Received October 14, 2021, accepted November 5, 2021, date of publication December 10, 2021,
date of current version December 29, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3134930

ConvFusion: A Model for Layer Fusion in Convolutional Neural Networks

LUC WAEIJEN[1,2], SAVVAS SIOUTAS[1], MAURICE PEEMEN[3], MENNO LINDWER[1], AND HENK CORPORAAL[2]

[1]GrAI Matter Labs, 5656 AG Eindhoven, The Netherlands
[2]Electronic Systems Research Group, Department of Electrical Engineering, Eindhoven University of Technology, 5612 AP Eindhoven, The Netherlands
[3]Thermo Fisher Scientific, 5651 GG Eindhoven, The Netherlands

Corresponding author: Luc Waeijen (lwaeijen@tue.nl)

ABSTRACT The superior accuracy and appealing universality of convolutional neural networks (CNNs) as a generic algorithm for many classification tasks have made the design of energy efficient CNN accelerators an important topic in both academia and industry. Of particular interest in the design and use of CNN accelerators is the scheduling of the computational workload, which can have a major impact on the quality of the final design. The many inherently independent operations in CNNs result in a vast scheduling space however, rendering the selection of the optimal schedule(s) non-trivial. To aid in this complex task, this work introduces a generic mathematical cost model of the external memory accesses, internal memory footprint, and compute load for CNN execution schedules. The model enables fast exploration of the scheduling space, including loop tiling, loop reordering, explicit data transfer scheduling, recomputation, and, crucially, layer fusion, which recently has attracted interest as a method to reduce external memory accesses. An accompanying open source tool is released to perform schedule space exploration for CNNs using the introduced cost model. Leveraging the code generation capabilities of this tool the proposed model is validated on six real world networks, demonstrating that layer fusion can reduce the external memory accesses by more than two orders of magnitude compared to the best non-fused schedules. Confusing at first glance however, a high-level energy analysis shows that the practical benefits of layer fusion may be overestimated if other parts of the system are not tuned accordingly.

INDEX TERMS Energy efficiency, modeling, neural networks, scheduling.

I. INTRODUCTION

There is no longer any debate regarding the advantages of the class of convolutional neural network (CNN) algorithms. Many important problems, previously deemed difficult if not impossible to compute, are now being solved by CNNs. The plethora of application domains includes: control systems, pattern recognition, power systems, robotics, forecasting, manufacturing, art, and medical diagnosis [1].

Despite their successful application to many computational problems over the last decade, CNNs also have several major drawbacks. In particular, they are both compute and memory intensive algorithms. In the early years this kept the execution of CNNs confined to data centres, as evaluation on available general purpose, embedded processors required too much energy to be practical in mobile, energy constrained devices. To overcome this, many dedicated CNN accelerators have been proposed since to bring CNNs to the edge, and in general reduce CNN energy consumption [2]. In modern technology nodes the main challenge in achieving a high energy efficiency for such accelerators is not so much the compute complexity, but rather the required memory accesses. Compared to an ALU operation, accessing an SRAM requires about 5× the energy, and going to external DRAM about 200× [3]. This phenomenon, commonly referred to as the memory wall [4], will only aggravate with further technology scaling. To attain high energy efficiency it is therefore imperative that compute devices use their memory systems optimally.

Apart from techniques at the algorithmic level to reduce the total required memory accesses, minimizing the energy spent on the memory system constitutes of maximizing data reuse captured in small local memories. In essence this reduces the problem to finding a beneficial execution schedule for a given CNN. Due to the massive amount of independent operations

The associate editor coordinating the review of this manuscript and approving it for publication was Felix Albu.

in CNNs many valid schedules exist however, and finding the optimal schedule(s) for a given network and compute platform is exceedingly complex. In particular, the combination and parametrisation of scheduling techniques such as loop tiling [5], loop reordering [6], and more recently loop/layer fusion [7]–[10], results in a vast scheduling space. To deal with this vast space existing research typically restricts itself to a subset of the complete space. This leads to the selection of potentially suboptimal schedules, and prohibits the generic application of the obtained results across different compute platforms.

This work introduces a generic cost model which can efficiently compute the cost of a CNN schedule in terms of external memory accesses, required internal buffer space, and total multiply accumulates. The model is platform agnostic, and capable of handling CNN schedules that can be created using loop tiling, loop reordering, explicit scheduling of memory transfers, and layer fusion. This enables a fast search through possible schedules for CNNs without the need to perform profiling runs to obtain the cost of a particular schedule. The model is generic, and as such can be integrated in auto-schedulers for various accelerators and architectures by adequately bounding the schedule space based on specific architectural properties. A proof of concept, open source tool is developed building upon Keras/TensorFlow [11] as a front-end, and Halide [3] as a back-end. This tool is capable of performing exhaustive design space exploration for selected CNNs using the proposed model. Furthermore the Halide back-end generates code for each schedule, and instruments that code such that the modelled costs can be verified.

Various related works have demonstrated the potential of schedules that employ layer fusion to reduce external memory accesses [7]–[10]. However, the effect of these reductions on the net energy efficiency is typically overlooked. This work includes a high-level energy analysis based on the introduced schedule cost model, which results in some potentially surprising conclusions regarding the benefits of layer fusion.

The main contributions of this work are:

- Introduction of a platform agnostic, mathematical model of the cost of a CNN schedule in terms of memory accesses, memory footprint, and compute load, considering the vast scheduling space defined by loop interchange, loop tiling, loop fusion, recomputation, and explicit data transfer scheduling (Sections III & IV).
- An open source tool that implements the introduced model, enabling exhaustive design space exploration for CNNs [12] (Section V).
- Validation of the proposed models accompanied with detailed analysis of the effects of various scheduling techniques on six real-world networks (Section VI).
- Generic energy evaluation using the modelled schedule costs, which shows that the reduction in external memory accesses achieved by layer fusion does not automatically translate to significant net energy reduction (Section VII).

The remainder of this paper is organised as follows. First, the related work on CNN scheduling is discussed in Section II. Next, the scheduling space is formally defined in Section III. The cost model defined on this space is introduced in Section IV. Section V details the open source tool and experimental setup. Results on model validation and design space exploration are provided and analysed in Section VI. Section VII contains an energy evaluation of the discovered schedules for a platform with a multi-level memory hierarchy. Finally, Section VIII discusses current limitations and future work, and Section IX concludes this work.

II. RELATED WORK

Deep neural networks are both compute and memory intensive algorithms, and only have become viable methods by the merit of increased compute capacity about a decade ago. The recent renewed interest in deep neural networks was initiated by the successes in image classification of general purpose computing on graphical processing unit (GP-GPU) based implementations of CNNs [13], [14] circa 2011-2012. Since then tremendous effort has been made to enable the efficient execution of deep neural networks on energy-constrained (embedded) devices. Because the basic algorithm does not change much over different applications, CNNs in essence provide a universal solution to many compute tasks. This makes them a highly eligible target for dedicated hardware solutions, and as such has inspired the design of many CNN hardware accelerators [2].

Because CNNs have a large memory footprint, these accelerators typically require a form of external memory to store the network parameters and intermediate results. One of the first published accelerator designs to recognize the importance of minimizing accesses to this external memory was the Eyeriss by Chen *et al.* [5]. Based on manual analysis an execution schedule of CNNs is proposed for Eyeriss, which exploits spatial features of the architecture, and leverages strip-mining (a subset of tiling) to handle networks that do not match naturally with the dimensioning of the compute elements inside the accelerator. The iteration order over the CNN operations is fixed however, and tailored towards the architecture.

Rather than fixing the iteration order, SmartShuttle [6] takes more of the scheduling space into account by defining three iteration strategies, each targeting capturing reuse of different data. In particular, there are schedules that primarily optimise capturing parameter/weight reuse, input feature map data reuse, or output feature map reuse. Selecting between these schedules depends on the dimensions of the network layers. This choice already starts to outline the trilemma of selecting of which data elements to capture the reuse in local buffers, and the difficulty of finding the schedule that minimises the overall external memory accesses, as optimising for one subset of the data typically hurts the captured reuse of other parts.

To address the selection of a schedule based on tiling Peemen *et al.* [15] introduce generic formulas to enable fast schedule space exploration. The proposed formulas require manual tuning for different loop orders however, which is addressed by the model proposed by Waeijen *et al.* [16]. This model can generically compute the cost of schedules that include loop reordering, loop tiling, and explicit scheduling of data transfers. A very extensive framework that combines the same scheduling techniques with multi-level memory mapping is ZigZig [17]. Missing in these models and framework, however, is the capability to handle loop fusion, or layer fusion as first introduced by Alwani *et al.* [7].

Alwani *et al.* [7] introduce the concept of fusing the computation of two consecutive layers in order to avoid the transfer of intermediate results to external memory. In essence this is an on-demand computation of the intermediate results, which are immediately consumed by the next layer. Because of overlap in tiles, as the authors note, there is the option to recompute intermediate results of which not all uses fall within a single tile, or to store this subset of intermediate results. Eventually only schedules without recompute are considered however. The layer fusion proposed by Alwani *et al.* [7] suffers from another shortcoming, aptly dubbed the computation pyramid by the authors. This term refers to the phenomenon that when more layers are fused, the number of points a tile in the output layer depends on expands rapidly, creating a pyramid of dependencies towards the input of the network. The AivoTTA accelerator [18] exhibits this very same imperfection.

This particular issue is addressed by Li *et al.* [8] by modifying the CNN algorithm and removing several dependencies between layers, while maintaining acceptable accuracy. An arguably preferable solution which does not require changes to the CNN algorithm is proposed by Goetschalckx and Verhelst [10], who employ line buffers to fuse layers, or execute depth-first in their terminology. This solution is more attractive as the functionality of the network remains unchanged, while large gains can be achieved in particular for networks with large layer dimensions. However, the proposed approach prohibits tiling in the channel dimension, and requires to always store the entire weight set on-chip, which may be highly suboptimal for networks with relatively small layer dimensions, i.e., layers of which the memory footprint is not dominated by data, but by weights. Despite these shortcomings, the depth-first methodology of Goetschalckx and Verhelst [10] sets the standard for schedules with layer fusion, and will be used as an important benchmark throughout this work.

Finally, several recent works have included layer fusion in GPU code generation [19], [20]. These works clearly show the potential gains of layer fusion, although both rely on heuristics to find good schedules, and require profiling runs on the target hardware. The models proposed in this work can be adapted towards SIMD and GPU execution as is further discussed in Section VIII, and can give insight into the performance of a schedule without execution on a target machine. Consequently they may be used to speed-up and expand the design space searches of such heuristic (GPU) auto-schedulers.

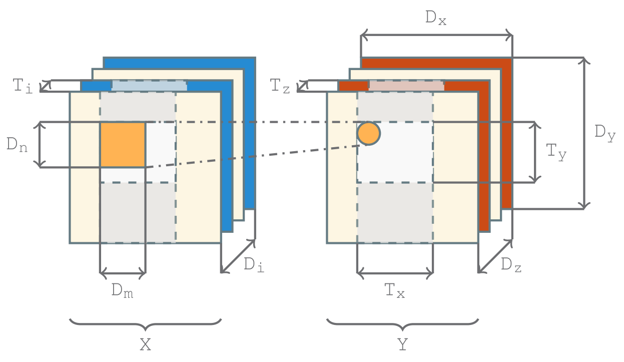


FIGURE 1. Single convolutional layer with input array $\mathtt{X}$ of and output array $\mathtt{Y}$. Dimensional notation shown here is used throughout this work, where $\mathtt{D_n}$ and $\mathtt{D_m}$ denote the kernel height and width, $\mathtt{D_i}$ and $\mathtt{D_z}$ the number of feature maps in the input and output array, and finally $\mathtt{D_x}$ and $\mathtt{D_y}$ the width and height of the output array, respectively. Furthermore, $\mathtt{T_i}$, $\mathtt{T_z}$, $\mathtt{T_y}$ and $\mathtt{T_x}$ denote a tile size in the input feature maps, output feature maps, and output array height and width, respectively.

TABLE 1. Set of structural parameters of a convolution layer.

Parameter	Description
$\mathtt{D_x}$	Width of output ($\mathtt{Y}$) feature maps
$\mathtt{D_y}$	Height of output ($\mathtt{Y}$) feature maps
$\mathtt{D_z}$	Number of output ($\mathtt{Y}$) feature maps
$\mathtt{D_i}$	Number of input ($\mathtt{X}$) feature maps
$\mathtt{D_m}$	Convolutional kernel width
$\mathtt{D_n}$	Convolutional kernel height

III. SCHEDULING SPACE

To facilitate the definition of the cost models later in Section IV, first the covered scheduling space is formally defined in this section. This definition starts with a high-level description of CNNs, followed by detailed descriptions of the considered scheduling techniques.

In a nutshell, a convolutional neural network functionally consists of a series of parallel, convolutional filters, or layers, connected by non-linear activation functions. The weights of these filters are determined during a learning phase in such a way that the network can perform its intended classification task. Once a suitable set of weights has been established, it remains static throughout the classification, or inference, phase, which is the focus of this work. Readers left desiring a more detailed description can find an excellent in-depth introduction to convolutional neural networks in the 'Deep Learning' book by Goodfellow *et al.* [21].

A single convolution layer consists of a set of convolution filters which are to be applied to a set of input surfaces to produce a set of output surfaces. These surfaces are referred to as feature maps. In the general case, a filter is applied for each pair of input and output feature maps, which is also the type of convolutional layer considered in the remainder of this work. More advanced layer types, such as depthwise convolution, are not directly considered, although possible model extensions are discussed in Section VIII-C.

Structurally a standard convolution layer is completely defined by the set of parameters listed in Table 1. Figure 1 is a visual representation of such a layer, and its various dimensions. Convolutions are applied to the source feature maps on the left ($\mathtt{X}$), and their results, after application of a non-linear transformation, are aggregated in the feature maps on the right ($\mathtt{Y}$). From an implementation viewpoint, a convolutional layer is a deep loop nest. The pseudocode of a single layer is shown in Code 1. Here variables $\mathtt{Sx}$ and $\mathtt{Sy}$ represent the stride of the filter on the input, which typically is one. A complete neural network consists of several of these layers connected through their feature maps. As such, a neural network can represented as a directional graph $G(V, E)$ with the network layers V as nodes, and directional edges E to indicate their producer — consumer relationships.

```
1  for(int z=0; z<Dz; z++)
2   for(int y=0; y<Dy; y++)
3    for(int x=0; x<Dx; x++){
4     Y[z][y][x]=bias[z];
5     for(int i=0; i<Di; i++)
6      for(int n=0; n<Dn; n++)
7       for(int m=0; m<Dm; m++)
8        Y[z][y][x]+= \
9         X[i][y*Sy+n][x*Sx+m] \
10        * W[z][i][n][m];
11    Y[z][y][x]=act(Y[z][y][x]);
12   }
```

Code 1. Loopnest for a single convolution layer.

From Code 1 it can be seen that the multiply-accumulate operations (lines $8-10$) within a layer are completely independent. As such they may be executed in any order, yielding $\left(\mathtt{D_z} \times \mathtt{D_i} \times \mathtt{D_y} \times \mathtt{D_x} \times \mathtt{D_m} \times \mathtt{D_n}\right)!$ scheduling options, ignoring the bias initialisation and the application of the activation function which even further increase the scheduling space. Reordering these operations will result in different reuse-distance distributions for the input data elements $\mathtt{X}$, output data elements $\mathtt{Y}$, and weights $\mathtt{W}$. A smart reordering will capture more data-reuse in an internal buffer of given size, and as such minimize the accesses to an external memory. However, many of these schedules are highly irregular, and impossible to capture within reasonable code size. Therefore, this work only considers those schedules that can be generated using loop reordering and loop tiling at the layer level, as will be further discussed in Sections III-A, and III-B respectively.

