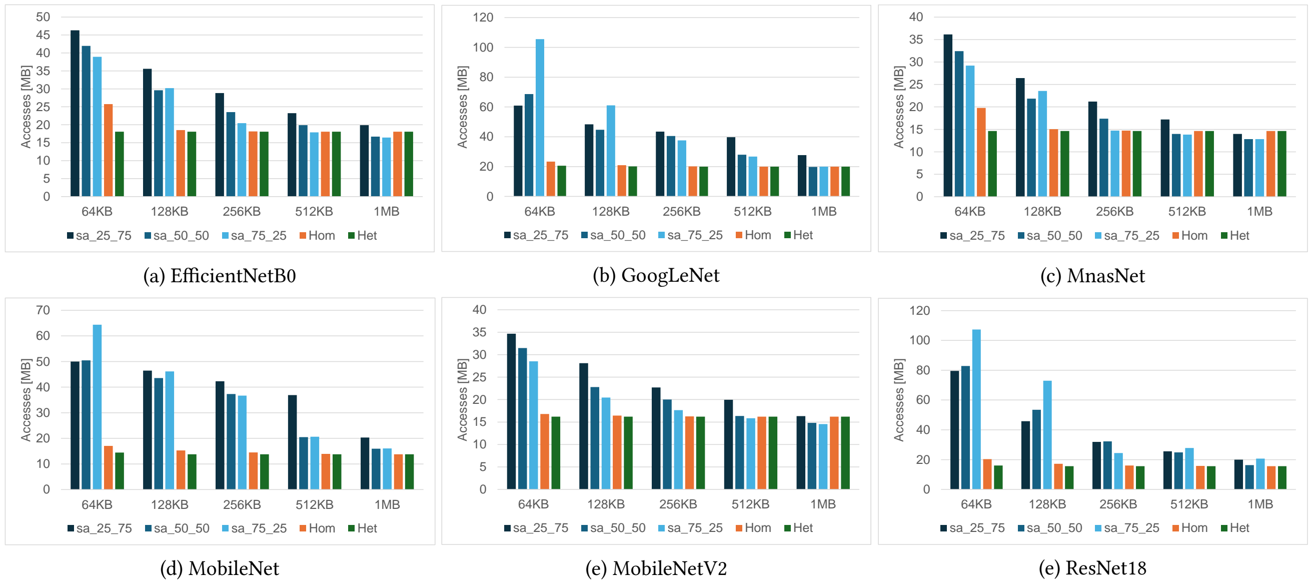
5 RESULTS

The next sections present the estimated performance metrics for the simulated architectures. Experiments for the two objectives, optimize for off-chip accesses (Section 5.1) and optimize for latency (Section 5.2) were performed. Section 5.3 analyses the impact on accesses/latency when optimizing for latency/accesses, respectively. Section 5.4 investigates the benefits of enabling *inter-layer reuse*.

5.1 Optimize for accesses

The first results we present, show how important it is to partition the on-chip memory space in a flexible way as to capture the different reuse patterns and layer footprints. In Figure 5 we present the volume of accesses to off-chip memory in MB for different on-chip buffer memory size configurations and all target DL models. Each chart includes 5 bars corresponding to the memory accesses for the different schemes: baseline with a fixed partition of 25-75% ifmap-filters (*sa_25_75*), baseline with a fixed partition of 50-50% (*sa_50_50*), baseline with a fixed partition of 75-25% (*sa_75_25*), our proposed homogeneous management scheme that offers the least off-chip accesses (*Hom*) and our heterogeneous management scheme that offers the least off-chip accesses (*Het*). Note that *Hom* is the scheme where each layer is executed using the same policy, while *Het* is the scheme that is executed using the best policy for each of its layers (see Section 3.3).

From the different baseline configuration results we can confirm that there is not a single fixed way of splitting the memory between the features that achieves the best results for all models.



(a) EfficientNetB0 (b) GoogLeNet (c) MnasNet

(d) MobileNet (e) MobileNetV2 (e) ResNet18

Figure 5: Volume of off-chip memory accesses for the baseline configurations and proposed schemes for different models and buffer sizes.

