Sunstone: A Scalable and Versatile Scheduler for Mapping Tensor Algebra on Spatial Accelerators

MohammadHossein Olyaiy*, Christopher Ng*, Alexandra (Sasha) Fedorova, Mieszko Lis
The University of British Columbia
{mohamadol, chris.ng, sasha, mieszko}@ece.ubc.ca

***Abstract*—Tensor algebra, the main component of several popular machine learning techniques, benefits from modern accelerators due to the massive parallelism and data reuse available. To achieve the benefits, however, optimizing the dataflow is crucial: prior works showed that 19× energy savings are possible by tuning the dataflow. This optimization is challenging because: (1) the optimization space for modern chip architectures with several levels of memory and multiple levels of spatial processing is vast, and (2) distinct tensor computations follow different memory access and reuse patterns.**

In this manuscript, we algebraically analyze the possible reuse when executing tensor workloads on an accelerator. Based on our analysis, we develop several principles that significantly reduce the dataflow optimization space even for modern, complex chip architectures. Moreover, these principles are transferable to various tensor workloads with different memory access patterns.

Compared to prior work, our techniques can find dataflow for typical tensor workloads up to 800× faster and with up to 1.9× better energy-delay products.

***Index Terms*—dataflow computing; accelerator architectures; scheduling algorithms; neural network hardware; parallel processing;**

I. Introduction

Both modern and classic machine learning (ML) techniques rely heavily on tensor algebra. Some common examples include convolution for computer vision [34, 45, 50], tensor factorization for social networks [13, 48], compressing neural networks [29, 38], and embedding generation for recommender systems [6, 19]. Since the computation pattern of tensor algebra is predictable, and because of the massive parallelism and the ample data reuse available, several dedicated tensor-compute accelerators have been proposed [10, 20, 23, 30, 44, 46, 53, 61, 67, 69]. Most of these comprise array(s) of processing elements (PEs) arranged in a 1D or 2D grid, paired with a multi-level memory hierarchy [10, 20, 23, 30, 44, 46, 67].

While specialized tensor-compute accelerators achieve better performance and efficiency over general-purpose CPUs and GPUs [26, 30], a key challenge is how to *schedule* a tensor computation on the accelerator. This problem, called dataflow optimization, significantly impacts performance [7, 10, 43, 68]. For example, Timeloop has shown that better dataflows can make inference 19× more energy efficient.

The reason dataflow optimization is challenging is twofold. First, the search space is non-convex, non-differentiable, and astronomically large. For example, temporal and spatial tiling choices, as well as interchanging independent loops, yield on the order of 10^{19} solutions even for executing *one* convolution (conv) layer on a conventional accelerator [7]. This problem becomes even more severe as recent state-of-the-art (SOTA) accelerator designs include more memory levels and multiple levels of parallel processing primitives on a single chip [3, 12, 46, 60], which exponentially increases the combination of potential spatiotemporal tilings and loop orderings in each level. Ideally, dataflow optimization techniques should be *scalable* and still find optimized dataflow in a reasonable time as accelerators include more levels in their architecture.

Although several proposals have been made for scheduling workloads on tensor-compute accelerators, many of these [14, 43, 68] have been designed for an accelerator with 2 or 3 levels of memory and only a single level of spatial processing (e.g., a flat PE grid). However, recent research from the architecture community shows that having more memory and parallel processing levels (Fig. 1b) results in more efficient hardware [12, 46, 60]: for example, in MAGNet [60] increasing the size of vector computation unit inside the PE improves the energy efficiency due to the more spatial reuse available. Meanwhile, Simba [46] showed that small registers with low access energy inside the vector MACs could facilitate the reuse of some operands over several MAC operations.

The fixed-number-of-levels assumption means that prior schedulers either do not support these modern, more efficient architectures [14, 68] or become extremely slow and inefficient [43] when applied to them; this inefficiency usually leads to early termination of the search and a suboptimal solution. When considering the more recent architectures such as [46], Timeloop is often the only available mapper, and it also can only be invoked if the user significantly constrains the search space (Section V), leading to suboptimal mappings.

To achieve scalable optimization, we must dramatically shrink the optimization space. Based on the observation that performance requires maximizing data reuse [10, 43], we analyze mathematical equations that represent reuse, and based on this analysis, propose a principled approach to mapping search.

Our **key observation** is that during the dataflow optimization of a single memory or parallel processing level, only some problem dimensions improve the reuse — and therefore only those need to be considered. This means fewer optimization space dimensions to consider at each level without losing the ability to discover optimal solutions — and, consequently, scalability as more levels of memory and spatial processing

*These authors contributed equally.

TABLE I: Number of dimensions that construct the space under optimization for each tool and the size of this space for an example convolution layer. Since some tools rely on empirical-based heuristics to reduce the optimization space, they can often find suboptimal or invalid mappings.