Apart from scheduling the compute operations, the data transfers between memory levels can also be explicitly scheduled. For accelerators that typically use scratchpad memory such scheduling is imperative, but machines using caches can also benefit from grouping data transfers. Explicitly scheduling these transfers consists of specifying what data will be stored and reused, and when this data is loaded from the external memory. To specify this, the store level and compute level concepts of the Halide language [3] are used, and made part of the considered scheduling space as described in Section III-C. Apart from scheduling the data transfers these concepts also allow for a precise expression of recomputation of intermediate results; a scheduling technique which provides a trade-off between (external) memory accesses and compute workload, as described in Section III-E.

Finally the scheduling space is expanded beyond scheduling individual layers by allowing layer fusion. Assume two convolutional layers A and B, which are connected in a network in such a way that B consumes the output of A. Following the dependencies, it is clear that some operations in B can already be executed, even if not all operations that belong to A are completed. Therefore it is possible to move (part of) the production of layer A into the loop nest of layer B using loop fusion. In this manner the results of layer A can potentially be consumed and discarded by B shortly after their production, effectively reducing their lifetime. Compared to an approach without layer fusion this has the potential to significantly reduce the accesses to a large external memory. The technique can furthermore be applied recursively, allowing any number of consecutive convolutional layers to be fused. Details on how this affects the overall scheduling space are provided in Section III-D.

A. LOOP REORDERING

To formally define the loop order of a schedule, let $\mathtt{L}$ denote the set of all loop variables in a convolutional layer. In accordance to Code 1, $\mathtt{L} = \{\mathtt{z}, \mathtt{y}, \mathtt{x}, \mathtt{i}, \mathtt{m}, \mathtt{n}\}$. The loop order $O \subseteq \mathtt{L} \times \mathtt{L}$ defines a set of binary relations over $\mathtt{L}$, where, with $\mathtt{l}, \mathtt{l'} \in \mathtt{L}$, $\mathtt{l} \prec \mathtt{l'}$ yields true iff $\mathtt{l}$ is inner to $\mathtt{l'}$ in the loop nest, resulting in a total ordering of $\mathtt{L}$. In Code 1, the following expression holds: $\mathtt{m} \prec \mathtt{n} \prec \mathtt{i} \prec \mathtt{x} \prec \mathtt{y} \prec \mathtt{z}$.

With this definition of loop order in place, consider the reuse distance of data elements in the $\mathtt{X}$ and $\mathtt{Y}$ arrays. Note that the accesses to array $\mathtt{Y}$ are independent of loop variable $\mathtt{i}$ (Line 8 in Code 1), while those to array $\mathtt{X}$ (Line 9 in Code 1) are dependant on $\mathtt{i}$. Henceforth, because $\mathtt{i} \prec \mathtt{z}$ in Code 1, the accumulations to a single $\mathtt{z}$-coordinate in the $\mathtt{Y}$ array on line 8 are relatively close in time. However, for each of these accumulations an element from a unique $\mathtt{i}$ index has to be loaded from the $\mathtt{X}$ array on line 9. Therefore the reuse distances on array $\mathtt{X}$ are relatively long, while those on array $\mathtt{Y}$ are relatively short. Yet, when loops $\mathtt{i}$ and $\mathtt{z}$ are interchanged, i.e., $\mathtt{z} \prec \mathtt{i}$, the reverse holds. Which of these orders is favourable depends, amongst other factors, on the particular dimensioning of the layer. To complicate matters further, the other loops can also be reordered, and the data reuse of array $\mathtt{W}$ is also significant, rendering a complex trade-off. Nonetheless, it can be stated that moving kernel loops $\mathtt{m}$ and $\mathtt{n}$ will likely not be beneficial, as typically $\mathtt{D_m}$ and $\mathtt{D_n}$ are very small (common values encountered in practice include one, three, and five). As such, loop reordering in this work is restricted to the remaining loop levels.

B. LOOP TILING

Tiling is a classic scheduling technique to alter the execution order of operations. As discussed in the previous section, a particular loop order may decrease the reuse on one data array, but increase it in another one. A different loop order may achieve the reverse. Loop tiling enables a hybrid approach, allowing a balanced average reuse distance for all data accesses. By splitting a loop $\mathtt{l} \in \mathtt{L}$ that iterates over a complete dimensions into an inner part $\mathtt{li}$, and outer part $\mathtt{lo}$, it is possible to only compute part of a dimension inner to the iteration over another dimension.

For CNN layers in particular, each loop in Code 1 can be split. Again, because the kernel dimension $\mathtt{D_m}$ and $\mathtt{D_n}$ are typically very small, tiling loops $\mathtt{m}$ and $\mathtt{n}$ are not considered. However, the remaining loops, i.e., $\{\mathtt{i}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$, can all be tiled into parts of size $\mathtt{T_l}$, where $\mathtt{l} \in \{\mathtt{i}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Since $\mathtt{T_l}$ can be set to one, it is possible to rewrite Code 1 into Code 2 without loss of generality.

```
1   //outer tile loops
2   for(int zo=0; zo<Dz; zo+=Tz)
3    for(int yo=0; yo<Dy; yo+=Ty)
4     for(int xo=0; xo<Dx; xo+=Tx)
5       for(int io=0; io<Di; io+=Ti)
6       //inner tile loops
7       for(int zi=zo; zi<zo+Tz; zi++)
8        for(int yi=yo; yi<yo+Ty; yi++)
9         for(int xi=xo; xi<xo+Tx; xi++){
10         if(io==0)
11          Y[zi][yi][xi]=bias[zi];
12         for(int ii=io; ii<io+Ti; ii++)
13          for(int n=0; l<Dn; n++)
14           for(int m=0; m<Dm; m++)
15            Y[zi][yi][xi]+= \
16               X[ii][yi*Sy+n][xi*Sx+m] \
17               * W[ii][zi][m][n];
18         if(io+Ti>=Di)
19          Y[zi][yi][xi]= \
20           act(Y[zi][yi][xi]);
21        }
```

Code 2. Tiled loopnest for a single convolution layer.

For the remainder of this work, Code 2 will be used to define schedules of a single layer. As such, a tiled schedule formally consists of an ordering O on the set of tiled loop levels $\mathtt{TL} = \{\mathtt{zo}, \mathtt{yo}, \mathtt{xo}, \mathtt{io}, \mathtt{zi}, \mathtt{yi}, \mathtt{xi}, \mathtt{ii}, \mathtt{n}, \mathtt{m}\}$, and a set of tile sizes $\mathtt{T} = \{\mathtt{T_z}, \mathtt{T_y}, \mathtt{T_x}, \mathtt{T_i}\}$.

C. STORE & COMPUTE LEVELS

Besides the computations, transfers between external memory and local buffers can be scheduled explicitly as well. To capture these memory operations the store and compute level concepts from the Halide language [3] are employed. These levels are defined for each array $\mathtt{X}$, $\mathtt{Y}$, and $\mathtt{W}$, and dictate respectively what data volume is transferred when.

Let $\mathtt{ARR} = \{\mathtt{X}, \mathtt{Y}, \mathtt{W}\}$ denote the set of all data arrays in a layer. The store level is then defined as follows:

> The **store level** $SL_{\mathtt{arr}} \in \mathtt{TL}$ for array $\mathtt{arr} \in \mathtt{ARR}$ determines that at all data accesses to elements in $\mathtt{X}$ inside a single iteration of loop $SL_{\mathtt{arr}}$ have to be served from local memory after an initial load.

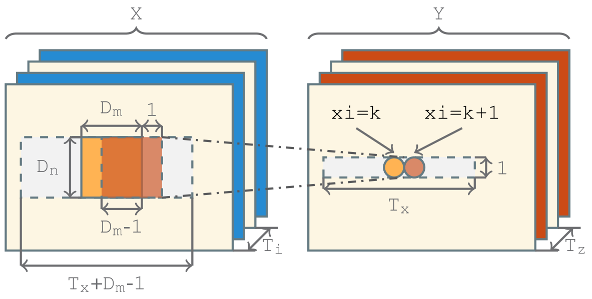


FIGURE 2. Overlap of ($\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{D_m} - 1)$) elements in the input data $\mathtt{X}$ between iterations 'n' and 'n+1' of loop $\mathtt{xi}$ in Code 2 N. B. The volume required for $SL_{\mathtt{X}} = \mathtt{yi}$ is ($\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{T_x} + \mathtt{D_m} - 1)$), yet a rotating buffer of size ($\mathtt{T_i} \times \mathtt{D_n} \times \mathtt{D_m}$) is sufficient.

In Code 2, for example, if $SL_{\mathtt{X}}$ is set to loop level $\mathtt{xi}$ on line 9, the data required for the ($\mathtt{T_i} \times \mathtt{D_n} \times \mathtt{D_m}$) operations inside one iteration of $\mathtt{xi}$ need to be served from local memory. Note that there is also data reuse of elements in $\mathtt{X}$ between two iterations of loop $\mathtt{xi}$, as illustrated in Figure 2. In particular, there is an overlap of ($\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{D_m} - 1)$) elements between two consecutive iterations.[1] The store level does not specify to capture this reuse in a local memory. To capture this reuse, the store level has to be moved one loop level up, to $\mathtt{yi}$. Since one iteration of loop $\mathtt{yi}$ encapsulates $\mathtt{T_x}$ iterations of loop $\mathtt{xi}$, the reuse between these iterations must now be captured by the internal buffer as well. As a consequence the volume of data that needs to be captured increases from ($\mathtt{T_i} \times \mathtt{D_m} \times \mathtt{D_n}$) to ($\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{T_x} + \mathtt{D_m} - 1)$), as visualised in Figure 2. Note that $\mathtt{T_x}$ is defined on the output layer, and because of the kernel size $\mathtt{D_m}$ a tile of size ($\mathtt{T_x} + \mathtt{D_m} - 1$) is thus required of the input layer. This demonstrates the trade-off between required on-chip buffer size and number of external memory accesses that can be explored using the store level.

Apart from the store level $SL_{\mathtt{arr}} \in \mathtt{TL}$, also a compute level is defined.

> The **compute level** $CL_{\mathtt{arr}} \in \mathtt{TL}$ determines at what loop iteration new data is produced/loaded for each array $\mathtt{arr} \in \mathtt{ARR}$.

This additional directive enables an optimization known as buffer folding. For $SL_{\mathtt{X}} = \mathtt{yi}$ the data volume that has to be delivered by on-chip memory is equal to ($\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{T_x} + \mathtt{D_m} - 1)$) elements[1], but that does not require that this data is all live at the same time. In fact, as the ($\mathtt{T_i} \times \mathtt{D_m} \times \mathtt{D_n}$) kernel moves from left to right as the $\mathtt{xi}$ loop proceeds, old data to the left will no longer be reused within the current iteration of loop $\mathtt{yi}$. By selecting $\mathtt{xi}$ as the compute level of array $\mathtt{X}$, i.e., $CL_{\mathtt{X}} = \mathtt{xi}$, new data is only produced, i.e., fetched from external memory, at each iteration of $\mathtt{xi}$. Since there are ($\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{D_m} - 1)$) elements overlap between each iteration, as discussed before, for each iteration only $\mathtt{T_i} \times \mathtt{D_n}$ new elements are required.

[1]Assuming stride $\mathtt{Sx} = 1$ for simplicity.

These can be kept in a rotating buffer of only $(\mathtt{T_i} \times \mathtt{D_m} \times \mathtt{D_n})$ elements. Note that the reuse captured by the internal buffer is unchanged, and dictated only by the store level. The addition of the compute level enables folding of buffers, such that the same reuse can be captured with less buffer space. Combined, the store and compute levels respectively dictate what data is transferred when.

Unlike loop ordering and tiling, the store and compute levels can not be chosen freely. In particular, data dependencies dictate that the production of the weights and input must be scheduled before, or in parallel with, the production of the feature maps, i.e., $SL_{\mathtt{Y}} \preceq SL_{\mathtt{W}}$ and $SL_{\mathtt{Y}} \preceq SL_{\mathtt{X}}$. Furthermore, the store level is always to be selected from one of the inner loops, or one level higher, i.e., any loop in Code 2 between lines 5–12. Setting the store level any higher would encompass at least one outer and inner loop of the same dimension, cancelling the effect of tiling. The same can be achieved by equating the tile size to the dimension, and as such these schedules are covered without the need to consider the remaining outer loop levels.

Finally the compute level of a data array should always be equal to, or lower than the store level of that array, i.e., $CL_{\mathtt{arr}} \preceq SL_{\mathtt{arr}}$. This requirement originates from the trivial dependency between the production of an element and the allocation of its storage. If no storage is allocated, the element can not be produced.

D. LAYER FUSION

Apart from reordering computations within a layer, as performed by loop reordering and tiling, there is also the possibility to reorder operations between layers. In particular, if one layer is computed partially, some of the computations of the succeeding layer may already have all their input operands ready, enabling their execution. This concept is best described in terms of producers and consumers, where a first layer produces data which is consumed by a second layer. Rather than computing the producer completely before starting the computation of the consumer, the computation of the producer can be inlined to the computation of the consumer. Again, these transformations alter reuse distances and lifetimes of the various data arrays. Critically, the results from the producer can be consumed much earlier. Unless there is already sufficient on-chip memory to buffer an entire layer, loop fusion can be used to consume the results of intermediate layers, rather then sending them out to external memory only to be retrieved again later.

The data dependencies between two convolutional layers are illustrated in Figure 3. Note that for a layer v fused into a layer u, the output array $\mathtt{Y}$ of layer v is the same as the input array $\mathtt{X}$ of u. Generically, $\mathtt{Y}_k = \mathtt{X}_{k+1}$ Therefore, in Figure 3, the $\mathtt{Y}$ arrays have been named by their $\mathtt{X}$ array equivalent. In this figure a tile of size $\mathtt{T_z} \times \mathtt{T_x} \times \mathtt{T_y}$ is to be produced in array $\mathtt{X2}$. Assume $\mathtt{X1}$ is not yet computed. When the production of $\mathtt{T_z} \times \mathtt{T_x} \times \mathtt{T_y}$ is about to start, first a tile in $\mathtt{X1}$ of size $\mathtt{T_i} \times (\mathtt{T_x} + \mathtt{D_{m1}} - 1) \times (\mathtt{T_y} + \mathtt{D_{n1}} - 1)$ is produced.

```
//outer tile loops of X1->X2
for(int zo=0; zo<Dz; zo+=Tz)
 for(int yo=0; yo<Dy; yo+=Ty)
  for(int xo=0; xo<Dx; xo+=Tx)
   for(int io=0; io<Di; io+=Ti){

    //Inline production of X0->X1
    for(int z=0; z<Ti; z++)
     for(int y=0; y<Ty+Dn1-1; y++)
      for(int x=0; x<Tx+Dm1-1; x++){
       X1[z][y][x]=bias[z];
       for(int i=0; i<Di0; i++)
        for(int n=0; n<Dn0; n++)
         for(int m=0; k<Dm0; m++)
          X1[z][y][x]+= \
           X0[i][y*Sy0+n][x*Sx0+m] \
           * W01[z][i][n][m];
       X1[z][y][x]=act(X1[z][y][x]);
      }

    //inner tile loops of X1->X2
    for(int zi=zo; zi<zo+Tz; zi++)
     for(int yi=yo; yi<yo+Ty; yi++)
      for(int xi=xo; xi<xo+Tx; xi++){
       if(io==0)
        X2[zi][yi][xi]=bias[zi];
       for(int ii=io; ii<io+Ti; ii++)
        for(int n=0; n<Dn1; n++)
         for(int m=0; k<Dm1; m++)
          X2[zi][yi][xi]+= \
           X1[ii][yi*Sy1+n][xi*Sx1+m]\
           * W12[ii][zi][m][n];
       if(io+Ti>=Di)
        X2[zi][yi][xi]= \
         act(X2[zi][yi][xi]);
      }
   }
```

Code 3. Code for 2 fused layers as illustrated in Figure 3.