We can observe two clear trends. For *EfficientNetB0*, *MnasNet*, and *MobileNetV2*, the baseline benefits from a larger portion of memory assigned to the *ifmap* (*sa_75_25*). For *GoogLeNet*, *MobileNet*, and *ResNet18*, the results show the opposite effect, the baseline for those models benefits if configured with a larger portion of the memory assigned to the *filters* (*sa_25_75*). That is because the models in the first group have larger ifmaps while the models in the second group have larger filters (see Figure 3 for *ResNet18*). This proves the point that relying on a fixed partitioning of the on-chip memory space leads to a sub-optimal solution since a larger portion should be awarded to the dominant feature of a model and this changes from model to model.

When comparing our proposed management schemes (*Hom* and *Het*) to the baseline configurations, we observe that our schemes are able to significantly reduce the number of accesses for the smaller buffer sizes. For the 64kB memory buffer for *Hom* the reduction ranges from 32.2% for *MnasNet* to 74.5% for *ReNet18* while for *Het* the reduction ranges from 43.2% for *MobileNetV2* to 79.8% for *ResNet18*.

Regarding the larger buffer sizes, the differences between the baseline configurations and the proposed schemes are not significant for most cases. It is important to note that these buffer sizes are actually quite large. With a 16×16 engine configuration, 1MB buffer means that there is 4kB per PE which is a much larger value than what is found in current accelerators (e.g. the Google TPU has 0.375kB per PE [14]). For *EfficientNetB0*, *MnasNet*, and *MobileNetV2* for the 1MB buffer size we can observe that both *Hom* and *Het* show slightly higher accesses compared to the baseline. This is due to the fact that unlike in the baseline, we consider padding of the ifmap in our estimations. That means that all the comparisons are not fair and the benefit presented from our policies is actually higher.

Another observation is that for *Het* the number of accesses is almost constant independent of the buffer size, meaning that the flexibility offered by this scheme enables it to be able to capture the minimum accesses from the smallest buffer size. Table 4 contains the policies used in each of the networks. From the description

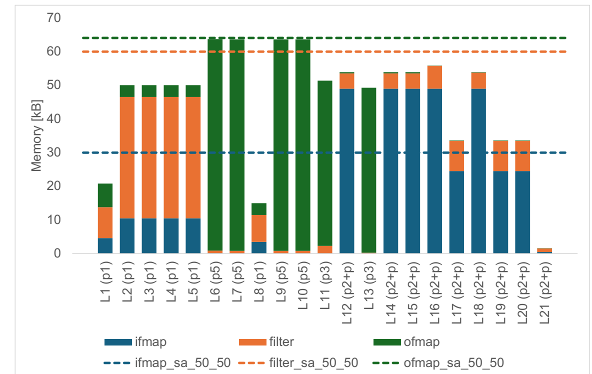


Figure 6: Heterogeneous scheme memory breakdown for ResNet18 with buffer size of 64kB.

Table 4: Memory policies for 64kB GLB size.

Network	Memory policies used
EfficientNetB0	intra-layer reuse (+p), policy 1 (+p), policy 2 +p, policy 3 (+p), policy 5 +p
GoogLeNet	intra-layer reuse (+p), policy 1 (+p), policy 2 +p, policy 3 (+p), policy 4, policy 5
MnasNet	policy 1 (+p), policy 2 +p, policy 3 (+p)
MobileNet	policy 1, policy 2, policy 3, policy 4, policy 5
MobileNetV2	intra-layer reuse, policy 1, policy 2, policy 3
ResNet18	policy 1, policy 2, policy 3, policy 5

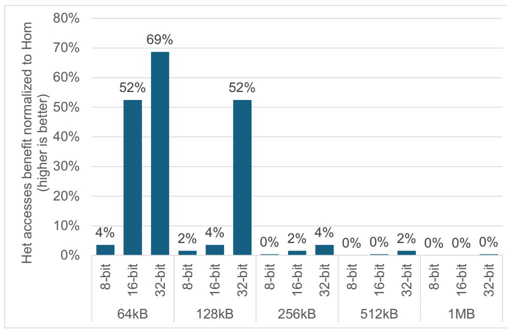


Figure 7: Benefit of using the heterogeneous over homogeneous scheme for reduction of off-chip accesses for different data width sizes and buffer sizes for MobileNetV2.

in Section 3.2, we know that for all policies, except 4 and 5, the elements are transferred only once from/to the off-chip memory. However, policies 4 and 5 can also achieve minimum transfers for depth-wise layers, which contain only one filter. In *EfficientNetB0*, policy 5 is applied only on some of the depth-wise layers, resulting in the minimum off-chip communication for the total network.