	Timeloop [43]	CoSA [28]	Marvel [7]	Interstellar [68]	dMazeRunner [14]	**ours**
dimensions to build each temporal level tile	all the dimensions (7)					**only the reuse dimensions (4)**
dimensions to unroll at each spatial level	all the dimensions (7)			input and output channel dimensions (2)	dimensions that do not need spatial reduction (4)	**only the reuse dimensions (4)**
pruning methods to reduce the space	nothing	linear approximation to use linear optimizers	decoupled off-chip and on-chip, high buffer utilization	high throughput	high buffer utilization, high throughput	**alpha-beta, high throughput**
estimated space size for an Inception V3 example layer	3.69×10^{10}	similar to Timeloop, but blackbox optimizer might prune it	1.36×10^{9}	1.40×10^{9}	1.97×10^{5}	$\mathbf{5.89 \times 10^{3}}$
worse mappings than other tools?	yes, up to 1.9× worse EDP	yes, up to 1.5× worse EDP	not open source	yes, up to 1.4× worse EDP	yes, up to 1.1× worse EDP	**no**
invalid mappings?	no	yes, 60% of the time	not open source	yes, 10% of the time	yes, 30% of the time	**no**

are added.

Table I compares the characteristics of the optimization process of prior tools with our method. Observe that all the prior optimizers use *all seven dimensions* to create the optimization space for each temporal tiling level, and most of them do the same for each spatial unrolling level, too. Our observations, in contrast, help us identify only the necessary dimensions at each temporal or spatial level (4 dimensions for the convolution example in Table I). As a result, the search space is *much smaller* than what prior tools are optimizing — as much as $10^7\times$ smaller.

The second challenge behind dataflow optimization is that computation and memory access patterns, and therefore the reuse behavior, of various tensor applications can vary significantly [53, 66]. Ideally, a dataflow optimizer should automatically infer the reuse characteristics of a given workload to support a broad range of tensor computations. We refer to this as the *versatility* of the optimizer.

Unfortunately, most prior dataflow optimization proposals [7, 14, 28, 40, 68] are designed for a specific tensor operation, such as convolution. As a result, they do not work on other important tensor computations, such as matricized tensor times Khatri-Rao product (MTTKRP) [51] or tensor-times-matrix chain (TTMc) [2], used in various fields, including neural network compression [21, 29, 38] and recommender systems [6, 19].

In contrast, our observations for reducing the search space are based on algebra rather than workload-specific heuristics. They can be applied to several tensor workloads, including MTTKRP, TTMc, and convolution. As a proof-of-concept, we developed Sunstone, a scheduler that implements our algebra-derived optimization techniques using novel intermediate representations (IRs) for each search stage. As we will demonstrate in Section V, Sunstone is up to 800× faster than the prior schedulers, covers many tensor workloads, and scales to modern hierarchical architectures. Our key contributions are:

- algebra-based dataflow optimization techniques that not only are transferable to various types of tensor algebra workloads but also significantly reduce the optimization space for them without rejecting good solutions;
- access and compute IRs that help to automatically analyze reuse information, remove search space dimensions, and optimize the dataflow;
- an open source, proof-of-concept optimizer available in our GitHub repository[†].

II. Background

A. DNN accelerator architecture

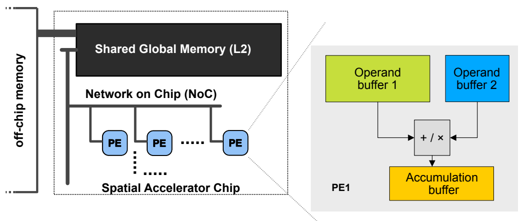
Fig. 1a shows the architecture of a conventional DNN accelerator [1, 10, 11, 17, 23, 44, 67, etc.] implemented as a 2D array of processing elements (PEs). Each PE has a multiply-and-accumulate (MAC) functional unit and local (L1-level) memories; these either consist of separate memories for each datatype (i.e., ifmap, weights, and ofmap) or are unified memories that store all three datatypes. L1 is commonly double-buffered to overlap computation and memory refill.

The PE of a modern accelerator, shown in Fig. 1b, is different from a conventional one. Specifically, instead of a single MAC unit, a Simba-like [46] PE has a row of vector MACs and registers. Furthermore, the PE has a distributed buffer that can supply different operands to each of the vector MACs, and a broadcast buffer that broadcasts to all the units. Lastly, each of the registers in a vector MAC reuses one of the MAC's operands temporally over several MAC operations.

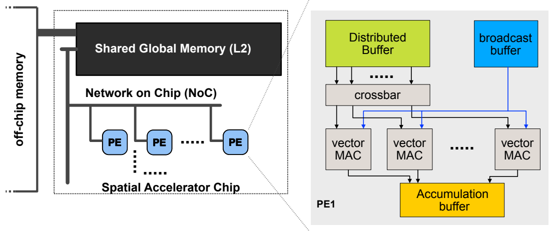
Accelerators also commonly include a larger memory shared among the PEs (L2-level) and a large off-chip DRAM. The PEs and L2 are typically interconnected via a simple on-chip interconnect per datatype [10].

This architecture supports three kinds of reuse. First, operands may be reused *spatially*: that is, broadcast to a subset of (or all) PEs or, within the PEs shown in Fig. 1b, broadcast to all the vector MACs. L1 memories support short-term *temporal* reuse of operands *within* each PE; finally, L2 memories support temporal reuse *across* multiple PEs.

[†]`https://github.com/compstruct/sunstone`



(a) A conventional accelerator with one level of spatial processing.



(b) A modern accelerator with multiple levels of spatial processing.