Once this tile is ready, the computation of $\mathtt{T_z} \times \mathtt{T_x} \times \mathtt{T_y}$ in $\mathtt{X1}$ commences.

The basic code of two fused layers is given in Code 3. As can be seen, the production of a $\mathtt{T_i} \times (\mathtt{T_x} + \mathtt{D_{m1}} - 1) \times (\mathtt{T_y} + \mathtt{D_{n1}} - 1)$ sized tile $\mathtt{X1}$ is inlined in the loop nest of $\mathtt{X2}$. This technique is generically known as loop fusion. Since in this particular context it is applied to loop nests of CNN layers the term layer fusion is used.

Although not shown for simplicity in Code 3, it is entirely possible to also tile and reorder the production of the inlined producer. From this perspective, tiling and reordering are orthogonal concepts to layer fusion. Furthermore, layer fusion can be applied recursively, fusing an unlimited number of consecutive layers. This increases the scheduling space tremendously, complicating the task of finding an optimal schedule for a given network.

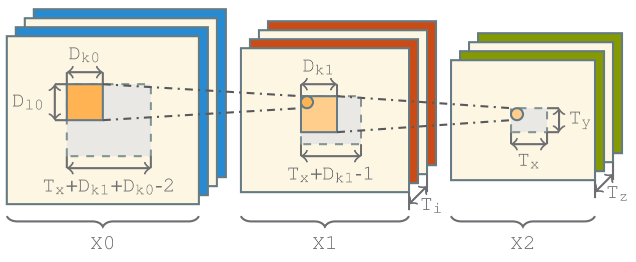


FIGURE 3. Three state arrays X of two consecutive convolutional layers. To produce tile $T_z \times T_x \times T_y$ on X2, a tile of $T_i \times (T_x + D_m - 1) \times (T_y + D_n - 1)$ is required from array X1. In a fused schedule, this tile of X1 is produced in-line to the production of the tile in X2, rather than first computing X1 completely.

The connections between layers in a neural network can generically be captured in a directed graph $G(V, E)$ where V represents the set of individual layers and their associated structural parameters as listed in table 1, and E is a set of tuples (src, dst) with src, $dst \in V$ that define a directional relation from src to dst. The production of a layer may be fused into one of its direct successors in this network graph G or it may not be fused at all. To denote this, each layer $v \in V$ is assigned a fuse target $Fuse(v) \in successors(v) \cup \{v\}$, where $successors(v) = \{v' \mid (v, v') \in E\}$ is the set of all direct successors of layer v in $G(V, E)$. The production of layer v is then scheduled inline into the production of $Fuse(v)$. When the fuse target is set to layer v itself, the layer is consequently not fused.

E. RECOMPUTATION

Apart from shortened data lifetimes, layer fusion also introduces another interesting trade-off. As discussed, depending on tiling and store levels not all data reuse may be captured from a local buffer. The same naturally holds for the data of an intermediate, fused layer. In Figure 3 the data of X1 has multiple uses in the production of X2. If not all uses of an element are captured, there is the option to store the intermediate value of X1 in external memory and reload it for future uses. Alternatively it can be discarded, and recomputed from X0 when it is needed again. In this way a trade-off can be made between compute load and external memory traffic. This is particularly interesting for modern and future technology nodes, where (re)compute typically can be orders of magnitude cheaper in both time and energy than re-accessing external memory [3].

F. FORMAL SCHEDULE

As stated, a convolutional layer is structurally defined by the set of parameters listed in table 1. For each layer $v \in V$, where V represents the complete set of layers that make up a particular CNN, a layer schedule s can be defined according to the various scheduling options discussed in this section. Such a scheduled layer s consists of the parameters listed in Table 2. A network schedule $S = \{(v, s) \mid v \in V\}$ is consequently defined as the set of tuples of layers and accompanying layer schedules for each layer in the network.

TABLE 2. Layer schedule s of a convolution layer $v \in V$.

Parameter	Description
$O \subseteq \mathtt{TL} \times \mathtt{TL}^{*}$	Loop ordering
$\mathtt{T_z}, \mathtt{T_y}, \mathtt{T_x}, \mathtt{T_i} \leq \mathtt{D_z}, \mathtt{D_y}, \mathtt{D_x}, \mathtt{D_i}$	Tile sizing
$SL_{\mathtt{X}}, SL_{\mathtt{Y}}, SL_{\mathtt{W}} \in \mathtt{TL}$	Store levels
$CL_{\mathtt{X}}, CL_{\mathtt{Y}}, CL_{\mathtt{W}} \in \mathtt{TL}$	Compute levels
$Fuse \in successors(v) \cup \{v\}^{**}$	Fuse target

[*] Recall, $\mathtt{TL} = \{\mathtt{zo}, \mathtt{zi}, \mathtt{yo}, \mathtt{yi}, \mathtt{xo}, \mathtt{xi}, \mathtt{io}, \mathtt{ii}\}$ denotes the set of all tiled loop levels in a convolution layer.
[**] Note, when $Fuse = v$ the layer is not fused.

TABLE 3. Model summary.

Function	Description
$\mathrm{Buf}_{\mathtt{W}}, \mathrm{Buf}_{\mathtt{X}}, \mathrm{Buf}_{\mathtt{Y}}$	Buffer sizes of the weight, input, and output arrays respectively.
$\mathrm{Acc}_{\mathtt{W}}, \mathrm{Acc}_{\mathtt{X}}, \mathrm{Acc}_{\mathtt{Y}}$	Number of weight, input and output elements respectively transferred from/to external memory.
MACS	Total number of multiply-accumulate operations.

IV. COST MODELS

For real-world neural networks, merely iterating through the entire scheduling space as described in Section III already presents a significant task. Benchmarking each of these schedules on a target machine to find the best match is simply intractable. This section describes a set of mathematical expressions which, given a network schedule, accurately model the required number of external memory accesses, the required internal buffer space, and the number of computations measured in multiply-accumulates, as summarized in Table 3. These expressions only require a handful of computations compared to benchmarking a network schedule on a target machine, and as such enable fast design space exploration. The remainder of this section defines these expressions precisely. Readers primarily interested in applying these models may skip ahead to Section V which introduces the open source implementation of these equations in the form of the `ConvFuser` tool [12]. Also the final results in Section VI can be interpreted without in-depth understanding of the detailed model presented in this section.

A. PREREQUISITES

To aid the formulation of these models, a number of notational shorthands and auxiliary functions are defined first. In general, the multiplication of each element in an arbitrary set S will be abbreviated to $\prod S$, i.e.,

$$\prod S = \prod_{s \in S} s.$$

Note that in accordance with the common definition of the product operator, the product of the empty set $\emptyset$ is defined as one.

Given a layer and an associated schedule $(v, s) \in S$, the (sub)set of structural dimensions of layer $v \in V$ as defined in table 1, and the (sub)set of tile sizes in schedule s as defined in table 2, that belong to a given (sub)set of loop levels $\mathtt{L}' \subseteq \mathtt{L}$, is defined by the following two auxiliary functions respectively:

$$D(v, \mathtt{L}') = \{\mathtt{D_l} \mid \mathtt{l} \in \mathtt{L}' \wedge \mathtt{D_l} \in v\},$$
$$T(s, \mathtt{L}') = \{\mathtt{T_l} \mid \mathtt{l} \in \mathtt{L}' \wedge \mathtt{T_l} \in s\}.$$

Furthermore a translation function $\kappa(v, \mathtt{l})$ is defined, with layer $v \in V$ and loop level $\mathtt{l} \in \{\mathtt{x}, \mathtt{y}\}$. This function converts loop levels $\mathtt{x}$ and $\mathtt{y}$ to their spatially related kernel dimensions $\mathtt{D_m}$ and $\mathtt{D_n}$ respectively:

$$\kappa(v, \mathtt{l}) = \begin{cases} \mathtt{D_m} \in v, & \mathtt{l} = \mathtt{x} \\ \mathtt{D_n} \in v, & \mathtt{l} = \mathtt{y}. \end{cases}$$

The corresponding set operator K, which translates all loop levels in a set $\mathtt{L}' \subseteq \mathtt{L}$, is defined as:

$$K(v, \mathtt{L}') = \{\kappa(v, \mathtt{l}) \mid \mathtt{l} \in \mathtt{L}'\}.$$

Another helper function translates a loop level $\mathtt{l} \in \mathtt{L}$ into the corresponding inner tiled loop level $\mathtt{li} \in \mathtt{TL}$:

$$inner(\mathtt{l}) = \mathtt{li}.$$

Since for many models it matters whether or not the inner loop of a particular loop level $\mathtt{l} \in \mathtt{L}$ is preceded by the store level in a given layer schedule s, the set of all loop levels in set $\mathtt{L}' \subseteq \mathtt{L}$ which are preceded by the store level $SL_{\mathtt{arr}}$ of array $\mathtt{arr} \in \mathtt{ARR}$, i.e., the collection of loop levels below/inner to the store level for array $\mathtt{arr}$, is defined as:

$$LT_{\mathtt{arr}}(s, \mathtt{L}') = \{\mathtt{l} \mid \mathtt{l} \in \mathtt{L}' \wedge inner(\mathtt{l}) \prec SL_{\mathtt{arr}}\},$$

where $SL_{\mathtt{arr}} \in \mathtt{TL} \in s$, $\prec \in O$, and $O \in s$.

The complement of this set, i.e., the set of loop levels which are equal to or above/outer to the store level, is defined as:

$$GE_{\mathtt{arr}}(s, \mathtt{L}') = \mathtt{L}' - LT_{\mathtt{arr}}(s, \mathtt{L}').$$

Furthermore the set of folded loop levels F, i.e., the levels between the store and compute level is defined as:

$$F_{\mathtt{arr}}(s, \mathtt{L}') = \{\mathtt{l} \mid \mathtt{l} \in \mathtt{L}' \wedge CL_{\mathtt{arr}} \preceq inner(\mathtt{l}) \prec SL_{\mathtt{arr}}\},$$

where $SL_{\mathtt{arr}}, CL_{\mathtt{arr}} \in \mathtt{TL} \in s$, $\prec \in O$, $O \in s$, and the operator $(\mathtt{l} \preceq \mathtt{l}') = (\mathtt{l} \prec \mathtt{l}' \vee \mathtt{l} = \mathtt{l}')$. Finally a set selection function is defined, which selects set A if the layer is fused, or set B otherwise.

$$FuseSel(v, s, A, B) = \begin{cases} A, & fuse \neq v \\ B, & \text{o.w.,} \end{cases}$$

with fuse target $fuse \in s$.

For all these helper functions, when it is clear only a single layer v or schedule s is described, the v and s arguments are omitted for further brevity. An overview of these helper functions is provided in Table 4.

TABLE 4. Helper functions summary.

Function	Description
$D(v, \mathtt{L}')$	Set of loop dimensions $\mathtt{D_l}$ belonging to loop levels $\mathtt{l} \in \mathtt{L}' \subseteq \mathtt{L}$
$T(s, \mathtt{L}')$	Set of tile dimensions $\mathtt{T_l}$ belonging to loop levels $\mathtt{l} \in \mathtt{L}' \subseteq \mathtt{L}$ in schedule s
$\kappa(v, \mathtt{l})$	Translation of loop level $\mathtt{l} \in \{\mathtt{x}, \mathtt{y}\}$ to loop dimension of corresponding kernel in layer v
$K(v, \mathtt{L}')$	Application of $\kappa(v, \mathtt{l})$ to complete set of loop levels $\mathtt{l} \in \mathtt{L}' \subseteq \mathtt{L}$
$inner(\mathtt{l}) = \mathtt{li}$	Translates a non-tiled loop level $\mathtt{l} \in \mathtt{L}$ into the corresponding inner tiled loop level $\mathtt{li} \in \mathtt{TL}$
$LT_{\mathtt{arr}}(s, \mathtt{L}')$	Set of loop levels in $\mathtt{L}' \subseteq \mathtt{L}$ that are below or equal to the store level in layer schedule s
$GE_{\mathtt{arr}}(s, \mathtt{L}')$	Set of loop levels in $\mathtt{L}' \subseteq \mathtt{L}$ that are above the store level in layer schedule s
$F_{\mathtt{arr}}(s, \mathtt{L}')$	Set of loop levels in $\mathtt{L}' \subseteq \mathtt{L}$ that are in between the compute and store level in layer schedule s
$FuseSel(v, s, A, B)$	If layer v is fused in layer schedule s return set A, else return set B.

B. INTERNAL MEMORY FOOTPRINT

With this notation in place, the required internal buffer size of a single scheduled layer $(v, s) \in S$ can be concisely and accurately modelled. This buffer size is comprised of three parts, the sum of the memory footprints of the $\mathtt{X}$, $\mathtt{Y}$, and $\mathtt{W}$ arrays respectively. All these footprints can be obtained by computing the volume of data below the respective store level $SL_{\mathtt{arr}}$, since this is the volume that will be loaded by the scheduled external memory access.

In general, for each loop level $\mathtt{l} \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{i}, \mathtt{m}, \mathtt{n}\}$ a selection has to be made between two options for each data array $\mathtt{arr} \in \{\mathtt{X}, \mathtt{Y}, \mathtt{W}\}$, one contribution to the data volume if said dimension is below $SL_{\mathtt{arr}}$, and one when it is equal or above. The product of these contributions yields the complete data volume.

1) WEIGHT ARRAY

For weight array $\mathtt{W}$ these options for the dimensions are explicitly listed in Table 5. Since the accesses to $\mathtt{W}$ are independent of loop levels $\mathtt{x}$ and $\mathtt{y}$, as can be seen in line 10 of Code 1, these loop levels do not contribute to the memory footprint of $\mathtt{W}$ (set to 1 for unit operation in the final product). For loop levels $\mathtt{m}$ and $\mathtt{n}$ the full dimension $\mathtt{D_m}$ and $\mathtt{D_n}$ has to be counted respectively, since these loops are excluded from tiling in the defined schedule space, and they are also always below the store level. More interesting are loop levels $\mathtt{z}$ and $\mathtt{i}$, which require a full tile $\mathtt{T_z}$ or $\mathtt{T_i}$ to be counted when they are below the store level, or only a single slice if they are not. The product of all correct contributions in Table 5 yields the initial memory footprint of the $\mathtt{W}$ array. However, care has to be taken when the compute level $CL_{\mathtt{W}}$ is below the store level, and buffer folding is applied. In the case that $\mathtt{z}$ and/or $\mathtt{i}$ are folded (below the store level, but above or equal to the compute level), they only contribute as if they were above the store level.

TABLE 5. Memory footprint contributions of $\mathtt{W}$ for all loop levels.