The reason for the larger benefit for the smaller buffer configurations is related to the fact that it is for those sizes that the fixed partitions of the baseline configurations are less able to capture the different requirements of the different layers for the models tested. Figure 6 depicts the memory breakdown for the different features of our *Het* scheme for the execution of the *ResNet18* model with a 64kB buffer configuration. The stacked bars represent the memory allocated by our memory management to the different features (*ifmap*, *filter* and *ofmap*) for the execution of each layer (L1-21) of the model. In the label in parenthesis we present the policy selected by the memory management (p1-5) and if prefetching was applied to that particular layer (*+p*). The dashed lines represent the static partition of a baseline with a 50-50% memory distribution (*sa_50_50*). From these results it is obvious that the fixed partition approach is not able to capture the required flexibility. In the layers at the beginning (L2-5) the largest portion is assigned to the *filter*, for the middle layers the largest portion is assigned to the *ofmap*, while for the layers at the end the largest portion is assigned to the *ifmap*. In most cases the baseline fixed partitions are not able to store all the data needed for the corresponding feature. In contrast, the proposed *Het* scheme captures this heterogeneity with applying different policies to the different layers: *p1* to the first, *p5* to the middle and *p2+p* to the last layers.

While we observe heterogeneity in the models and their execution over time, the differences observed between the off-chip accesses achieved by the two proposed schemes - *Hom* and *Het* - are minimal except for *EfficientNetB0* and *MnasNet* for the 64kB buffer size. This is due to the fact that not only *Het* but also *Hom* are able to achieve the minimum accesses as described above. This is more evident for the results presented when we are using a data width of 8-bit. In Figure 7 we show what is the impact on the off-chip accesses for different data widths and in particular what is the benefit for the different data widths for using the *Het* instead of the *Hom* scheme. From these results we can observe that indeed for the 32-bit data width the *Het* scheme is able to reduce further the off-chip accesses by 69% for 64kB and 52% for 128kB when compared to using the *Hom* scheme. This difference fades out for larger buffer sizes but still shows the relevance of using the *Het* scheme for the cases when there is increased pressure on the available on-chip memory space.

Overall, the proposed memory management is able to exploit the flexibility of using the single buffer to capture the changing requirements for the different models and different execution layers of each model. The significant reduction in the number of accesses (up to 80% for *Resnet18*) for the smaller buffer sizes means that we are able to offer a solution that is bale to considerably reduce the energy consumption for that setup which is extremely relevant for example for small battery operated accelerators.

5.2 Optimize for latency

As mentioned previously, the flexible management of the buffer space means that we are able to try to exploit its space for different purposes, such as using a portion of it for prefetching data used in a next phase (tile or layer) and thus be able to overlap the memory loads with the model execution, leading to a reduction in the latency.

In Figure 8 we present the inference latency achieved by the baseline and the proposed schemes optimized for reduces accesses (*Hom_a, Het_a*) and optimized for reduced latency (*Hom_l, Het_l*). The baseline SCALE-Sim simulation is done for zero stalls therefore the latency is not affected by the sizes of the buffers and thus we present only one baseline configuration bar in the charts. Our *Hom* and *Het* results are done with a fixed bandwidth equal to the maximum of the reported SCALE-Sim average bandwidth. As such, our results will show higher latency as we will consider the cases of the peak bandwidth requirements in our estimations. Also, the limited bandwidth for our cases leads to the largest latency reduction for the largest buffer size, as expected.