Fig. 1: Spatial accelerator architectures

B. Tensor algebra workloads

We target tensor computations that consist of nested loops with no inter-loop dependencies, i.e., loops that can be freely reordered. The computations may include sliding-window access patterns, as found in, e.g., convolution operations. These workloads span a range of real-life problems, such as convolution layers and fully connected layers for neural networks, *MTTKRP* [51] and *TTMc* [2] (bottleneck kernels in tensor decompositions), and various tensor contraction workloads that permeate the optimization domain [16, 33, 35, 49, 54, 58]. Table II shows examples of the tensor algebra we target, their distinct memory access patterns, and diverse applications.

C. Dataflow mapping

Dataflow mapping consists of tiling, loop reordering, and spatial unrolling. To explain each with a concrete example, we consider the convolution of K 1D filters of length R with 1D input feature map (*ifmap*) of size P to generate K output feature maps (*ofmap*) of size P:

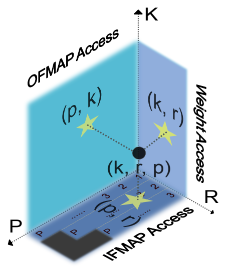
$$\mathit{ofmap}[k,p] = \sum_r \mathit{ifmap}[p+r] \times \mathit{weight}[k,r].$$

Typically, this is expressed as a nested loop [43] similar to Algorithm 1:

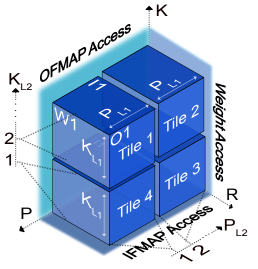
Algorithm 1 1D convolution algorithm

```
1: for $k \leftarrow [0, K)$ do
2:     for $p \leftarrow [0, P)$ do
3:         $\mathit{ofmap}[k,p] \leftarrow 0$
4:         for $r \leftarrow [0, R)$ do
5:             $\mathit{ofmap}[k,p] \mathrel{+}= \mathit{ifmap}[p+r] \times \mathit{weight}[k,r]$
```

This defines a 3D *operation space* of $K \times P \times R$ MAC operations shown in Fig. 2a, where the operands for each MAC can be obtained by projecting its point in the operation space onto the "walls." The walls thus correspond to the accessed elements of the *ifmap*, *ofmap*, and *weight* tensors.



(a) operation space



(b) operation space tiling

Fig. 2: Operation space of 1D convolution. *Ifmap* is shifted, replicated, and 0-padded along the sliding window dimension R. After tiling, P dimension is divided into P_{L2} tiles of size P_{L1}, and K dimension into K_{L2} tiles of size K_{L1}.

Tiling. The per-PE (L1) memories are far too small to contain the entire *ifmap*, *ofmap*, and *weight* tensors. Thus, the operation space must be *tiled* into *L1 tiles* with memory footprints that fit in the L1 memories to support temporal reuse.

Fig. 2b shows this for the running convolution example. The volume of each tile shows the MAC operations performed in this tile, while the surfaces *W1*, *O1*, and *I1* correspond to the regions of the *weight*, *ofmap*, and *ifmap* accessed. If these are stored in L1 memories, they can be reused: e.g., *W1* can be temporally reused across the P extent of the tile.

This corresponds to the pseudocode of Algorithm 2, where the K dimension is divided into K_{L2} equal tiles of size K_{L1}, and the P dimension into P_{L2} equal tiles of size P_{L1}:

Algorithm 2 a 2-level tiled 1D convolution algorithm

```
1: for $k_2 \leftarrow [0, K_{L2})$ do                              L2 tile
2:     for $p_2 \leftarrow [0, P_{L2})$ do
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
3:         for $k_1 \leftarrow [0, K_{L1})$ do                      L1 tile
4:             for $p_1 \leftarrow [0, P_{L1})$ do
5:                 $k \leftarrow k_2 \times K_{L1} + k_1$
6:                 $p \leftarrow p_2 \times P_{L1} + p_1$
7:                 $\mathit{ofmap}[k,p] \leftarrow 0$
8:                 for $r \leftarrow [0, R)$ do
9:                     $\mathit{ofmap}[k,p] \mathrel{+}= \mathit{ifmap}[p+r] \times \mathit{weight}[k,r]$
```

Loop reordering. In addition to the intra-tile reuse described above, some tensor regions can be further temporally reused over multiple tiles. In Fig. 2b, for example, region *W1* can remain in L1 if tile 2 is processed in the same PE right after tile 1 (as in Algorithm 2).

We can control what is reused between tiles by changing the tile traversal order. We will write orders by listing loop bounds outermost-to-innermost, so the pseudocode above is $K_{L2}P_{L2}K_{L1}P_{L1}R$. If we swap lines 1 and 2 (order $P_{L2}K_{L2}K_{L1}P_{L1}R$), tile 4 will be processed right after tile 1, reusing the *I1* region of *ifmap* in L1.