Loop Lvl ($\mathtt{l}$)	$SL_{\mathtt{W}} \prec \mathtt{l}$	$SL_{\mathtt{W}} \succeq \mathtt{l}$
$\mathtt{x}$	1	1
$\mathtt{y}$	1	1
$\mathtt{m}$	$\mathtt{D_m}$	–
$\mathtt{n}$	$\mathtt{D_n}$	–
$\mathtt{z}$	$\mathtt{T_z}$	1
$\mathtt{i}$	$\mathtt{T_i}$	1

Using the introduced notation, the memory footprint of the weight array $\mathtt{W}$ of a single convolution layer can be expressed as follows:

$$\mathrm{Buf}_{\mathtt{W}} = \prod T(LT_{\mathtt{W}}(\Lambda) - F_{\mathtt{W}}(\Lambda)) \times \prod D(\{\mathtt{m}, \mathtt{n}\}) \times \mathit{FuseSel}(\mathtt{D_i}, 1),$$

where $\Lambda = \mathit{FuseSel}(\{\mathtt{z}\}, \{\mathtt{z}, \mathtt{i}\})$. I.e., for the kernel loops $\mathtt{m}$ and $\mathtt{n}$ the full dimensions $\mathtt{D_m}$ and $\mathtt{D_n}$ are counted. For the non-fused case as described in Table 5, the tile sizes of loop dimensions $\mathtt{z}$ and $\mathtt{i}$ are taken into account, provided they are below the store level and not folded. When the layer is fused into a successor however, the computation changes slightly because dimension $\mathtt{i}$ can no longer be tiled. Considering the nonlinear activation function on line 11 of Code 1, all contributions in the $\mathtt{i}$ dimension have to be reduced before this activation can be applied, and the next layer can start its dependent computations. This can be more clearly seen in Code 3, where, due to the activation function on line 18, the complete (untiled) $\mathtt{i}$ loop on line 12 has to be computed before the next layer can start production on line 22. Thus, $\mathtt{i}$ can not be tiled and instead the full $\mathtt{D_i}$ dimension is required, as is covered by the *FuseSel* selection function.

2) INPUT ARRAY

For input array $\mathtt{X}$ a similar equation can be derived. The notable differences are that $\mathtt{X}$ is independent of loop level $\mathtt{z}$, and that at loop levels $\mathtt{x}$ and $\mathtt{y}$ at least $\mathtt{D_m}$ and $\mathtt{D_n}$ input elements are required. When a tile is required in these dimensions, i.e. $\mathtt{x} \prec SL_{\mathtt{X}}$ or $\mathtt{y} \prec SL_{\mathtt{X}}$, the kernel size also comes into play and, as illustrated in Figure 2, a contribution of $\mathtt{T_x} + \mathtt{D_m} - 1$ or $\mathtt{T_y} + \mathtt{D_n} - 1$ is required respectively. Another complicating factor is formed by the strides in the $\mathtt{x}$ and $\mathtt{y}$ dimensions, which change these terms to $(\mathtt{T_x} - 1) \times \mathtt{S_x} + \mathtt{D_m}$ and $(\mathtt{T_y} - 1) \times \mathtt{S_y} + \mathtt{D_n}$ respectively. The resulting memory footprint of a single layer is captured by the following expression:

$$\begin{aligned}\mathrm{Buf}_{\mathtt{X}} = \mathit{FuseSel}&\left(\mathtt{D_i}, \prod T\left(LT_{\mathtt{X}}(\{\mathtt{i}\}) - F_{\mathtt{X}}(\{\mathtt{i}\})\right)\right)\\ &\times \prod K\left(GE_{\mathtt{X}}(\{\mathtt{x}, \mathtt{y}\}) \cup F_{\mathtt{X}}(\{\mathtt{x}, \mathtt{y}\})\right)\\ &\times \prod\{(\mathtt{T_d} - 1) \times \mathtt{S_d} + \kappa(\mathtt{d}) \mid\\ &\qquad \mathtt{d} \in LT_{\mathtt{X}}(\{\mathtt{x}, \mathtt{y}\}) - F_{\mathtt{X}}(\{\mathtt{x}, \mathtt{y}\})\}.\end{aligned}$$

Again, *FuseSel* is used to account for the full dimension in $\mathtt{i}$ when the layer is fused, for the same reason as in $\mathrm{Buf}_{\mathtt{W}}$, i.e., due to the nonlinear activation function on line 11 in Code 1.

3) OUTPUT ARRAY

Finally, the memory footprint of the output array $\mathtt{Y}$ is relatively straightforward, and depends on $\mathtt{x}$, $\mathtt{y}$, and $\mathtt{z}$. For each of these dimensions the contribution is equal to the tile size, unless the dimension is above the store level or folded. That is, unless the current layer will be fused into the next layer, in which case the output array $\mathtt{Y}$ is effectively replaced by data array $\mathtt{X}$ of the next layer, yielding no memory contribution for $\mathtt{Y}$. Furthermore it is important to note that once an output is complete, i.e., all $\mathtt{D_i}$ contributions of the preceding layer have been processed and the activation function is applied, there is no need to keep the completed output on-chip. From a memory viewpoint this resembles buffer folding, which is also how this optimization is taken into account in the final equation. The footprint contributions of those loop levels in $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ which are above $\mathtt{i}$ fold to one, resulting in:

$$\mathrm{Buf}_{\mathtt{Y}} = \mathit{FuseSel}(0, \prod T(LT_{\mathtt{Y}}(\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}) - F_{\mathtt{Y}}(\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}))).$$

C. EXTERNAL MEMORY ACCESSES

The number of required external memory accesses can be derived in a similar manner as the internal memory footprint. The crucial difference is to not only account for the data volume transferred, but also how many times such a transfer takes place. These two terms can be considered separately, such that for data array $\mathtt{arr} \in \mathtt{ARR}$, the external memory accesses $\mathrm{Acc}_{\mathtt{arr}}$ are expressed as the volume of a data transfer $\mathrm{Vol}_{\mathtt{arr}}$, multiplied by the number of those transfers $\mathrm{Trans}_{\mathtt{arr}}$:

$$\mathrm{Acc}_{\mathtt{arr}} = \mathrm{Vol}_{\mathtt{arr}} \times \mathrm{Trans}_{\mathtt{arr}}.$$

1) WEIGHT ARRAY

For weight array $\mathtt{W}$, the volume of a transfer $\mathrm{Vol}_{\mathtt{W}}$ is nearly identical to its internal memory footprint $\mathrm{Buf}_{\mathtt{W}}$, with the notable exception that for the transfer volume buffer folding has no effect. The fact that the buffer is smaller due to liveness of the variables does not invalidate the requirement to transfer the complete volume eventually. The transfer volume of array $\mathtt{W}$ can therefore be expressed as:

$$\mathrm{Vol}_{\mathtt{W}} = \prod T\left(LT_{\mathtt{W}}(\Lambda)\right) \times \prod D\left(\{\mathtt{m}, \mathtt{n}\}\right) \times \mathit{FuseSel}(\mathtt{D_i}, 1),$$

where $\Lambda = \mathit{FuseSel}(\{\mathtt{z}\}, \{\mathtt{z}, \mathtt{i}\})$. Note that when the current layer is fused into the next, all inputs $\mathtt{D_i}$ need to be handled before the results can be passed to the next layer. Again, the nonlinear activation function on line 11 of Code 1 prevents partial updates of only $\mathtt{T_i}$ inputs to be consumed.

Next the number of transfers is to be determined. In general, if a loop level $\mathtt{l}$ is above or equal to the store level, the associated volume needs to be transferred for every $\mathtt{D_l}$ iterations. When $\mathtt{l}$ is beneath the store level however, that volume will have to be transferred only $\left\lceil \frac{\mathtt{D_l}}{\mathtt{T_l}} \right\rceil$ times. The ceiling operator is used here to arrive at a conservative bound, which accounts for a full tile transfer in case tile size $\mathtt{T_l}$ is not an exact multiple of $\mathtt{D_l}$. For weight array $\mathtt{W}$ the number of transfers is given by:

$$\mathrm{Trans}_{\mathtt{W}} = \prod D\left(GE_{\mathtt{W}}(\Lambda)\right) \times \prod \left\{ \left\lceil \frac{\mathtt{D_d}}{\mathtt{T_d}} \right\rceil \,\middle|\, \mathtt{d} \in LT_{\mathtt{W}}(\Lambda) \right\},$$

where $\Lambda = FuseSel(\{\mathtt{x},\mathtt{y},\mathtt{z}\}, \{\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{i}\})$ is used to compensate when due to fusion the entire volume is transferred.

2) INPUT ARRAY

For data array X the transfer volume resembles the internal buffer size of array X, $\mathrm{Buf}_{\mathtt{X}}$, again ignoring any buffer folding:

$$\begin{aligned}\mathrm{Vol}_{\mathtt{X}} &= FuseSel\left(\mathtt{D_i}, \prod T\left(LT_{\mathtt{X}}(\{\mathtt{i}\})\right)\right)\\ &\times \prod K\left(GE_{\mathtt{X}}(\{\mathtt{x},\mathtt{y}\})\right)\\ &\times \prod \left\{(\mathtt{T_d}-1)\times \mathtt{S_d} + \kappa(\mathtt{d}) \mid \mathtt{d} \in LT_{\mathtt{X}}(\{\mathtt{x},\mathtt{y}\})\right\}.\end{aligned}$$

The number of transfers for X is in fact equal to those of W ($\mathrm{Trans}_{\mathtt{W}}$), apart from checking against the store level of X instead of W, i.e.,

$$\mathrm{Trans}_{\mathtt{X}} = \prod D(G\mathtt{T}_{\mathtt{X}}(\Lambda)) \times \prod \left\{ \left\lceil \frac{\mathtt{D_d}}{\mathtt{T_d}} \right\rceil \,\middle|\, \mathtt{d} \in LT_{\mathtt{X}}(\Lambda) \right\},$$

where $\Lambda = FuseSel(\{\mathtt{x},\mathtt{y},\mathtt{z}\}, \{\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{i}\})$.

3) OUTPUT ARRAY

Finally, for the number of external memory accesses for array Y, it is easier to deviate from the volume/transfer approach used above. When a layer is fused, the output simply does not contribute to the external transfers, as the outputs are stored directly in the X array of the layer that is being fused into. When a layer is not fused, eventually the complete output, i.e. $\mathtt{D_x D_y D_z}$ elements, have to be transferred at least once to the external memory. More than one transfer per output element may be required if partial results are stored in (and later loaded from) external memory. Here, a partial result is a partial sum in array Y which is not yet ready to be passed to the nonlinear activation function. For internal buffer space it could be interesting to evict some of these partial results from the local buffer, and load them back later. This happens only if tiling is applied to the i loop. In that particular case the partial output elements have to be transferred twice for each tile in i, once for storing the partial results externally, and once for loading them back (excluding the first update of Y). Combined this yields the following expression for the number of elements transferred for output array Y:

$$\mathrm{Acc}_{\mathtt{Y}} = \mathtt{D_x D_y D_z} \times \left(\left\lceil \frac{\mathtt{D_i}}{\mathtt{T_i}} \right\rceil \times 2 - 1 \right).$$

D. COMPUTE

The final part of the model represents the number of multiply-accumulate operations (MACs) required to complete a schedule. Without recompute, this number is trivial to obtain by multiplying all dimensions of a layer $\mathtt{D_x} \times \mathtt{D_y} \times \mathtt{D_z} \times \mathtt{D_c} \times \mathtt{D_n} \times \mathtt{D_m}$. However, to account for recompute due to overlap of input tiles detailed later in Section IV-E, this formula is split into three terms: the number of MACs to produce a single output pixel PMACs, the volume of a produced output tile OVol, and the number of such output volumes in a layer.

The number of MACS for a single output pixel is fairly straightforward, and is determined by the number of input feature maps multiplied by the kernel size:

$$\mathrm{PMACs} = \prod K(\{\mathtt{x},\mathtt{y}\}) \times \mathtt{D_i}.$$

The produced volume in number of features for every transfer is also straightforward, and amounts to those tiles which are below the store level of array X:

$$\mathrm{OVol} = \prod T(GE_{\mathtt{X}}(\{\mathtt{x},\mathtt{y},\mathtt{z}\})).$$

Note that the store level of array X is used, since the input volume determines the produced output volume. Consequently, the number of such volumes is simply equal to the number of transfers of array X, $\mathrm{Trans}_{\mathtt{X}}$.

The total number of MACS is thus given by:

$$\mathrm{MACS} = \mathrm{PMACS} \times \mathrm{OVol} \times \mathrm{Trans}_{\mathtt{X}}.$$

E. LAYER FUSION

Now that models have been established for the memory footprint, external memory accesses, and number of multiply-accumulate operations per layer for each of the W, X, and Y arrays in the preceding sections, these models can be combined to provide the same properties for complete sets of fused layers. This is achieved by 'chaining' the provided layer models in a recursive fashion. This is best understood by observing the consumption of X0 in Figure 3 used to produce X1. Instead of producing complete array X1, i.e., $\mathtt{D_{x1}} \times \mathtt{D_{y1}} \times \mathtt{D_{i1}}$ as the output size of X0, in a fused schedule only a single tile $(\mathtt{T_x} + \mathtt{D_{k1}} - 1) \times (\mathtt{T_y} + \mathtt{D_{l1}} - 1) \times \mathtt{D_{i1}}$ needs to be produced at a time. Substituting $\mathtt{D_x}, \mathtt{D_y}, \mathtt{D_z}$ with $(\mathtt{T_x} + \mathtt{D_{k1}} - 1) \times (\mathtt{T_y} + \mathtt{D_{l1}} - 1) \times \mathtt{D_{i1}}$ respectively in the models provides the costs of this intermediate tile:

- Multiply accumulates: The number of multiply accumulates required to produce the tile from x0 to X1 are simply given by PMACs($L01$), where $L01$ represents the layer that consumes X0 and produces X1.
- Memory footprint: The local memory footprint of the tile in X0 is given by $\mathrm{Buf}_{\mathtt{X}}(L01)$, the required footprint of the tile in X1 is given by $\mathrm{Buf}_{\mathtt{X}}(L12)$. The footprints of the weights can used without any substitution.
- Accesses: The intermediate tile in X0 of course does not require any external data accesses, as it is produced and consumed in a fused fashion. However, the number of transfers $\mathrm{Trans}_{\mathtt{X}}(L12)$ specify how many times the tile needs to be produced in the case of recomputation, and tile overlap in general. For the overall cost of the fused set of layers, the multiply accumulates for the tile are to be multiplied with the number of productions. Same holds for the loads of the weights required to produce the tile, unless they are completely stored in the local buffer. Also, this number of required productions moves further up the set of fused layers, as consequently any producers of this tile may in turn need to be produced

multiple times if they are not stored for the complete lifetime of the network depending on their store level and tile sizes. Only the first layer of a set of fused layers, has external memory accesses. If this first layer needs multiple transfers of its own, i.e., not all uses of the data elements are captured on first load, the number of additional accesses can grow quite quickly due to the recomputation effect. The more profitable schedules therefore typically ensure the input is loaded only once, such that the cost of recomputation indeed only affects the compute cost, and does not increase the number of external accesses.

Hence, using substitution of tile dimensions for the output dimensions, and propagation of the number of productions required for each tile, the total cost of a fused segment can be derived from the individual cost models presented in the previous sections.

F. COMPLETE NETWORK MODEL

The total costs of a complete network are now trivial. The network schedule effectively partitions the layers into groups of fused layers. These groups contain one or multiple layers, for which the costs can be derived using the fusion approach described in the previous section. The total number of multiple accumulates required by a network schedule is simply obtained by adding the multiply accumulates of all groups. Same holds for the number of external accesses, the sum of the accesses of each group forms the cost for the entire network. The memory however only requires the maximum buffer size over all groups, since only a single group is active at any given time during network evaluation assuming no pipelining of the execution. Using these simple rules, the costs of an entire network can be computed.

V. AUTOMATED DESIGN SPACE EXPLORATION

The formal model introduced in Section IV enable automated exploration of the vast scheduling space described in Section III. To achieve complete automation Section V-A describes a strategy to efficiently traverse the entire scheduling space. Section V-B introduces `ConvFuser` [12], an open source tool which implements the presented traversal strategy and cost models, enabling automated design space exploration and verification of any neural network described in the popular Keras framework [11].

A. SPACE TRAVERSAL

To effectively explore the scheduling space as described in Section IV four steps are required:

1) IDENTIFICATION OF SEQUENCES

Since the proposed models do not include provisioning to handle residual/skip connections, only consecutive layers without forks or joins are considered for fusion. The first step of a design space exploration (DSE) is therefore to select sets of layers that may be fused. Such a set of eligible layers is referred to as a sequence, an example of which is shown in Figure 4.