For these results it is possible to observe that *Hom_a* and *Het_a* achieve less latency than the baseline due to reduced off-chip accesses. The benefit for optimizing for reduced latency is clearer when comparing to optimizing for reduced accesses. As we see, *Hom_l* achieves less latency than *Hom_a* (up to 23% for *MobileNet* for 256kB) and *Het_l* achieves less latency than *Het_a* (up to 19% for *MobileNet* for 64kB).

Also as observed before in the access results, there are cases (*GoogLeNet* and *ResNet18*) for which the latency for the *Hom* and *Het* schemes is worse than what is achieved for the baseline. The reason for this, except of the bandwidth as mentioned before, is the padding in the ifmap, resulting in a slightly increase in the data footprint, which is not considered in the baseline.

Overall, these results show that using a memory management policy for a single common buffer with a goal to optimize the latency results in considerable gains (up to 56% for *MnasNet* for 1MB buffer). This happens due to the flexible management of the buffer space allowing to implement different policies and using the space for data prefetching in a selective way for different layers.

5.3 Optimization for accesses vs latency

In the previous sections we presented the best results among the different cases tested for the reduction of off-chip accesses and latency, respectively. These results were achieved with the proposed

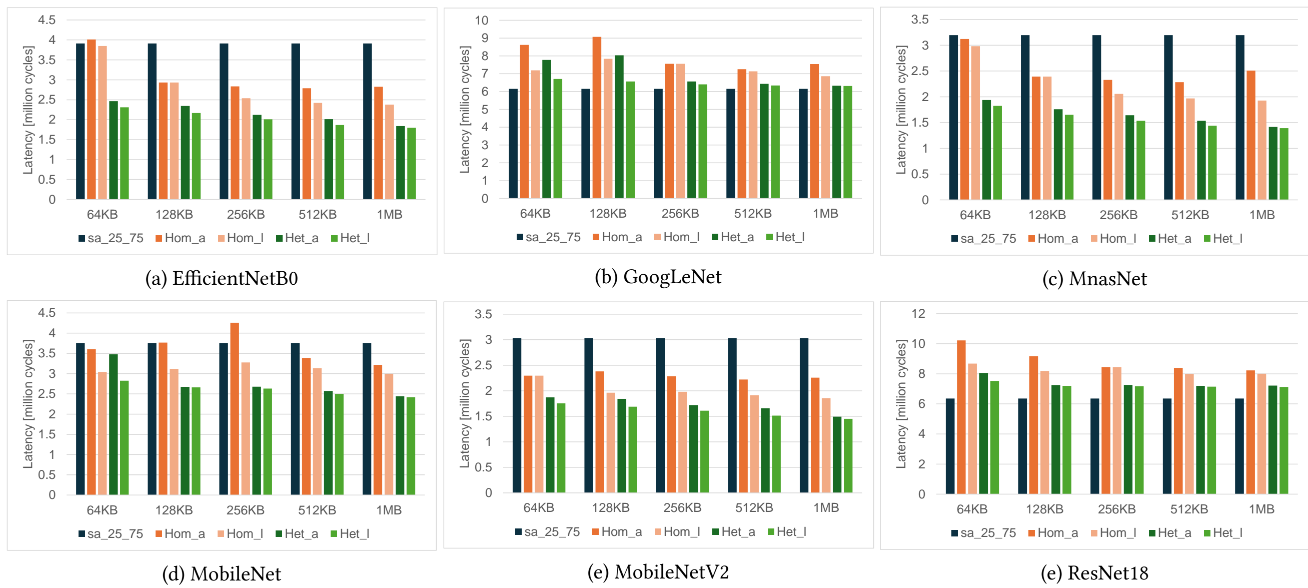


Figure 8: Latency for the baseline configurations and proposed schemes for different models and buffer sizes.

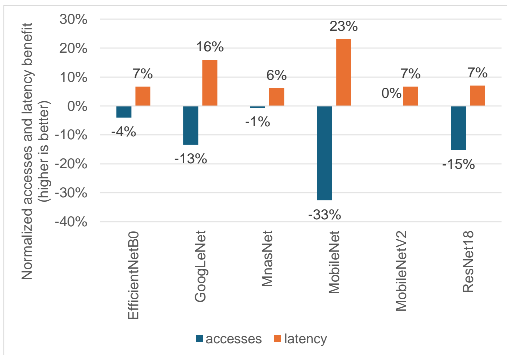


Figure 9: Comparison of the accesses and latency benefit for the heterogeneous scheme optimized for reduced latency when compared to a heterogeneous scheme optimized for reduced accesses. These results are for all models for a buffer size of 64kB.