Spatial unrolling. Finally, loops can be *spatially* unrolled so that different tiles are assigned to different PEs. For example, in Fig. 3 the K dimension is unrolled spatially across two PEs, so tiles 1 and 4 will be computed by PEs 1 and 2 in the first step, and then tiles 2 and 3 in the next. This allows inter-tile *spatial reuse*:

TABLE II: Representative tensor computations

Workload	Algebraic Definition	Application	Application Instance
Conv	$\mathit{ofmap}[p,q,k,n] = \sum_{c,r,s} (\mathit{ifmap}[p+r,q+s,c,n] \times w[r,s,c,k])$	CNN	ResNet [24], Inception-v3 [55]
MTTKRP	$\mathit{out}[i,j] = \sum_{l,k} (A[i,k,l] \times B[k,j] \times C[l,j])$	CP-Decomposition	nell2 [52], netflix [52], poisson1 [52]
SDDMM	$\mathit{out}[i,j] = A[i,j] \times \sum_{k} (B[i,k] \times C[k,j])$	Alternating Least Squares	bcsstk17 [15], cant [15]
TTMc	$\mathit{out}[i,l,m] = \sum_{k,j} (A[i,j,k] \times B[j,l] \times C[k,m])$	Tucker Decomposition	nell2 [52], netflix [52], poisson1 [52]
MMc	$\mathit{out}[i,l] = \sum_{k,j} (A[i,j] \times B[j,k] \times C[k,l])$	NLP (Transformers)	Attention Model [59]
TCL	$\mathit{out}[l,m,n] = \sum_{k,j,i} (A[i,j,k] \times B[i,l] \times C[j,m] \times D[k,n])$	Tensor Contraction Layer [33]	AlexNet [34], VGG [50]

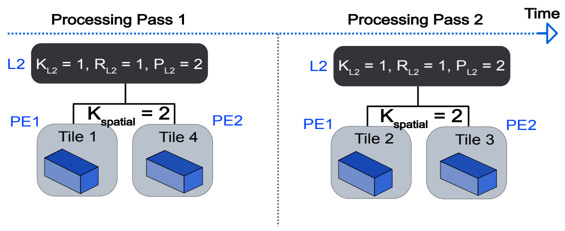


Fig. 3: Processing the tiles of Fig. 2b in parallel. K dimension is spatially unrolled, so both PEs need the same *ifmap* data but distinct *weights* and *ofmaps*.

each pair of tiles processed concurrently by the PEs accesses the same region of the *ifmap*, which can be broadcast to both PEs.

Considering all combinations of tiling, traversal order, and unrolling results in an enormous search space for tensor computations: for example, Table I shows that the search space for an Inception V3 layer can be on the order of 10^{10} possibilities.

D. Loop ordering observations from prior work

Prior works [14, 40] have thoroughly analyzed the potential loop orderings and identified principles for choosing a subset that would still lead to optimal reuse. We also use their analysis to reduce the possible loop orderings in dataflow optimization. To summarize their observations, we make the 1D convolution example of the previous section slightly more complex, adding an extra dimension of C input channels to it. A two-level, tiled version of this algorithm is shown in Algorithm 3.

Algorithm 3 a 2-level tiled 1D convolution algorithm with multiple input (C) and output (K) channels

```
1: for $k_2 \leftarrow [0, K_{L2})$ do
2:     for $p_2 \leftarrow [0, P_{L2})$ do
3:         for $c_2 \leftarrow [0, C_{L2})$ do
4:             for $r_2 \leftarrow [0, R_{L2})$ do                L2 tile
-----------------------------------------------------------------------
5:                 L1 tile computation                           L1 tile
```

The prior-work observations can be summarized as follows: **Ordering Principle 1.** A loop that iterates over a dimension not used to index a tensor (i.e., non-indexing) can reuse that tensor [10, 14, 40]. For example, in Algorithm 3, dimensions R and C are not used to index *ofmap*. Therefore the two innermost loops (line 4 and line 3) lead to the reuse of the *ofmap* tensor.
Ordering Principle 2. For a loop to reuse the tensor it does not index, it must either be innermost, or the loops inside it must be limited to the other non-indexing dimensions of that same tensor [14, 40]. It can be observed from Algorithm 3 that even though K is a non-indexing dimension of *ifmap*, in this loop order *ifmap* actually cannot be reused across K. This is because the loop that iterates over C (line 3) is inside the K loop (line 1) and replaces the *ifmap* tensor multiple times within each iteration of the K loop.
Ordering Principle 3. Only a subset of the loops — precisely, the innermost loops that reuse the same tensor — determine the reuse, and hence only the ordering of those loops needs to be optimized [14]. In Algorithm 3, although R and C loops lead to the reuse of the *ofmap* tensor, reordering the loops *above* them (i.e., line 1 and line 2) does not change the number of accesses to any of the tensors.

III. Tiling and Unrolling Principles

In this section, we show how to take advantage of properties that span *across* memory levels to reduce the search space at each level significantly. Throughout the discussion, we will use the same 1D convolution example of Section II-D with C input channels, K filters, and R and P as the filter and feature map widths respectively. The observations we make, however, are generalizable to other tensor workloads, as we will demonstrate in our evaluations in Section V.

A. Tiling

First, we will show why only a subset of problem dimensions need to be considered for tiling, thus reducing the optimization space. We will also explain how these dimensions can be selected based on the tensor access pattern and upper-level memory loop order. We rely on algebraic analysis of memory access equations in a tiled dataflow to do this. For clarity, we start by considering the L1 tile configuration of a 2-level tiled dataflow with a specific L2 loop ordering that executes the running 1D convolution example. This is shown in Algorithm 4.