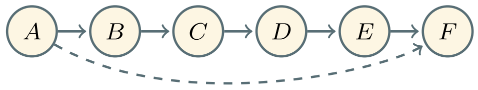


FIGURE 4. Example layer graph with sequence `[B, C, D, E]`, and a residual connection between `A` and `F`.

2) SEGMENTATION

Within each sequence it needs to be decided which layers to fuse (if any) to obtain the Pareto optimal schedules of the network. A set of fused layers within a sequence is referred to as a segment. The sequence `[B,C,D,E]` of Figure 4 contains the following valid segments: `B, C, D, E, BC, CD, DE, BCD, CDE, BCDE`. Each of these segments is evaluated individually, yielding a vector of schedules $\vec{S}$ per segment.

3) PARTITIONING

Once all possible segments have been identified and their costs have been evaluated, those segments that cover the entire sequence need to be combined. For example segments `BC` and `DE` cover sequence `[B,C,D,E]`, but also segments `B, C,` and `DE`, as well as many more. Combining the schedule vectors $\vec{S}$ of each segment into an overall schedule vector $\vec{SS}$ for the sequence is done by taking their product, and using the rules outlined in Section IV-F. Note that this procedure can be significantly accelerated by first Pareto-filtering the segment schedule vectors $\vec{S}$. It can be trivially proven that considering only the Pareto points of each segment is sufficient to yield all the Pareto points of the entire partition, since the combining (reduction) functions of Section IV-F, i.e., summation and maximum selection, are monotonically increasing.

4) NETWORK SCHEDULE COST COMPUTATION

Finally the costs of all valid partitions are combined using the same rules of Section IV-F to obtain the cost of the entire network. Similarly to the partition cost computation, segment schedule vectors $\vec{SS}$ can be Pareto-filtered before combination with other partitions to yield a Pareto optimal scheduling of the entire network.

B. CONVFUSER

An embodiment of the automated design space exploration described in this work is provided in the form of an open source tool: `ConvFuser` [12]. Apart from automated DSE, `ConvFuser` also features code generation for any selected schedule, enabling reliable validation of the models.

`ConvFuser` builds upon the popular Keras/TensorFlow framework [11] to read standard HDF5 graph models (See Figure 5). After loading a network using Keras/TensorFlow, a graph is constructed from the network layers, and a custom canonicalisation pass is employed to normalize the network

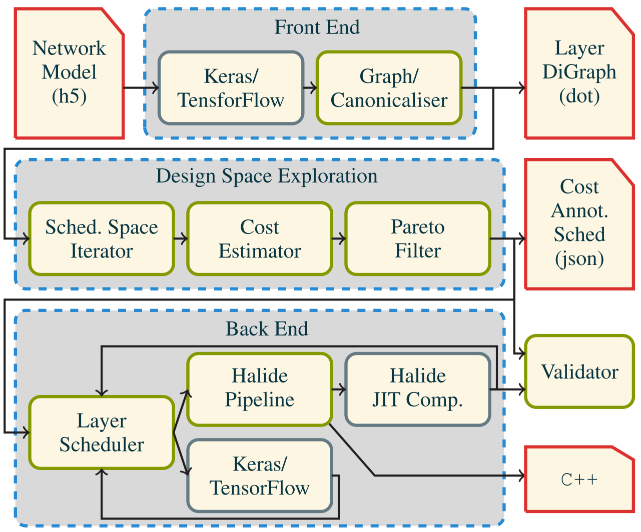


FIGURE 5. Schematic overview of the `ConvFuser` tool. Green boxes are developed for this work.

description. An important part of this canonicalisation is performing trivial layer merges, including but not limited to:

- merging of batch normalization layers into convolutional layers, which can be achieved by modification of the weights of the target convolutional layer.
- merging of activation layers into convolutional layers.

Note that some literature refers to these trivial layer merges as layer fusion. This term is apt in the case of merging activation layers, which also involves loop fusion, but it should not be confused with the much more complicated fusing of consecutive convolutional layers as described in this work.

After canonicalisation, design space exploration can be performed. Many smart search strategies could be employed here, but by virtue of the mathematical models and their fast evaluation, straightforward exhaustive search is feasible for smaller networks. Additionally the tool provides several options to restrict the design space, such as limiting the number of layers considered for fusion, selection of tile sizes such as only exact multiples of their respective dimension, or only powers of two, and whether to consider recomputation.

Finally, to enable validation of the found schedules, a hybrid back end based on the Halide language [3] and Keras/TensorFlow [11] is provided. Any non-convolutional layers are evaluated directly by Keras/TensorFlow. The scheduled convolutional layers however are implemented using a modified version of Halide. In particular this modification consists of additional python bindings to be able to insert instrumentation code. This code utilizes Halide's internal tracing mechanism to keep track of accesses to buffers, and sizes of allocated buffers. To emulate an external memory and internal buffers, a construct similar to Halide's recently added `in` operator is used. This construct adds an extra layer of buffering, where one large buffer essentially mimics external memory, and working data is loaded into a smaller, internal buffer. By keeping track of the accesses to these two levels of buffering, the required external accesses can be exactly monitored. An optional validation step can be used to check the external accesses, internal buffer size, and multiply-accumulates measured from Halide execution with the values predicted by the models.

VI. MODEL VALIDATION & EVALUATION

The validity of the models introduced in Section IV is confirmed experimentally using the `ConvFuser` tool introduced in Section V. The experiments consist of a design space exploration for several synthetic and real-world networks, followed by code generation for each of the Pareto optimal schedules. This code is automatically instrumented by the `ConvFuser` tool to measure the number of multiply accumulates, internal memory size, and external accesses, which are then compared against the modelled values.

Besides this model validation, the tool also enables the evaluation of the impact of various scheduling techniques. In essence, the scheduling space described in Section III can be restricted to subsets by disallowing or limiting certain scheduling techniques. For this evaluation four progressively inclusive scheduling spaces are defined:

1) Baseline: The baseline scheduling space follows the straightforward implementation in Code 1 for each layer. It thus excludes loop reordering, tiling, and fusion, but does allowing different store and compute levels.
2) Tiling & Reorder: This space adds both loop tiling and reordering to the allowed options to the baseline space. Tiling is restricted to sizes that are integer factors of the dimensions to keep the space to a size that can be completely traversed in reasonable time. Other tiling options are built into `ConvFuser`, but are not further evaluated in this section.
3) Goetschalckx *et al.*: This space extends the tiling and reorder options with layer fusion. The allowed fusion however does not recompute any elements. Furthermore weights are not tiled, and are always stored on-chip for a fused section. This space matches the work of Goetschalckx *et al.* [10], enabling direct comparison. One notable exception for this space is that the tiling factor is not limited as is the case in the work of Goetschalckx *et al.*, since for this particular design space it is possible to use a fast branch and bound algorithm on the tiling factor while still guaranteeing an optimal result.
4) Fusion & Recomputation: This extends the *Goetschalckx et al.* space with tiling of weights and recomputation. The addition of tiling the weight accesses however disallows the previously mentioned branch and bound technique on the tiling factor without losing optimality. Therefore this space again is constrained to the same tiling limitations as the Tiling & Reorder space, i.e., integer factors of the dimensions.

This partitioning of the scheduling space enables the investigation of the impact of tiling, loop reordering, fusion, and

TABLE 6. Layer dimensions of L2Net and L3Net for 20 × 20 and 4K inputs.

Net	Dx	Dy	Di	Dz	Dn	Dm
L2Net	18	18	3	4	3	3
	16	16	4	4	3	3
L3Net	20	20	3	4	3	3
	18	18	4	4	3	3
	16	16	4	4	3	3
L2Net 4K	3838	2158	3	4	3	3
	3836	2156	4	4	3	3
L3Net 4K	3838	2158	3	4	3	3
	3836	2156	4	4	3	3
	3834	2154	4	4	3	3
L2NetWide	18	18	3	1024	3	3
	16	16	1024	4	3	3
L3NetWide	20	20	3	1024	3	3
	18	18	1024	1024	3	3
	16	16	1024	4	3	3

recomputation on MAC count, required on-chip memory, and external accesses.

A. MICRO BENCHMARKS

In order to validate the models introduced in Section IV, and obtain insight into the effect of different scheduling techniques, first a number of micro benchmarks is performed. To this end two synthetic networks, L2Net and L3Net, are defined. These networks consists of respectively two or three convolutional layers with a 3×3 kernel. This enables a study into the effects on the resulting scheduling space when deepening networks. The dimensions of these layers are shown in Table 6 for both a 20×20 input and a $4K$ input, used to see the effects of network input size on the resulting schedules. Finally the number of feature maps of these two nets is increased, yielding L2NetWide and L3NetWide in Table 6, to examine the effect of this parameter on the effect of the modelled scheduling techniques.

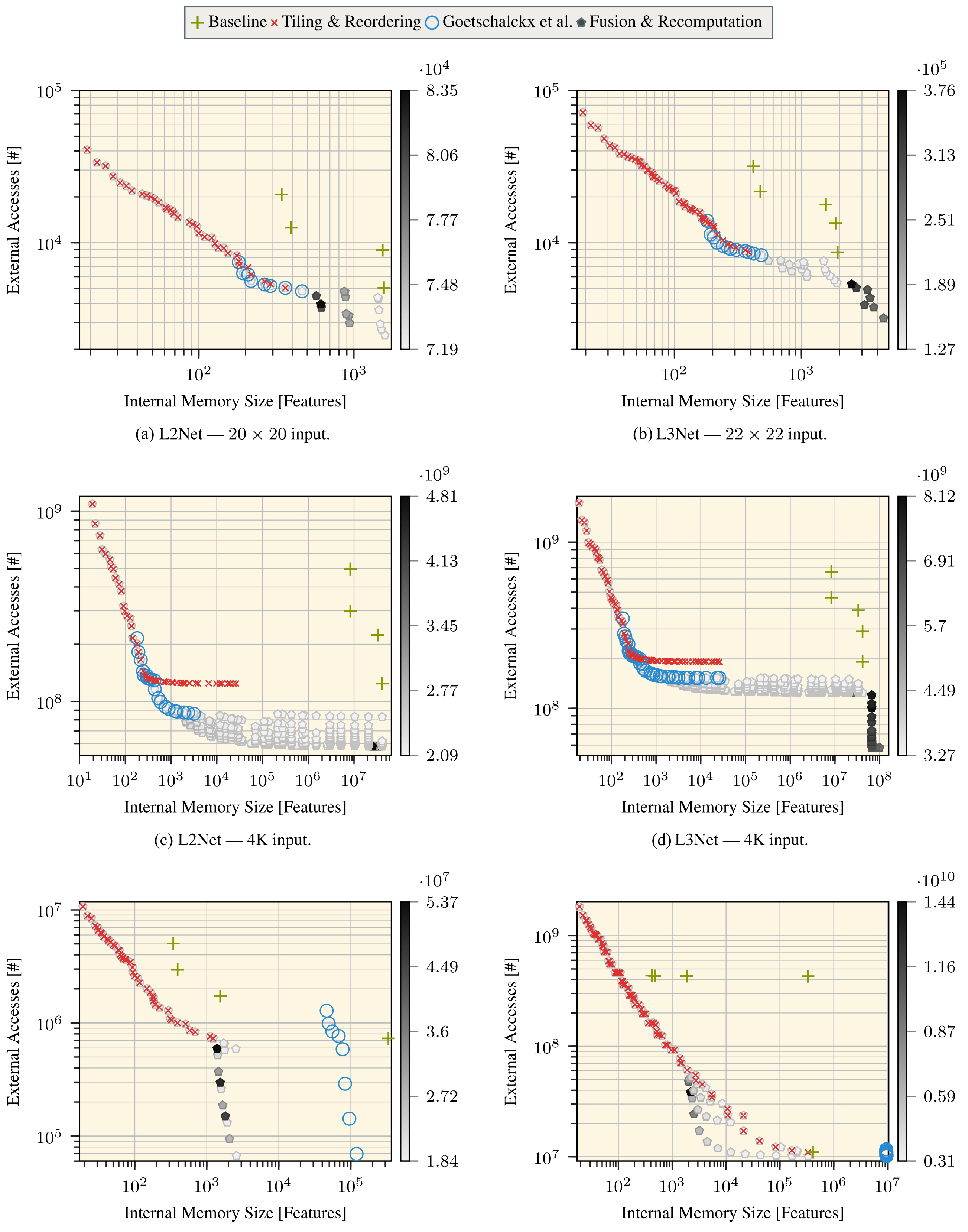
The results of the schedule space exploration for L2Net and L3Net with 20×20 and 22×22 inputs respectively are shown in Figures 6b and 6c. For all schedules of a network the number of multiply-accumulates is constant, with the exception of those schedules that include recomputation. To indicate the number of required multiply-accumulates, a colour coding is added for the Fusion & Recomputation schedules. The Pareto schedules for L2Net and L3Net with 4k inputs are shown in Figures 6d and 6e respectively. Finally the Pareto fronts for L2NetWide and L3NetWide are given in Figures 6f and 6g. Note that the verification with instrumented halide for the selected points in Figure 6 showed a one to one match between the models and the measurements, apart from some corner cases where Halide failed to apply complete buffer folding. These results confirm the accuracy of the models presented in section IV.

From these figures five key observations can be made:

1) The baseline schedules are consistently poor over all networks, although they always include a point that minimizes the external accesses when fusion and recomputation are not considered. This naturally happens for the store-level that cover all data uses, but due to the lack of tiling and smart loop reordering these points typically require a huge amount of memory.
2) Adding tiling and loop reordering on top of the baseline schedule results exclusively in schedules that require less internal memory, and thus completely Pareto dominate the baseline points.
3) Fusion without recomputation and weight tiling, i.e., the space defined by Goetschalckx and Verhelst [10] does not add many interesting points for the small input versions of L2Net and L3Net. The accesses saved by omitting transfers on the small intermediate layers does not add much for these small networks where the number of weights are relatively significant. When applied to the 4k input networks however, some gains can be observed as the weight transfers become insignificant compared to the data transfers. However, the gains are still rather modest since the remaining transfers on the input layer have also scaled with the input resolution. More interesting are the points for L2NetWide, where fusion does yield larger gains as the number of feature maps increases, and the intermediate transfers start to dominate the remaining input transfers. However, as the weights are again important in these networks, the ideal combination for the Goetschalckx scheduling space, i.e., large inputs and wide intermediate layers, is not achieved in these synthetic benchmarks.
4) Enabling weight tiling and recomputation however does yield some interesting points. In particular for the wide networks with small input, i.e., those networks where weights are significant, some schedules with even modest recomputation are found to outperform the Goetschalckx schedules by an order of magnitude in required memory size, demonstrating the added value of the more generic models presented in this work.
5) As the input size and number of layers increases, recomputation becomes more and more relevant, but only for very large memory sizes. Apart from these extreme cases, the memory saved by heavy recomputation is rather marginal compared to neighbouring schedules with minimal to no recomputation.

B. REAL-WORLD NETWORKS

Although above synthetic micro benchmarks provide insight into general trends, it is important to also evaluate the scheduling techniques in the context of real world networks. To this end six widely used networks are selected for experimentation: ResNet50, VGG16, InceptionV3, MobileNetv2, and XCeption as implemented in the Keras framework [11], and DMCNN-VD, the demosaicing as described by Syu *et al.* [22] and also evaluated by Goetschalkx *et al.* allowing for direct comparison. The scheduling Pareto fronts of these networks are given in Figures 7b, 7c, 7d, 7e, and 7g respectively. Because the complete design spaces of these networks are exceptionally large,



(a) L2Net — 20 × 20 input.

(b) L3Net — 22 × 22 input.

(c) L2Net — 4K input.

(d) L3Net — 4K input.

(e) L2NetWide — 20 × 20 input.