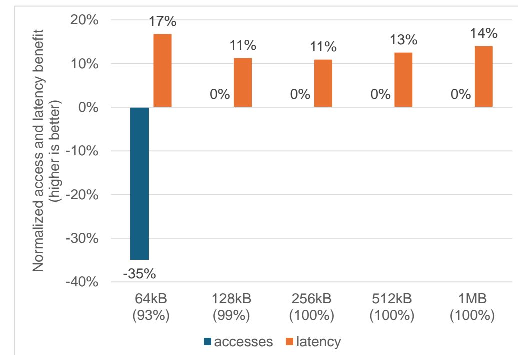


Figure 10: Comparison of the accesses and latency benefit for the heterogeneous scheme with prefetching enabled when compared to the heterogeneous scheme with disabled prefetching. The results are for MobileNet and all the buffer size configurations tested. In parenthesis we present the prefetching coverage for each configuration.

schemes but since in each case a different goal was selected, the policies chosen for the execution of the different layers are actually different. This means that effectively an execution plan for access reduction may be different than an execution plan for latency reduction. For both goals, the key aspect is the flexible management of the buffer space. Intuitively, if the goal is reduction of accesses, the space should be used to capture the data reuse while if the goal is reduction of the latency, the space should be used to prefetch data, thus overlapping the execution with the loading of data.

In this section we analyse the impact on the latency when a *Het* scheme optimized for latency is used as opposed of a *Het* scheme optimized for accesses. In Figure 9 we depict the benefit for the accesses and latency for using a *Het* scheme for latency reduction when compared to using a *Het* scheme for access reduction, for each model tested in this work. The benefit represents the change in the metric (accesses or latency) when using the *Het* scheme optimized for latency. A negative benefit represents a penalty.

The results in Figure 9 show, for example, that the highest benefit in terms of latency is achieved for *MobileNet* with 23% latency improvement when compared with using the scheme optimized for access reduction. This benefit though comes with a price, the penalty on the number of accesses by 33%. This is expected as a technique used to improve latency is prefetching and this may results in an increase in the number of accesses due to part of the space having been reserved for prefetching thus not being available to fully exploit the reuse.

To confirm that prefetching is indeed being applied in those cases, in Figure 10 we show the accesses and latency benefits for the *Het* scheme that is enabled to use prefetching when compared to the *Het* scheme that has prefetching disabled. We show this for a single model, in this case *MobileNet*, just to illustrate the point. The other models show similar results. In this figure we show the accesses and latency benefits for the different buffer sizes.

In these results we observe that latency benefit when using prefetching is close to 15% for most configurations. We also observe that for the smaller buffer size (64kB) the latency benefit comes with a penalty on the number of accesses by 35%. This clearly shows that for the smaller buffer sizes we have a tradeoff between access and latency when deciding on what to prioritize for the buffer: space for data reuse or space for prefetching. Larger buffer configurations do not suffer from this tradeoff as there is enough space to exploit the data reuse and thus any extra space may be effectively used for prefetching data. It is also relevant to note the high coverage achieved by the *Het* scheme with prefetching enabled. For 64kB, 93% of the layers use prefetching while for 256kB and larger buffer sizes, this coverage goes up to 100%.

5.4 Inter-layer reuse

As mentioned in Section 2, in *inter-layer reuse* the output of a layer is used as input to the next layer of the model. This is effectively the implementation of dataflow between two consecutive layers and translates to a considerable reduction in off-chip memory accesses. Nevertheless, in the case of layer-by-layer execution this also means that it can only be exploited if there is enough on-chip memory space to store the whole output of a layer as the execution of layer $i + 1$ only starts after the completion of the execution of layer i. With the proposed memory management, it is possible to capture the reuse with only as much buffer space as needed so the on-chip memory buffer is used more efficiently. Consequently, and depending on the output size of each layer, there are certain cases when it is possible to exploit the *inter-layer reuse*.