Algorithm 4 a 2-level tiled 1D convolution dataflow

```
1: for $p_2 \leftarrow [0, P_{L2})$ do
2:     for $k_2 \leftarrow [0, K_{L2})$ do
3:         for $c_2 \leftarrow [0, C_{L2})$ do                    L2 tile
-----------------------------------------------------------------------
4:             for $p_1 \leftarrow [0, P_{L1})$ do                L1 tile
5:                 for $k_1 \leftarrow [0, K_{L1})$ do
6:                     for $c_1 \leftarrow [0, C_{L1})$ do
7:                         for $r_1 \leftarrow [0, R)$ do
8:                             computation
```

Here, the L1 tile sizes are $P_{L1} \times K_{L1}$ for *ofmap*, $C_{L1} \times K_{L1} \times R$ for *weight*, and $(P_{L1} + R - 1) \times C_{L1}$ for *ifmap*. Furthermore, the total number of L1 tile iterations is the product of L2 factors,

$P_{L2} \times K_{L2} \times C_{L2}$. Thus, to execute the full workload, the total number of L2 memory accesses would be $\#passes \times tile\_size$, broken down as:

$$\begin{aligned} \mathit{ifmap} : &K_{L2} \times P_{L2} \times C_{L2}(P_{L1} + R - 1) \times C_{L1} \\ &= K_{L2} \times C \times P_{L2}(P_{L1} + R - 1) \end{aligned} \tag{1}$$

$$\begin{aligned} \mathit{weight} : &K_{L2} \times P_{L2} \times C_{L2}(C_{L1} \times K_{L1} \times R) \\ &= \underbrace{C \times K \times R}_{\text{problem dimensions}} \times P_{L2} \end{aligned} \tag{2}$$

$$\begin{aligned} \mathit{ofmap} : &K_{L2} \times P_{L2} \times \cancelto{\text{reused}}{C_{L2}}(P_{L1} \times K_{L1}) \\ &= P \times K \times \cancelto{\text{reused}}{C_{L2}} = P \times K \end{aligned} \tag{3}$$

For the specified loop ordering, *ofmap* is reused C_{L2} times — that is, *ofmap* remains in L1 between L1 tiles — because C is the innermost L2 loop. The total L2 access count is the sum of Equations 1 to 3:

$$\begin{aligned} \text{L2 accesses} = &K_{L2} \times C \times P_{L2}(P_{L1} + R - 1) \\ &+ C \times K \times R \times P_{L2} + P \times K \end{aligned} \tag{4}$$

For best L1 reuse, our task is to minimize this under the constraint that the L1 tiles of all data types fit in the L1 memories.

What are the degrees of freedom here? Equations 1 to 3 involve either full problem dimensions (e.g., C, K and R in Equation 2) — which we cannot change — or loop bounds (e.g., P_{L2} in Equation 2) — which we can select to change L1 tile dimensions. For example, the *ofmap* access count $P \times K$ only includes full problem dimensions (Equation 3), so we cannot change it by altering L1 tile dimensions. Our options are, therefore, to decrease K_{L2} or P_{L2} to minimize *ifmap* and *weight* fetches.

Now, consider two configurations where (a) $K_{L2} = 2$ or (b) $K_{L2} = 3$, and no other loop bounds change. If both (a) and (b) fit in the L1 memories, then (a) offers strictly more reuse (and lower energy) since there are fewer *ifmap* and *weight* fetches.

Note that K and P, the dimensions of interest, are indexing dimensions for *ofmap*, the tensor being reused across L1 tiles in Algorithm 4. In general:

Tiling Principle

For any given tile T and operand OP that is being reused across the tiles, if any of the indexing dimensions for OP can be enlarged in the tile while still fitting in the respective memory, the larger tile leads to fewer data accesses to the upper-level memory; therefore T can be pruned.

Appendix A provides a more abstract discussion regarding the tiling principle.

The tiling principle helps us reduce the potential tiling options for each of the upper-level orderings when optimizing the dataflow at a specific memory level (e.g., L1), and yields a set of ordering-tiling candidates. Applying this observation reduces the L1 tile search space for ResNet-18 [24] CONV layers by up to 80%.

Algorithm 5 a tiled 1D convolution with a shared L2 memory, several parallel processing units each with their dedicated L1 memory.

```
for $k_2 \leftarrow [0, K_{L2})$ do                                   L2 tile
    for $p_2 \leftarrow [0, P_{L2})$ do
        for $c_2 \leftarrow [0, C_{L2})$ do
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            for $k_{spatial} \leftarrow [0, K_{spatial})$ do           spatial
                for $p_{spatial} \leftarrow [0, P_{spatial})$ do
                    for $c_{spatial} \leftarrow [0, C_{spatial})$ do
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                        for $k_1 \leftarrow [0, K_{L1})$ do            L1 tile
                            for $p_1 \leftarrow [0, P_{L1})$ do
                                for $c_1 \leftarrow [0, C_{L1})$ do
                                    for $r \leftarrow [0, R)$ do
                                        compute
```

B. Spatial unrolling

When processing units are arranged in SIMD fashion — such as the vector MACs of the Simba-like architecture of Fig. 1b — spatially unrolling the loops can reduce the number of upper memory accesses by broadcasting the data that all the units need. We again use algebraic analysis to identify a subset of the dimensions, which unrolling them increases the spatial reuse.