(f) L3NetWide — 22 × 22 input.

FIGURE 6. Pareto schedules of the synthetic networks for the four defined scheduling spaces. The colour maps only apply to fusion & recomputation, and represents the number of multiply-accumulate operations.

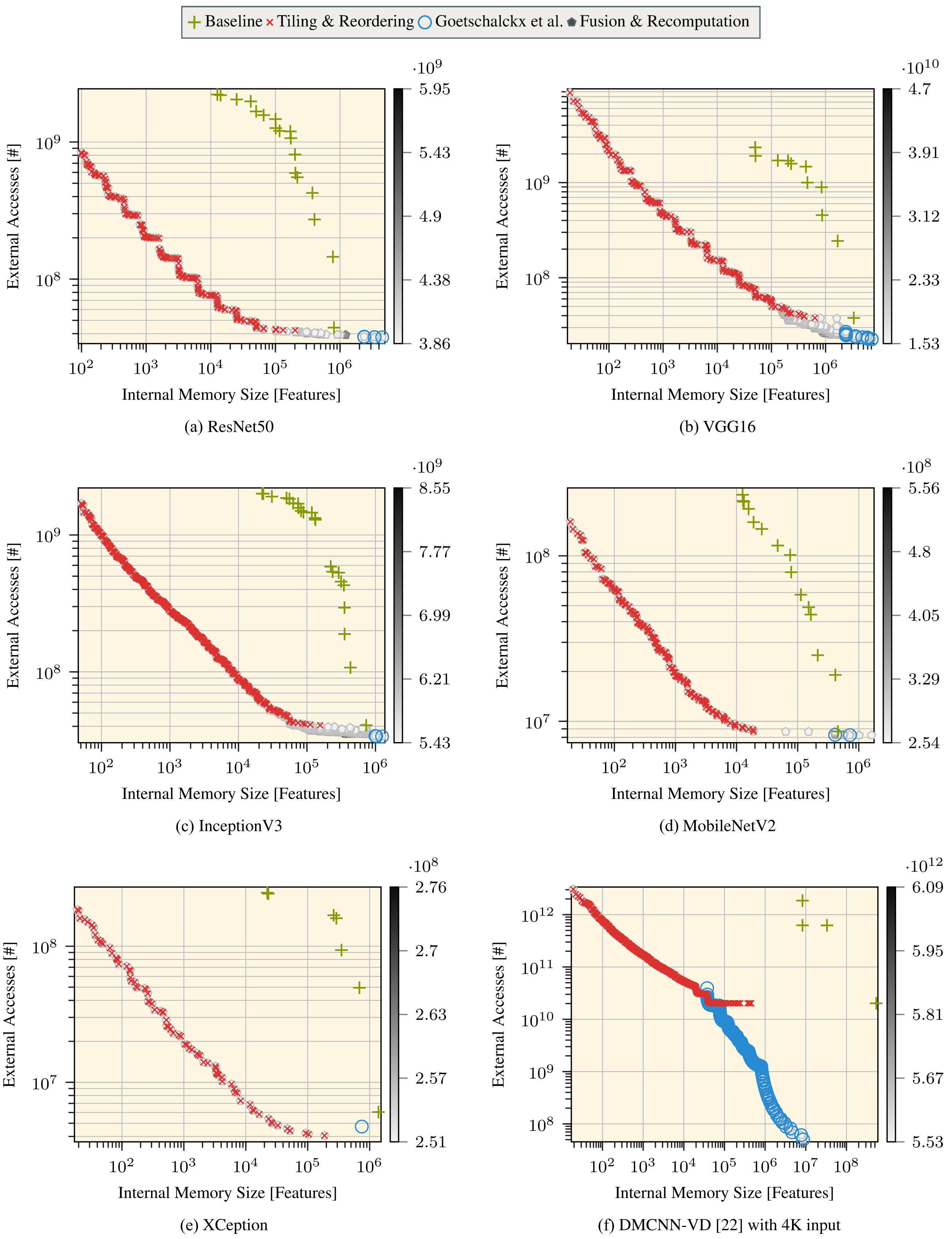


FIGURE 7. **Pareto schedules of six Real-World networks for the four defined scheduling spaces. The colour maps only apply to fusion & recomputation, and represents the number of multiply-accumulate operations.**

an exhaustive traversal of the design space is infeasible even with the presented fast models. Therefore the exploration of the design space of these networks was limited to ten million considered schedules per segment.[2] This limitation particularly impacts the Fusion & Recomputation space, since it is the largest of the four spaces. As a result the Pareto fronts are less smooth than those of the micro-benchmarks which are exhaustively searched. Nonetheless, the general trends can be easily observed, and plenty of schedules remain on the Pareto fronts for practical purposes. From Figure 7 the following three observations are made:

1) For typical real-world networks with 224×224 ImageNet input resolutions, the gain of fusion and recomputation on top of tiling and reordering is rather limited. This effect is expected based on the micro-benchmarks, in particular keeping in mind that in real-world networks various layer types prohibit the application of fusion with the presented models, limiting the benefits of fusion even further. Only for very large memories do the accesses decrease.
2) For networks with large input, and a straightforward structure of sequential convolution layers such as DMCNN-VD, the story is slightly different. Here extra memory can effectively be used for fusion to reduce the number of external accesses significantly.
3) The Fusion Pareto front of DMCNN-VD matches the experiments of Goetschalckx and Verhelst [10], further validating the more generic models presented in this work. Moreover, more Pareto schedules are found as the tiling factor was not limited in our experiments by virtue of a branch and bound design space strategy on the tiling factor which still guarantees optimality. This enabled exploration of the complete scheduling space of DMCNN-VD as defined by Goetschalckx and Verhelst [10] in a matter of minutes.

In general it can be concluded that the presented models are very accurate, and have added value over the state of the art in particular for networks where the number of weights is significant. Although for real-world networks the benefits of fusion are somewhat limited due to complex connections and various layer types, the external accesses can in most cases be reduced compared to loop reordering and tiling only. The best observed reduction in external accesses was as high as 99.75% for DMCNN-VD. On average the gain of fusion over tiling and loop reordering was 28.89% for the six selected networks.

VII. ENERGY CONSUMPTION

The scheduling results presented in the preceding section expose the potential to reduce the external memory accesses using advanced scheduling techniques. This section extends these results with a short study into the effects of this reduction in accesses, since for most practical designs not the raw accesses, but the energy consumption is of interest.

[2]See Section V-A2 for the precise definition of a segment.

Building upon the metrics produced by the models introduced in this work, the remainder of this section introduces progressively more realistic energy models starting from a single level, single bank SRAM internal memory in Section VII-A till a multi-level, multi-bank internal memory model in Section VII-C

A. SINGLE-BANK SRAM

The energy required to evaluate a neural network according to a specific schedule can be split into three distinct parts:

1) MULTIPLY-ACCUMULATE OPERATIONS

The first source of energy consumption is the execution of multiply-accumulate operations. The energy required for a single multiply-accumulate depends on the data type of the operation, the arithmetic architecture, the operating speed, and the technology node used for implementation. How these parameters influence the energy consumption is complicated, and instead of attempting to derive a generic model choices are made for these parameters in this evaluation. In particular, a 40 nm node is assumed for which energy numbers are available in the Aladdin tables [23], which are also used by the state-of-the-art Accelergy energy estimation tool from MIT [24]. In alignment with the accelergy framework, the cost of a multiplication and addition at 4 ns delay are summed to conservatively approximate a multiply-accumulate unit. Although the model metrics are agnostic to the width of a neuron output or weight value, a choice has to be made to be able to provide an energy estimate. Since the listed numbers are for 32b operations, and the operations in neural networks usually only require 16b, a scaling factor is applied. Specifically, the energy of the addition is halved since adders scale approximately linearly, and the multiplication energy is divided by four because of quadratic scaling of multipliers. Following this method, the energy cost of a single multiply-accumulate operation is approximated at 10.2 pJ.

2) ACCESSING EXTERNAL DRAM

The second factor influencing the energy consumption is accessing external memory. DRAM is a typical choice for off-chip memory, and is also assumed in this evaluation. According to the work of Malladi *et al.* [25] accessing a single bit in DDR3 memory requires about 70 pJ. For simplicity it is assumed that both the neuron outputs and weights are 16 bit wide, yielding an energy cost of 1120 pJ per DDR3 DRAM access. Note that this number is taken to be independent of the DRAM size, since the energy is dominated by IO logic rather than the size of the memory array.

3) ACCESSING INTERNAL SRAM

For internal memory SRAM is assumed. Estimating the energy of an SRAM access is slightly more involved, since the size of the memory array does matter significantly for the access energy. To be able to provide accurate estimates, a model is derived based on commercial-off-the-shelf SRAM modules on a 40 nm technology node. Figure 8 shows

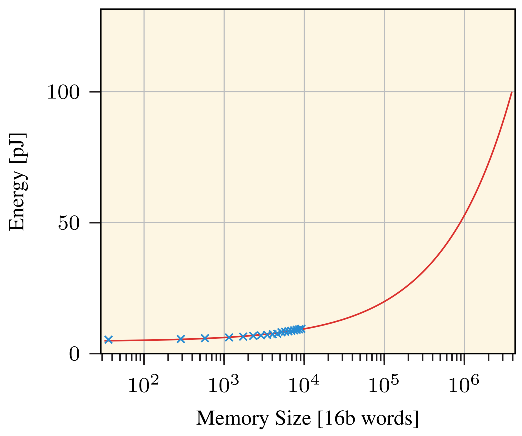


FIGURE 8. SRAM energy consumption based on commercial 40 nm SRAM modules at the typical corner, 25 °C, 1.1 V.

available data points for these modules in terms of access energy for 16b words based on the total module size in bits. These numbers are based on the typical corner, 25 °C, 1.1 V, averaging a read and write access. A square root function is fit through these points to enable extrapolation, yielding the energy per access of an s-bit SRAM cell in pJ:

$$E_{\mathrm{SRAM}}(s) = 0.012 \cdot \sqrt{s} + 4.61$$

A square root function is chosen as the energy cost of larger banks mainly seems to scale with the circumference of the array, and the additional sense amplifiers used to partition the bank internally. This model is used to extrapolate the energy cost of an SRAM bank in Figure 8. Note that the range of the extrapolation is rather extreme to be able to support the large memories required for some of the fusion schedules. This already indicates a different approach to constructing such large memories should be taken in practice, which will be further elaborated in Section VII-B. Although the selected square root function fits the relation quite well (see also Figure 10 which clearer shows the fit through the available data-points), this model can easily be exchanged for more sophisticated and accurate models when available.

Adding these three factors, i.e., multiply-accurate energy, DRAM energy, and SRAM energy, yields the total energy estimated by this basic model. For the number of multiply-accumulates the `MAC` numbers from the model can be directly used, the accesses to DRAM are also directly given by the various `Acc` terms, as is the size of the internal SRAM by taking the maximum of the required `Buf` formulas over all layers. This leaves the number of accesses to the SRAM however, which are given by the number of multiply-accumulates times four, since each MAC operation requires three reads and one write. Note that this basic model lacks a register file or accumulation register, and hence all accesses go to the internal SRAM. Finally each external read access will also write to the internal SRAM, so these accesses are also added.

Applying this model to the scheduling front of the L3NetWide and L3Net 4k networks presented in Section VI yields Figure 9. Since the energy model is monotonically increasing in all parameters, the Pareto schedules found in Section VI are guaranteed to contain those schedules that are also Pareto in terms of energy. The energy points are not Pareto filtered in this case however to highlight an important drawback of the used energy model. Using the simplistic $E_{\mathrm{SRAM}}(s)$ approximation, the internal SRAM memory quickly becomes very expensive to access as its size increases. In particular, past around 5×10^3 features the reduction in external DRAM accesses no longer outweighs the increased cost of accessing the internal SRAM, and the energy starts to increase again. The problem is that the access energy of a single SRAM bank does not scale very well, and for those schedules that require a large amount of internal memory a more sophisticated memory model is required.

B. MULTI-BANK SRAM

The shortcoming of the basic energy model is the assumption of a single SRAM bank also for large internal memories. In practice, however, such large memories will always be constructed out of several smaller SRAM banks. Such memories will incur an area penalty compared to the single bank approach, but the access energy is lowered significantly. In fact, the access energy is equal to the access energy of a single small bank that makes up the larger memory, plus some overhead for the bank selection logic. To correct for this energy overhead a scaling function is fit based on the work of Mai *et al.* [26]. To minimize the impact of the area overhead, and provide a pessimistic estimate of what can be achieved with a banked SRAM memory, the largest available SRAM block from the commercial 40 nm library is selected. Combined with the energy overhead function this yields the following model for the access energy of an s-bit multi-bank SRAM memory:

$$\begin{aligned} E_{\mathrm{BankedSRAM}}(s) &= E_{\mathrm{SRAM}}(\min(s, 16 \times 10^4)) \\ &\quad \times \left(1 + 3.05 \times 10^{-3} \times \log\left\lceil \frac{s}{16 \times 10^4} \right\rceil\right) \end{aligned}$$

This relation is visualized in Figure 10, from which it is clear that the overhead of the bank selection logic indeed is relatively small. Plugging this multi-bank SRAM model into the energy model described in Section VII-A, and again applying it to the L3NetWide and L3Net 4k networks yields Figure 11. It is immediately clear for this figure that the multi-bank SRAM model largely solves the energy problem for schedules that require large amounts of internal memory. For L3NetWide the problem is gone entirely, while for L3Net 4k the optimum memory size still lies around 1×10^3 entries. In both cases it is clear however the network schedules with larger internal memory requirements hardly benefit from the reduction in external accesses, even while accessing the DRAM is relatively expensive. Furthermore all schedules with recomputation always increase the energy consumption. Both these observations can be explained by the lack of a register file or other small localized memory, and as such each additional multiply-accumulate incurs four accesses to the internal SRAM.

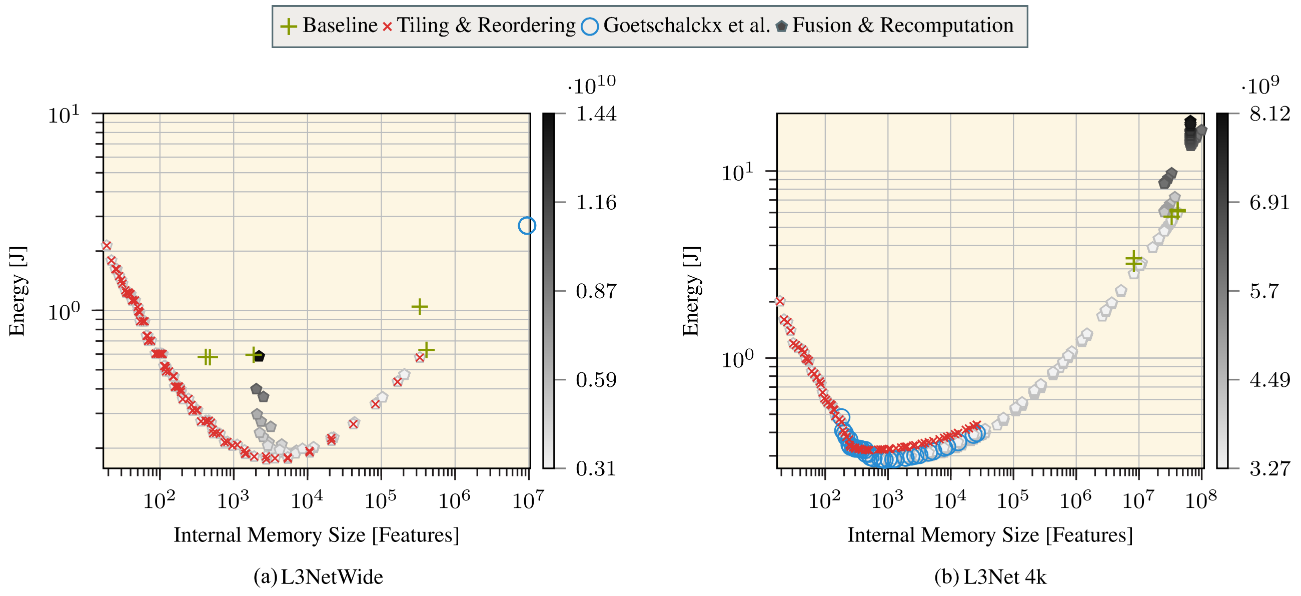


FIGURE 9. Energy front of L3Net based models using the single bank SRAM model outlined in Section VII-A. The colour maps only apply to fusion & recomputation, and represents the number of multiply-accumulate operations.