In order to evaluate the benefit of this optimization, in Figure 11 we depict the accesses and latency benefit for when we enable exploiting *inter-layer reuse* when compared to a baseline scheme with disabled *inter-layer reuse*.

The results show that while there are no visible benefits for the smaller buffer sizes, the benefits are considerable for the larger buffer sizes. For the smaller buffer sizes, the available space is barely enough to capture the required data reuse. As such, *inter-layer reuse* is rarely applied. In parentheses below the buffer size we present the coverage of the *inter-layer reuse*, i.e. how many times has *inter-layer reuse* been enabled when compared to all possible cases. For example, for 64kB the coverage was 0% so *inter-layer reuse* was never applied. The coverage grows to 4% for 128kB buffer, but high coverage only happens for the larger buffer sizes with 88% for the 512kB and 98% for the 1MB buffer sizes. As a consequence, for the larger buffer size of 1MB we observe the largest coverage (98%) and also the largest benefits for accesses, 70% reduction in the accesses when compared with not using *inter-layer reuse* at all, and for latency with 18% reduction. While we presented only the results for *MnasNet*, we have also observed similar behaviours for the other models. In particular the geometric mean of the performance

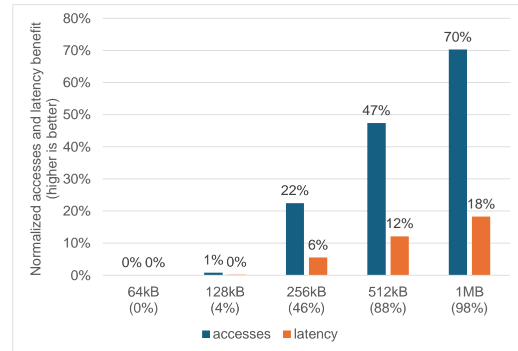


Figure 11: Comparison of the accesses and latency benefit for heterogeneous schemes with inter-layer reuse enabled when compared to it being disabled. This is shown for all on-chip buffer configurations for the MnasNet model.

benefit for 1MB buffer for all the models evaluated is 47% for the access reduction and 8% for the latency reduction.

Overall we can observe that exploiting the *inter-layer reuse* results in a significant improvement in the accesses and latency but this comes with a cost of requiring the use of considerably large on-chip memories.

6 CONCLUSIONS AND FUTURE WORK

In order to address the pressure on the DL hardware accelerator resources, in this work we propose a memory management technique for a unified global on-chip buffer. This buffer can store any of the different types of data to exploit reuse and at the same time allocate space for prefetching data. This solution selects among different data policies for different layers of the execution, utilizing the available buffer space in the best way for a specific goal. The goal can be either reduction of off-chip accesses or latency. The evaluation of our proposed memory management shows a reduction of up to 80% of the off-chip memory accesses for the smallest buffer size, or up to 56% of the latency for the largest buffer size, when compared to a baseline accelerator.

We are currently investigating how to integrate the memory management technique and policies into an open source DL compiler such as TVM as to offer these benefits to existing accelerators.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable feedback. This work has results from the VEDLIoT project, which received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 957197. This work was also partly supported by the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project and the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland.

REFERENCES

[1] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. 2014. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing* 22, 10 (2014), 1533–1545.

[2] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*. 73–78.

[3] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934* (2020).

[4] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. 2021. Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 159–172.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[6] Xiaoming Chen, Yinhe Han, and Yu Wang. 2020. Communication lower bound in convolution accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 529–541.

[7] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[8] Intel Corporation. 2024. *Quick overview of Intel's Neural Processing Unit (NPU)*. https://intel.github.io/intel-npu-acceleration-library/npu.html

[9] Saptarsi Das, Arnab Roy, Kiran Kolar Chandrasekharan, Ankur Deshwal, and Sehwan Lee. 2020. A systolic dataflow based accelerator for CNNs. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[10] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 92–104.

[11] Cristina Silvano et al. 2023. A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms. *CoRR* abs/2306.15552 (2023). https://doi.org/10.48550/ARXIV.2306.15552 arXiv:2306.15552

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[13] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[14] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[15] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[16] Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. 2016. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1646–1654.