We continue the running example and consider, as the spatial level, parallel PEs before the L2 memory; this adds a spatial unrolling factor in each dimension as shown in Algorithm 5, so that, for example, $P = P_{L2} \times P_{spatial} \times P_{L1}$. We assume we are looking for unrolling candidates for a specific ordering-tiling pair from all the pairs found in the previous section. With that, we analyze the L2 access equations to maximize the spatial reuse:

$$\begin{aligned} \mathit{ifmap} : &K_{L2} \times P_{L2} \times C_{L2}((P_{spatial} \times P_{L1} + R - 1) \\ &\times C_{spatial} \times C_{L1}) = K_{L2} \times C \times P_{L2}(P_{L1} + R - 1) \end{aligned} \tag{5}$$

$$\begin{aligned} \mathit{weight} : &K_{L2} \times P_{L2} \times C_{L2}(C_{spatial} \times C_{L1} \times K_{spatial} \\ &\times K_{L1} \times R) = C \times K \times R \times P_{L2} \end{aligned} \tag{6}$$

$$\begin{aligned} \mathit{ofmap} : &K_{L2} \times P_{L2} \times \cancelto{\text{reused}}{C_{L2}}(P_{spatial} \times P_{L1} \times K_{spatial} \\ &\times K_{L1}) = P \times K \times \cancelto{\text{reused}}{C_{L2}} = P \times K \end{aligned} \tag{7}$$

Observe that each equation (i.e., the L2 access count for each tensor) is affected *only* by the spatially unrolled dimensions that *index* that tensor. For example, $P_{spatial}$ does not affect the *weight* tensor accesses because it is not indexed by P and can be broadcast to all the PEs across which P is unrolled.

Since we are considering a L2 loop ordering that lead to temporal reuse of *ofmap* across L1 tiles, C_{L2} does not affect the total number of L2 accesses (i.e., the sum of Equations 5 to 7). Therefore, similar to Section III-A, to reduce the total access count we must reduce some combination of P_{L2} and K_{L2}.

Since we also assumed the L1 tile configuration is decided, each candidate tile has P_{L1} and K_{L1} already determined, so those cannot change. Instead, we can unroll P and K spatially (i.e., maximize $P_{spatial}$, $K_{spatial}$, or some combination of those) to reduce P_{L2} and K_{L2}.

Although we cannot make any conclusion about the combination of the factors that should be unrolled, we still inferred what dimensions should not be unrolled (e.g., C in this example). Besides algebraic analysis, we know that unrolling in the C dimension would spatially reuse *ofmap* to reduce its access to L2. In this scenario, however, *ofmap* was already temporally reused across L1 tiles, and the number of L2 accesses was already optimized for this tensor. In the general case, we want to unroll dimensions that lead to spatially reusing the operand(s) which is(are) not temporally reused:

Spatial Unrolling Principle

Given a parallel processing level between memories X and $X-1$, assume the operand OP is temporally reused across the tiles due to the loop ordering at X. Moreover, suppose the tiling is optimized to reduce the number of memory accesses for OP to X as much as possible. Then, when unrolling dimensions, we reject as unrolling candidates the *non-indexing* dimensions of OP which would lead to its spatial reuse. This way, we can maximize the spatial reuse for other tensors rather than the already optimized OP.

Using this principle, we can prune more than 90% of the spatial unrolling candidates for ResNet-18 [24] convolution layers and a 14×12 PE array, similar to the one used in [10].

C. Multiple spatial and temporal levels

At this point, we can optimize the dataflow for any 2-level memory system, even when one of them is spatially distributed among several parallel units. Based on Section II-D, we can reduce the set of possible orderings. For each loop ordering choice from the reduced set, we can find a reduced set of tiling candidates as discussed in Section III-A.

Then, according to Section III-B, we can find the potential unrollings that lead to high data reuse for each loop ordering-tiling pair, model the total number of memory accesses for each, and choose the ordering, tiling, and unrolling with the minimum number of data accesses. We can also apply this approach level-by-level when there are more memory and parallel processing levels. Next, we will introduce two representations that facilitate the automatic application of our observations in this section to distinct tensor computations.

IV. Representation and Optimization

Sunstone accepts a description of the tensor workload and uses it to infer the reuse pattern of that workload. The following shows an example of this that describes an operation on two input operands and one output, where the first operand is 2D and indexed by C and some combination of indices P and R (e.g., $p+r$), the second is 3D and indexed by K, C, and R, and the output is 2D and indexed by K and P (this corresponds to the 1D convolution example of Section II-D).

```
dimensions = {K:4, C:4, P:7, R:3}
tensor_description = {
    operand1 = [C, (P, R)],
    operand2 = [K, C, R],
    output = [K, P]
}
```

TABLE III: Inferred reuse of each tensor in 1D convolution

tensor	indexed by	reused by	partially reused by
ofmap	k, p	c, r	
ifmap	c, p, r	k	r, p
weight	c, k, r	p	

Here, the indices are bound by $0 \leq K, C < 4$, $0 \leq P < 7$, and $0 \leq R < 3$. From this problem description, Sunstone identifies *indexing* dimensions of each tensor involved in the computation. Based on that and the **Ordering Principle 1** discussed in Section II-D, it also infers what tensor can be reused across which dimensions in a loop. For example, in 1D convolution, C, R, and P are indexing dimensions for *ifmap*, while K is a non-indexing dimension. It follows that the tensor can be *fully reused* across any non-indexing dimension.

Finally, in some computations (e.g., convolution), partial reuse exists due to the sliding window. This means some loops (here, R or P) can reuse a subset of data for some tensors (here, *ifmap*) across the tiles [10]. Sunstone also considers this when finding the set of optimal orderings. The information that Sunstone extracts for the 1D convolution example is shown in Table III.