C. MULTI-LEVEL SRAM

To overcome the limitation of a single level internal SRAM memory, this section expands the energy model with multi-level internal memory. This does present a problem, however, since the models presented in this work do not support multi-level memories in a precise manner. Section VIII contains notes on how such support may be added as part of future work, but for this evaluation an approximation is used instead. In particular, when a memory level of size s is added, the best schedule available for that size is used to model the accesses to that level. Furthermore the 'external accesses' modelled for this schedule will now be added to the next level of memory. This next level is determined by a sweep over the remaining schedules, and using their internal memory size for this level. Using this methodology a register file with $64 \times 16b$ entries is added to the energy model. For accessing this register file an average accesses energy of 2.4 pJ is used in accordance with the findings of Wu *et al.* [27].

The resulting energy fronts of the L3NetWide and L3Net 4K networks are given in Figure 12. It can be seen that the addition of the register file ensures that the energy does decrease when larger internal memory is used. Also in L3Net 4k some of the schedules with recomputation have become beneficial, although the effect is rather marginal.

In the real-world networks evaluated in Section VI similar trends can be observed. For brevity only the energy fronts of VGG16 and DMCNN-VD are shown in Figure 13. Similarly to L3NetWide and L3Net 4k, layer fusion does result in new energy Pareto schedules, although the benefits are not as large as the reduction in external memory accesses in Figure 7 imply. This is in particular clear for the DMCNN-VD network, which has a significant reduction in DRAM accesses using layer-fusion (Figure 7g), but fails to capitalize on this in the energy front (Figure 13c). Furthermore, recomputation

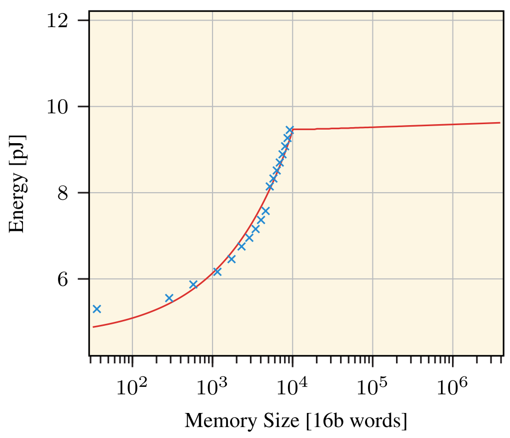


FIGURE 10. SRAM energy consumption based on commercial 40 nm SRAM modules at the typical corner, 25 °C, 1.1 V.

in the real-world networks does generally not result in a reduction in energy, as can be seen for VGG16 in Figure 13b.

The limited gains of fusion may seem counter-intuitive at first, in particular for the DMCNN-VD network which shows significant reduction in external memory accesses in Figure 7g. These reduced returns can easily be understood when inspecting the detailed energy breakdown in Figure 14 however, which lists the energy spent on multiply accumulates (E_{mac}), the register file (E_{rf}), the internal SRAM (E_{sram}), and the external DRAM (E_{dram}), for each point in the energy Pareto front. The reason the reduction in accesses does not result in a significant drop in energy consumption is effectively Amdahl's law applied to energy saving; As the scheduling techniques reduce the amount of accesses to the DRAM, the other parts of the system start to dominate. For the overall energy to improve even further, it makes most sense to address the E_{mac} and E_{rf} components by for example

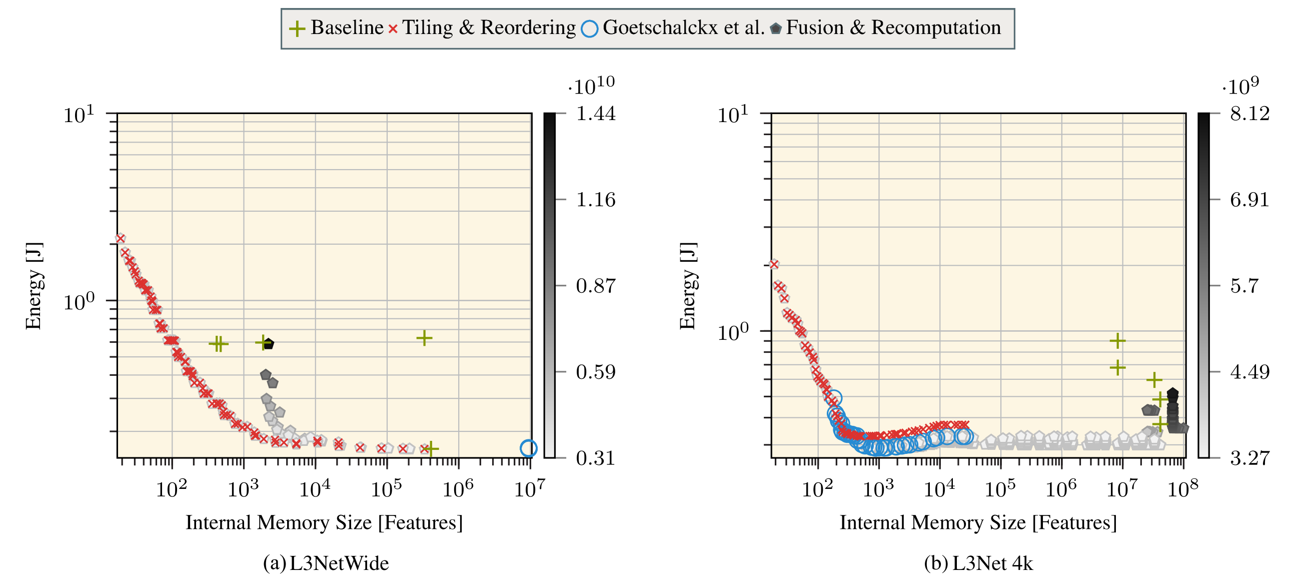


FIGURE 11. Energy front of L3Net based models using the multi-bank SRAM model outlined in Section VII-B. The colour maps only apply to Fusion & Recomputation, and represents the number of multiply-accumulate operations.

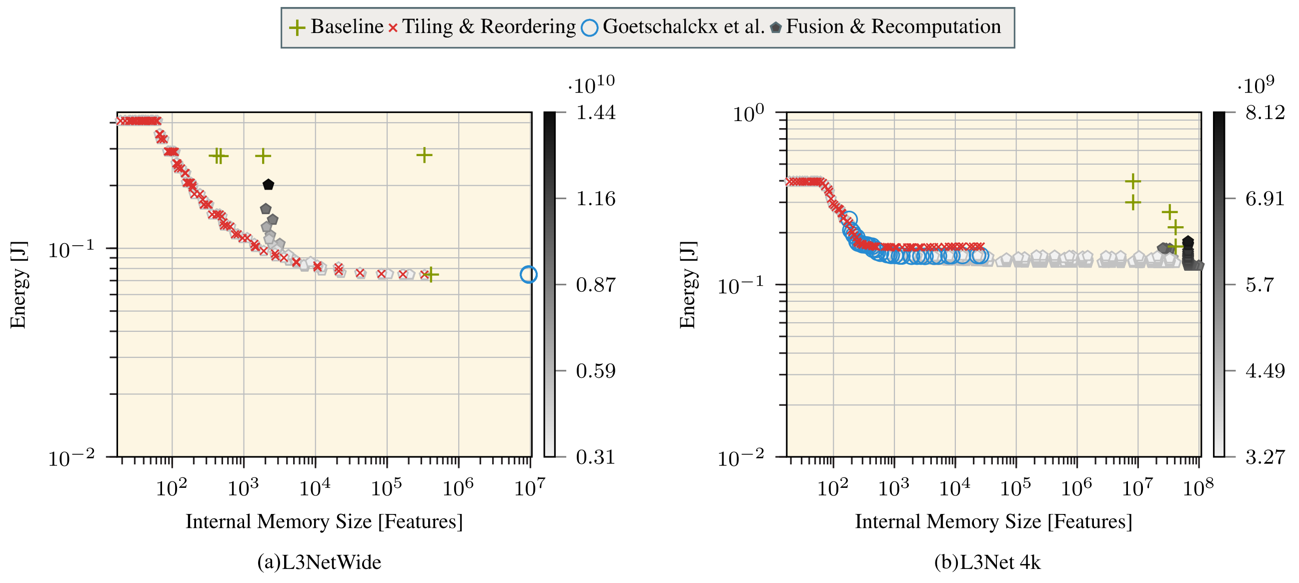


FIGURE 12. Energy front of L3Net based models using the multi-bank SRAM model outlined in Section VII-B and a 64 × 16*b* register file. The colour maps only apply to fusion & recomputation, and represents the number of multiply-accumulate operations.

chaining units to avoid intermediate accesses to the register file, dedicated accumulation registers, and quantization of the multiply-accumulate operands.

In conclusion the following statements can be made regarding the energy consumption of the explored schedules:

1) Compared to the baseline schedules, many schedules that require less internal memory can be found which drastically reduce the energy consumption.
2) Based on the energy evaluation a multi-banked and multi-level memory approach is required to benefit from the gains of fusion and recomputation, and even then energy gains are not always guaranteed.
3) The reduction in external accesses achieved by advanced scheduling techniques such as recomputation and fusion do not automatically translate in large improvements in energy, as other parts of the system become dominant in this region of the scheduling space for the investigated neural networks.

VIII. DISCUSSION & FUTURE WORK

By validation with instrumented Halide code, the introduced models are shown to be correct for the micro-benchmarks when the layer dimensions are exact multiples of the selected tile sizes. However, despite the low computational complexity of the models, complete traversal of the scheduling space is still infeasible for larger networks. This issue is discussed in further detail in Section VIII-A.

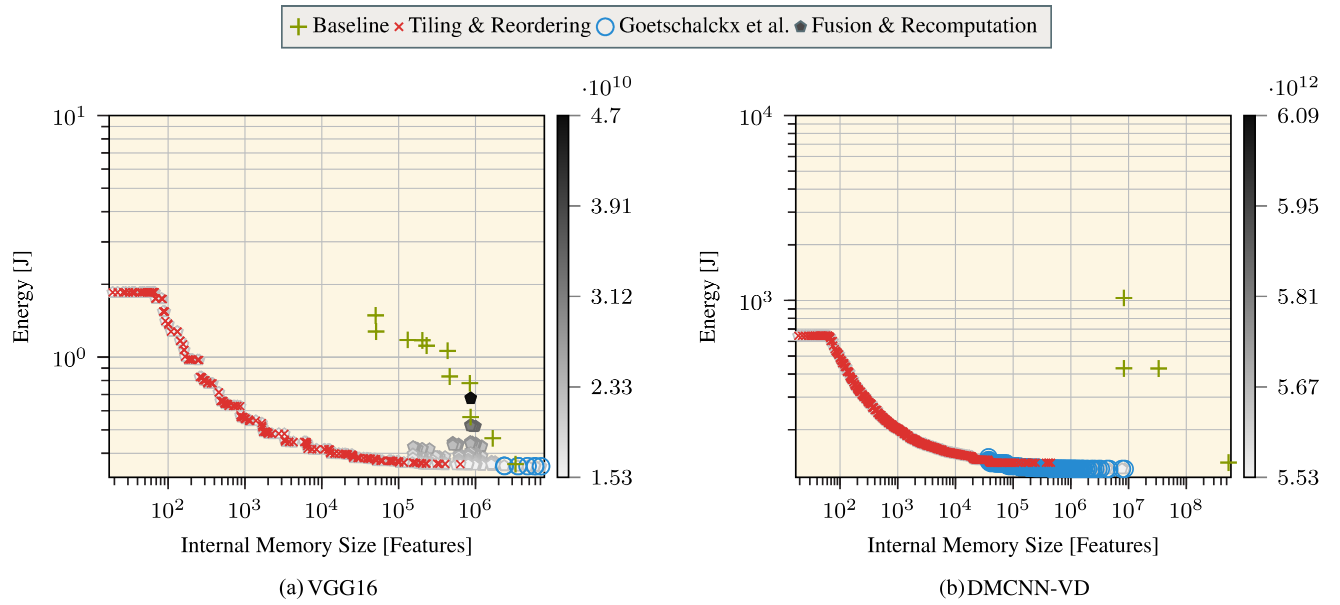


FIGURE 13. Energy fronts of VGG16 and DMCNN-VD. The colour maps only apply to fusion & recomputation, and represents the number of multiply-accumulate operations.

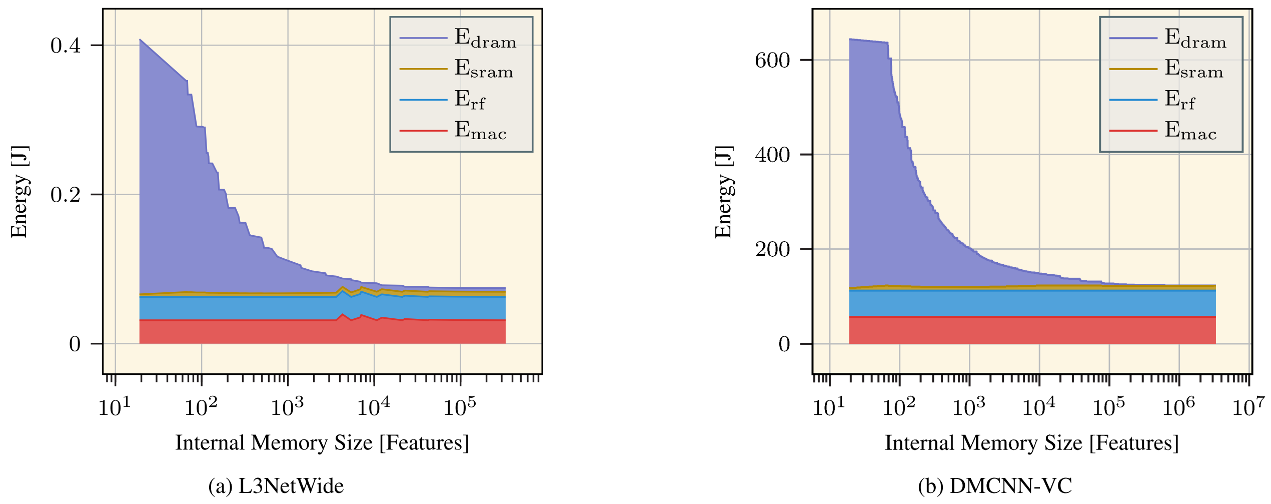


FIGURE 14. Detailed energy breakdown of energy Pareto front.

The models presented in this work are to a high degree hardware agnostic. A downside of this approach is that certain schedules that look beneficial using these models may be a bad fit on a particular target machine. Section VIII-B discusses approaches on how to adjust the design space to match with real machines.

Finally Section VIII-C discusses several scheduling space limitations of the current work, and possible ways to improve.

A. INTELLIGENT DESIGN SPACE EXPLORATION

For small networks the presented models are sufficiently fast to enable an exhaustive schedule space exploration. However, in particular for networks with larger eligible sequences,[3] the design space grows exponentially. This work does not provide a ready solution to this problem, but some suggestions can be made.