[17] Hsiang-Tsung Kung. 1982. Why systolic architectures? *Computer* 15, 1 (1982), 37–46.

[18] Gang Li, Zejian Liu, Fanrong Li, and Jian Cheng. 2021. Block convolution: toward memory-efficient inference of large-scale CNNs on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 5 (2021), 1436–1447.

[19] Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, and Xiaowei Li. 2018. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 343–348.

[20] Peisong Li, Xinheng Wang, Kaizhu Huang, Yi Huang, Shancang Li, and Muddesar Iqbal. 2022. Multi-model running latency optimization in an edge computing paradigm. *Sensors* 22, 16 (2022), 6097.

[21] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. 2016. DaDianNao: A neural network supercomputer. *IEEE Trans. Comput.* 66, 1 (2016), 73–88.

[22] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st international conference on computer design (ICCD)*. IEEE, 13–19.

[23] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. 2023. Massively Parallel Neural Processing Array (MPNA): A CNN Accelerator for Embedded Systems. In *Embedded Machine Learning for Cyber-Physical, IoT, and Edge Computing: Hardware Architectures*. Springer, 3–24.

[24] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 58–68.

[25] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[26] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. 2018. FPGA-based accelerators of deep learning networks for learning and classification: A review. *ieee Access* 7 (2018), 7823–7859.

[27] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer. In *2017 IEEE 25Th annual international symposium on field-programmable custom computing machines (FCCM)*. IEEE, 93–100.

[28] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos. 2018. Memory requirements for convolutional neural network hardware accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 111–121.

[29] Mohammadreza Soltaniyeh, Richard P Martin, and Santosh Nagarakatte. 2022. An accelerator for sparse convolutional neural networks leveraging systolic general matrix-matrix multiplication. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 3 (2022), 1–26.

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[31] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2820–2828.

[32] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[33] Saurabh Tewari, Anshul Kumar, and Kolin Paul. 2020. Bus width aware off-chip memory access minimization for CNN accelerators. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 240–245.

[34] Shikhar Tuli, Chia-Hao Li, Ritvik Sharma, and Niraj K Jha. 2023. CODEBench: A neural architecture and hardware accelerator co-design framework. *ACM Transactions on Embedded Computing Systems* 22, 3 (2023), 1–30.

[35] Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. 2022. Comprehensive accelerator-dataflow co-design optimization for convolutional neural networks. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 325–335.

[36] Rui Xu, Sheng Ma, Yaohua Wang, Xinhai Chen, and Yang Guo. 2021. Configurable multi-directional systolic array architecture for convolutional neural networks. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–24.

[37] Rui Xu, Sheng Ma, Yaohua Wang, Yang Guo, Dongsheng Li, and Yuran Qiao. 2021. Heterogeneous systolic array architecture for compact cnns hardware accelerators. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2021), 2860–2871.

[38] Jiaqi Yang, Hao Zheng, and Ahmed Louri. 2022. Adapt-Flow: A Flexible DNN Accelerator Architecture for Heterogeneous Dataflow Implementation. In *Proceedings of the Great Lakes Symposium on VLSI 2022*. 287–292.

[39] Li Zhang, Qishen Lv, Di Gao, Xian Zhou, Wenchao Meng, Qinmin Yang, and Cheng Zhuo. 2023. A fine-grained mixed precision DNN accelerator using a two-stage big–little core RISC-V MCU. *Integration* 88 (2023), 241–248.

[40] Yong Zheng, Haigang Yang, Yi Shu, Yiping Jia, and Zhihong Huang. 2022. Optimizing off-chip memory access for deep neural network accelerator. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 4 (2022), 2316–2320.

[41] Xian Zhou, Li Zhang, Chuliang Guo, Xunzhao Yin, and Cheng Zhuo. 2020. A convolutional neural network accelerator architecture with fine-granular mixed precision configurability. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[42] Stavroula Zouzoula, Muhammad Waqar Azhar, and Pedro Trancoso. 2023. RAINBOW: Multi-Dimensional Hardware-Software Co-Design for DL Accelerator On-Chip Memory. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 352–354.