A. Loop ordering representation

To automatically take advantage of the ordering principles from Section II-D and find a small set of possible orderings for a given tensor workload, Sunstone represents the loop ordering search space by a trie. An example of this is illustrated in Fig. 4 for the 1D convolution example.

Each node represents a partially-determined loop order and is annotated with the available reuse. At the root, the dimensions of all four nested loops are undetermined (denoted by x). The immediate children represent the possible choices for the innermost loop: e.g., xxxC means C is traversed in the innermost loop while the outer loops are undetermined. Their children, in turn, represent the traversal order of the innermost loop and the next-innermost loop: e.g., xxRK traverses K as the innermost loop and R in the next-innermost, and so on.

Each node is annotated with the operand(s) that can be reused: in ❶ *ofmap* (*of*) is reused when the innermost loop traverses the input channel (xxxC), in ❷ both *ofmap* and *ifmap* are reused when the innermost loop is the filter dimension R, and so on. Note that reuse can remain or disappear at higher levels as discussed in Section II-D (**Ordering Principle 3**): for example, in node ❸, the *ofmap* reuse across C is available because all innermost loops also reuse *ofmap* ❷, while the *weight* reuse across P in ❺ is not available because *weight* is not reused across R in ❷.

Once the trie has been constructed, some nodes can be pruned as strictly worse, relying on the two rules below. First, any nodes that offer no *further* reuse compared to their parent node can be pruned since none of their children will offer any further reuse either (**Ordering Principle 3**). For example, xxCK ❼ is pruned away because C reuses *ofmap*, but *ofmap* reuse has already been destroyed by the inner K loop.

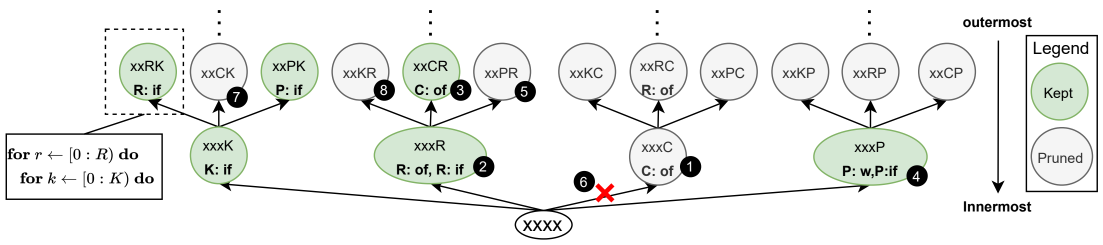


Fig. 4: Representing the order space of the 1D convolution problem as a trie, and how it can be pruned.

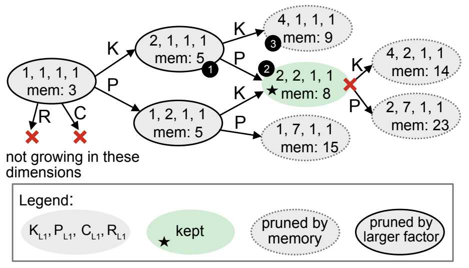


Fig. 5: L1 tile search space. The workload is the 1D convolution where $P = 14, K = 4, C = 4, R = 3$, and L1 size is 8 entries.

Second, if two child nodes either: (i) lead to reusing the same tensor α from the same dimensions A and B (possibly with different innermost orderings, such as xxAB and xxBA), or (ii) one node leads to reusing the same tensor α as the other but also reuses another tensor β, then one of the children can be pruned. Fig. 4 shows that since xxCR ❸ reuses the same operands as xxxC ❶ (i.e., *ofmap* via R and C), but also leads to additional partial reuse of *ifmap* via the R dimension, the xxxC node is pruned away ❻.

Once the set of promising orderings is established, tiling and unrolling candidates must be found based on the **Tiling and Unrolling Principles**, for each order in the set and at each temporal and spatial level, respectively. Next, we will show how Sunstone does this.

B. Tiling representation

To take advantage of the **Tiling and Unrolling Principles**, we again formulate the problem as a search tree. We will explain our representation with an example that assumes xxCR ordering — ❸ in Fig. 4 — as the loop ordering in L2 and aim to find good L1 tiling candidates.

We construct the tree using the smallest L1 tile possible (where every dimension is 1) as the root. Fig. 5 shows this for the running example where the total problem dimensions are $K = 4, P = 14, C = 4, R = 4$ and the unified L1 memory has 8 entries.

Each node is an L1 tile candidate, annotated with its L1 tile dimensions and the L1 memory footprint (we show a unified L1 here for clarity; there would be separate per-datatype footprints if L1 memories were separated by datatype). Each child node is a candidate identical to the parent node except for one dimension, which is enlarged to the next higher factor of the corresponding problem dimension. Since we were assuming xxCR ordering and showed in Section III-A that, based on the **Tiling Principle**, only indexing dimensions of *ofmap* (P and K) should be enlarged for this ordering, we only grow the tree in those two dimensions. For example, node ❶ represents the L1 tile with $K_{L1} = 2, P_{L1} = 1, C_{L1} = 1, R_{L1} = 1$, while its child ❷ is the same except that $P_{L1} = 2$.

Based on the **Tiling Principle**, nodes with at least one child still fitting in L1 can be pruned because the child offers strictly more reuse. For example, ❷ still fits in L1 and has more reuse than its parent ❶, so ❶ can be pruned. In contrast, node ❷ cannot be enlarged in any dimension without exceeding the L1 capacity. This is therefore a candidate for the optimal L1 tile, and remains unpruned. Lastly, node ❸ exceeds the L1 capacity and will not be considered any further.