The presented models run in constant time per layer, so there is not much to be gained by simplifying the models. Multithreading could be added since the cost estimations are largely independent, but this will only provide a linear improvement against the exponential growth in workload. The provided open source tool [12] supports limiting the fusion depth, which can be used to mitigate the problem. As mentioned in Section VI, the exploration of the real world networks was limited to ten million schedules per segment. Still the search completed within a maximum of two days for the selected networks using a fairly unoptimized python implementation of the models, running with only a single thread. Since proper training of a neural network typically takes much longer, and selecting the schedule only needs to be

[3]See Section V-A1 for the precise definition of a sequence.

done once before deployment, this may an acceptable runtime for many practical cases. Based on the found scheduling spaces, it also seems unlikely a complete exploration would yield significantly better schedules.

Nonetheless, for particularly large networks, or when schedule quality is extremely important, this approach may not be desired. For those cases two suggestions can be made:

1) BRANCH & BOUND

When an upper and or lower limit for the internal memory size is known, a 'branch and bound' search strategy can be used to quickly eliminate large sections of the design space. For example, if a schedule with a certain tile size does not fit the available memory, the same schedule with a larger tile sizes will by extension also not fit. Thus, by placing restrictions on the memory size, it is possible to quickly eliminate large parts of the design space.

2) HEURISTICS

If the search space can not be sufficiently limited, it is recommended to integrate the presented models into a heuristic based search strategy. For example simulated annealing, genetic algorithms, or a greedy approach with handcrafted heuristics. Another interesting approach is to use an artificial intelligence driven search. Since the models are fast to evaluate, a large training set can be made relatively quickly.

B. TARGETING REAL HARDWARE

The models presented in this work are almost completely hardware agnostic, as the only assumption made is the presence of a two level memory hierarchy. Without further adaptation this may seem to adversely impact the applicability of the models to real hardware that supports features such as burst-transfers, and vectorisation. However, targeting such hardware can be done easily by restricting the complete scheduling space as described in Section III. The remainder of this section discusses how to restrict the space with respect to burst transfers, vectorisation, and how to extend the models to multiple memory levels.

1) BURST TRANSFERS

Burst transfers can amortize the addressing and synchronisation overhead of memory accesses when the data is accessed in larger consecutive blocks. Without adaptation the scheduling space will include schedules that access only one memory element per transfer, and accesses consecutive in time may not at all be consecutive in memory. These schedules may look promising based on number of transfers and required memory size, but due to their inability to benefit from burst transfers may in fact be worse than schedules that did not seem beneficial. The straightforward method of dealing with this is to exclude such non-beneficial schedules from the scheduling space. For example, if data is stored x-major in memory, then it makes sense to enforce loop `xi` of Code 2 to be inner to `yi` and `ii`. The compute level should then also be kept above `xi`, to ensure a chunk with size $\mathtt{T_x}$ will be scheduled for transfer. By limiting $\mathtt{T_x}$ to multiples of the burst size, it can be ensured that a block of continuous data is accessed every time.

2) VECTORISATION

Targeting hardware with vectorisation capabilities is identical to targeting burst transfers. By selecting a loop for vectorisation, and limiting the tile size to multiples of the vector width, only beneficial schedules will be selected. Note that, depending on the capabilities of the target hardware, vectorisation can be orthogonal to optimising for burst transfers. For burst transfers the data layout in memory matters, while this restriction may not matter for vectorisation, or a rearrangement of data upon load may be cheap. As such $\mathtt{T_x}$ could for example be limited to match with a memory block size, while $\mathtt{T_y}$ is vectorised. This way multiple hardware features can potentially be optimised orthogonally. Note that if this is not possible, it does not mean both can not be addressed. On most architectures one would expect that it is feasible to select a $\mathtt{T_x}$ that matches both with the burst size and the vectorisation support.

3) MULTI-LEVEL MEMORIES

As the energy evaluation in Section VII demonstrates, the benefits of fusion are limited by the simple two-level memory hierarchy, i.e., one internal scratchpad and one external main memory, captured in the models. In contrast to burst transfers and vectorisation, handling multi-level memory is not a matter of limiting the design space. However, it is possible to extend the models with multiple levels of tiling, and multiple store/compute levels. If the target memory system is hierarchical, i.e., each level progressively contains a subset of the data, then the models can be updated to contain an explicit copy action. Essentially each layer can be prefixed with a number of 'dummy' layers, which represent the data of a layer in each memory level. The production of such a layer is simply defined as a copy from the previous layer. In the memory hierarchy this copy represents a transfer, as such these dummy layers will be referred to as transfer layers. The models can use the available attributes for layer fusion to fuse all transfer layers into the layer they belong to. The production of these layers, which represents a data transfer from one level to another one, can then be taken into account in the model for different tile sizes. Of course it is also possible to skip a transfer layer altogether when layer fusion is used, such that intermediate data does not need to go through the entire memory system.

C. SCHEDULE SPACE LIMITATIONS

Although the presented models cover a vast design space, several limitations apply. In particular, only regular convolutional layers with striding are considered since most layers are of this type. However the models could be extended to cover different convolutional layers types, such as dense and depthwise convolution, and recurrent neural network layers. Dense layers can be modelled as a layer where the kernel spans the entire input, i.e., $\mathtt{D_m} = \mathtt{D_x}$ and $\mathtt{D_n} = \mathtt{D_y}$. For such dense layers it could pay off to also consider tiling of these kernels. Depthwise layers can be modelled by setting the number of input and output feature maps to one, i.e., $\mathtt{D_i} = \mathtt{D_z} = 1$, and multiplying the estimates by the number of original feature maps to compensate. Recurrent layers are slightly more complicated, since they use neuron states computed in a previous evaluation. This state could either be stored entirely in on-chip memory, or it could be transferred back and forth to external memory. In the latter case, it is probably possible to partly reuse the models for loading input data at the first layer in a fused section, and add them to the cost of an intermediate recurrent layer. The output state of such a recurrent layer always has to be transferred out, which potentially could be modelled by applying the output model to an intermediate recurrent layer.

Furthermore, fusion over skip, or residual, connections such as present in ResNet for example are not supported by the developed open source tool. Support could be provided by adding a binary choice to the network schedule per residual connection whether it should be stored in external or internal memory, analogous to the work by Goetschalckx and Verhelst [10].

Finally one optimisation not covered by the presented models is the option to keep part of a layer in internal memory while transitioning from one fused segment to the next. This technique is included in the work of Goetschalckx and Verhelst [10] as part of an optimistic model for schedules without fusion, but could be applied generically between fused segments as well. However, the resulting control code is likely quite complex, as the iteration order of consecutive segments needs to be reversed between layer transitions. For practical applications of this optimisation more research is required.

IX. CONCLUSION

In this work a practical scheduling space of convolution layers in CNNs has been outlined in Section III, including loop reordering, tiling, recomputation, and fusion. Generic models on this design space have been proposed in Section IV for required external memory accesses, internal buffer space, and multiply accumulates. An efficient schedule space traversal method has been described in Section V, and an embodiment of the proposed models and schedule space traversal method has been described and published as open source tool [12]. Using this tool the accuracy of the proposed models has been verified on synthetic networks using instrumented Halide code [3]. The effects of various scheduling techniques, i.e., loop reordering and tiling, layer fusion, and recomputation, have been evaluated on six real world neural networks in Section VI. An evaluation of the impact on energy consumption of these techniques has been provided in Section VII. Results show that the proposed models are accurate, and by covering both layer fusion and tiling of weights provide additional Pareto points compared the state of the art. This effect is most notable in networks which are weight dominated. To capitalize on the benefits of fused scheduling techniques multi-level memory appears to be required. Section VIII discusses how the presented models may be extended to accurately model multi-level memory. The models presented in this work are hardware agnostic, and suggestions have been made in Section VIII on how to limit the scheduling space to target real hardware that supports burst-accesses and vectorisation, increasing the applicability of the presented models.

REFERENCES

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, Nov. 2018, Art. no. e00938. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405844018332067

[2] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2020, pp. 1–12.

[3] J. Ragan-Kelley, "Decoupling algorithms from the organization of computation for high performance image processing," Ph.D. Thesis, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, Jun. 2014. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2014/jrkthesis.pdf

[4] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, doi: 10.1145/216585.216588.

[5] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan. 2016, pp. 262–263.

[6] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 343–348.

[7] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. MICRO*, Oct. 2016, pp. 1–12.

[8] G. Li, F. Li, T. Zhao, and J. Cheng, "Block convolution: Towards memory-efficient inference of large-scale CNNs on FPGA," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1163–1166.

[9] N. Weber, F. Schmidt, M. Niepert, and F. Huici, "Brainslug: Transparent acceleration of deep learning through depth-first parallelism," NEC Lab. Eur., Syst. Mach. Learn. Group, Tech. Rep., 2018. [Online]. Available: https://dblp.uni-trier.de/rec/journals/corr/abs-1804-08378.html?view=bibtex

[10] K. Goetschalckx and M. Verhelst, "Breaking high-resolution CNN bandwidth barriers with enhanced depth-first execution," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 323–331, Jun. 2019.

[11] F. Chollet. (2015). *Keras*. [Online]. Available: https://keras.io

[12] L. Waeijen. *ConvFuser*. Accessed: Dec. 19, 2021. [Online]. Available: https://gitlab.com/lwaeijen/convfusion

[13] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Proc. 22nd Int. Joint Conf. Artif. Intell. (IJCAI)*, vol. 2, 2011, pp. 1237–1242.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, vol. 1, 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[15] M. Peemen, B. Mesman, and H. Corporaal, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2015, pp. 169–174. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755790

[16] L. Waeijen, S. Sioutas, Y. He, M. Peemen, and H. Corporaal, *Automatic Memory-Efficient Scheduling of CNNs*. Cham, Switzerland: Springer, Aug. 2019, pp. 387–400.

[17] L. Mei, P. Houshmand, V. Jain, J. S. P. Giraldo, and M. Verhelst, "ZigZag: A memory-centric rapid DNN accelerator design space exploration framework," 2020, *arXiv:2007.11360*.

[18] J. IJzerman, T. Viitanen, P. Jääskeläinen, H. Kultala, L. Lehtonen, M. Peemen, H. Corporaal, and J. Takala, "AivoTTA: An energy efficient programmable accelerator for CNN-based object recognition," in *Proc. 18th Int. Conf. Embedded Comput. Syst., Archit., Modeling, Simulation*, Jul. 2018, pp. 28–37, doi: 10.1145/3229631.3229637.

[19] S. Sioutas, S. Stuijk, T. Basten, H. Corporaal, and L. Somers, "Schedule synthesis for halide pipelines on GPUs," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, pp. 1–25, Aug. 2020, doi: 10.1145/3406117.

[20] Z. Zheng, P. Zhao, G. Long, F. Zhu, K. Zhu, W. Zhao, L. Diao, J. Yang, and W. Lin, "Fusionstitching: Boosting memory intensive computations for deep learning workloads," Alibaba, Tech. Rep., 2020. [Online]. Available: https://dblp.uni-trier.de/rec/journals/corr/abs-1811-05213.html?view=bibtex

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[22] N.-S. Syu, Y.-S. Chen, and Y.-Y. Chuang, "Learning deep convolutional networks for demosaicing," Dept. Comput. Sci. Inf. Eng., Nat. Taiwan Univ., Tech. Rep., 2018. [Online]. Available: https://dblp.uni-trier.de/rec/journals/corr/abs-1802-03769.html?view=bibtex

[23] Y. N. Wu, A. A. Ghasemazar, and P.-A. Tsai. *Accelergy-Aladdin-Plug-in*. Accessed: Oct. 14, 2021. [Online]. Available: https://github.com/Accelergy-Project/accelergy-aladdin-plug-in

[24] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.

[25] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *Proc. 39th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2012, pp. 37–48.

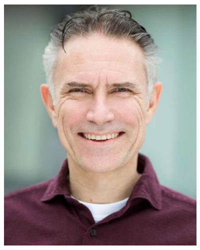
[26] S. Mai, C. Zhang, Y. Zhao, J. Chao, and Z. Wang, "An application-specific memory partitioning method for low power," in *Proc. 7th Int. Conf. ASIC*, Oct. 2007, pp. 221–224.

[27] Y. N. Wu, J. S. Emer, and V. Sze. *Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs Slides*. [Online]. Available: http://accelergy.mit.edu/slides.pdf

LUC WAEIJEN received the B.Sc. degree in electrical engineering and the M.Sc. degree in embedded systems from the Eindhoven University of Technology (TU/e), in 2012 and 2013, respectively, where he is currently pursuing the Ph.D. degree with the Electronics Systems Group. He is with the Architecture Group, GrAI Matter Labs, driving the design and specification of neuromorphic processors. His research interests include energy efficient computer architecture, neuromorphic/brain-inspired computing, and memory-centric scheduling techniques.

SAVVAS SIOUTAS received the Diploma degree in electrical and computer engineering, with a specialization in electronics and computers, from the University of Patras, Greece, and the Ph.D. degree from the Eindhoven University of Technology, The Netherlands. He is currently working as a Software Engineer in a high-tech company. His research interests include compiler engineering, code optimization, and analytical modeling of embedded computer architectures.

MAURICE PEEMEN received the M.Sc. and Ph.D. degrees in electrical engineering from the Eindhoven University of Technology, The Netherlands, in 2011 and 2017, respectively. During his Ph.D. degree, his main research topics were efficiency improvements for deep convolutional networks by algorithmic changes, data access optimizations, custom accelerators, and optimizing compilers. In 2015, he started as a Research Scientist at Thermo Fisher Scientific (formerly FEI Company) to work on high-performance microscopy workflow solutions.



MENNO LINDWER received the M.Sc. degree from the University of Twente and the P.D.Eng. degree from the Eindhoven University of Technology, in 1993.

He is leading GrAI Matter Labs (GML)'s Hardware Development Department, which includes developing neuromorphic and neural network processors. Prior to joining GML, he was a Processor Technology Architect, the Engineering Manager, and the Program Manager at Intel. Since then, he studied asynchronous devices, database matching technology, 3D graphics renderers, media processor design, and AI processors. In 1995, he joined Philips Research, working on the design of Java and media processors. In 2002, he was part of the team that started up Silicon Hive. He was the Director of product management for Silicon Hive's HiveLogic parallel processing platform. Silicon Hive was acquired by Intel, in February 2011. At Intel, he led teams that worked on processor construction tools and optimizing auto-targeting compilers. Also, he led Intel's activities in several EU projects. His work has led to over 30 scientific articles and patent publications (seven patents currently assigned in the USA). He coauthored several articles on ambient intelligence at DATE, on media processing at IEEE ISM and GSPX, on compiler technology at IJPP, and on design space exploration for multi-ASIP systems. His research interests include low-power computing systems, neuromorphic and neural network processors, and spatial compilers.

Dr. Lindwer served on the executive committee and technical program committees of the DATE Conference.

HENK CORPORAAL received the M.Sc. degree in theoretical physics from the University of Groningen and the Ph.D. degree in electrical engineering, in the area of computer architecture, from the Delft University of Technology.

He is a Professor in embedded system architectures at the Eindhoven University of Technology (TU/e), The Netherlands. He has coauthored over 500 journals and conference papers. Furthermore, he invented a new class of VLIW architectures, the transport triggered architectures, which is used in several commercial products and by many research groups. His research interests include low power multi-processor, heterogenous processing architectures, their programmability, and the predictable design of soft real-time systems and hard real-time systems. This includes research and design of embedded systems architectures, including CGRAs, SIMD, VLIW, and GPUs, on accelerators, the exploitation of all kinds of parallelism, fault-tolerance, approximate computing, architectures for machine and deep learning, optimizations and mapping of deep learning networks, and the (semi-)automated mapping of applications to these architectures. For further details, see his personal website (`corporaal.org`).