Note that we can only use this method to draw conclusions between a node and its descendants (such as ❶ and ❷). Our pruning rules cannot draw further conclusions about nodes where *different* dimensions have been enlarged.

V. Evaluation

A. Methodology

System. We run Sunstone, all the prior tools, and all of our simulations on a system with an 8-core Intel Xeon CPU running at 2.1 GHz. Each core is equipped with 32-KB private L1 data and instruction caches and 1 MB of private L2 cache. The cores share 16.5 MB of L3 cache and 32 GB of memory.

Implementation. We implemented Sunstone in Python and added support for multithreading. For Sunstone and all the other tools that support multithreading, we set the number of threads to 8. We implemented Sunstone in a bottom-up fashion, where dataflow is optimized starting from the lowest level of memory and then level by level all the way up to the off-chip memory. At each level, the tiling, loop reordering, and unrolling (if applicable) are optimized as discussed in Section III-C. Finally, we also implemented a top-down approach which starts the optimization from the off-chip memory and compared it against the bottom-up approach in Section V-C.

Evaluation Platform. For fair comparison across tools, we evaluate each proposed mapping using the hardware-validated cost model of Timeloop [43]. Timeloop assumes double buffering can hide the latency of data transmission and estimates the performance of spatial accelerators as the sum of the operation/access count for each hardware component multiplied by its operation/access energy. We simulated the energies using Accelergy [64], which itself relies on Cacti [41] for SRAM and Aladdin [47] for other components.

TABLE IV: Evaluated accelerator configurations

Name	Simba-like	Conventional
Technology	45 nm	45 nm
Precision	*Weights*: 8-bit *Ofmap*: 24-bit *Ifmap*: 8-bit	*Weights*: 16-bit *Ofmap*: 16-bit *Ifmap*: 16-bit
Vector width	8	N/A
MACs per PE	8 × 8 lanes of vector MACs (8-bit)	Single MAC (16-bit)
PE grid	4 × 4	32 × 32
L1	*Weights*: 32 KB *Ofmap*: 3 KB *Ifmap*: 8 KB	Unified: 512 B
L2	*Ofmap and Ifmap*: 512 KB	Unified: 3.1 MB
BW (words/cycle)	inter-PE: (Read: 32, Write: 32) intra-PE: (Read: 64, Write: 8)	intra-PE: (Read:64, Write:64) intra-PE: (Read:3, Write:4)
NoC	Interleaved multi-cast inter-PE ofmap communication	

TABLE V: Hyperparameters for fast and slow configurations for Timeloop (TL) and dMazeRunner (dMaze). For TL, TO = timeout and VC = victory condition. For dMaze, util. = minimum utilization threshold.

Prior Work		Fast/Aggressive	Slow/Conservative
Timeloop [43] (TL)	TO	20000	80000
	VC	25	1500
dMazeRunner [14] (dMaze)	L1 util.	80%	60%
	L2 util.	50%	40%
	PE util.	80%	80%
	spatial reduction	not allowed	allowed

We also modeled the interconnect similar to that of Eyeriss [10]. Specifically, a destination tag with X and Y PE coordinates is added to every package to be delivered. Moreover, a tag check hardware at each PE ensures that only the designated PEs receive the data. We used Accelergy to model the energy for all these and included it in the total energy cost.

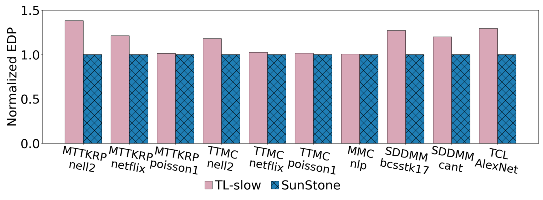
Merits. As is common practice, we use energy-delay product (EDP) as the key figure of merit for evaluating the performance of mappings. We use optimization wall-clock time to compare the dataflow optimizers' speed.

Accelerator Architectures. We evaluate the mappings for two representative architectures: a Simba-like [46] configuration and an Eyeriss-like conventional configuration similar to [30] and [10], detailed in Table IV. The Timeloop configuration files for these architectures were provided by the GitHub repository [65] of Accelergy [64], as well as that [27] of CoSA [28].

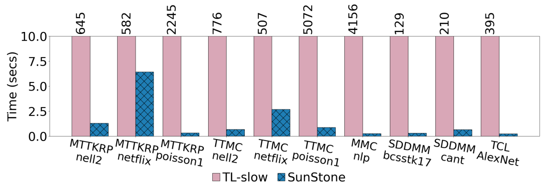
Benchmarks. We use a broad set of workloads listed in Table II.

Prior art. We compare against Timeloop [43] (TL), dMazeRunner [14] (dMaze), Interstellar [68] (INTER), and CoSA [28], the current state-of-the-art DNN mappers, using code from their respective repositories linked in their paper. We provide each tool with the same access/operation energy modeled with Accelergy for each hardware component.

For TL and dMaze, we use fast (*-fast*) and slow (*-slow*) configurations (see Table V); dMaze-fast is the default configuration from the repository. We use the default configurations for CoSA. For INTER, we preset the spatial unrolling to CK as prescribed in the paper [68], but allow unrolling of other dimensions whenever CK cannot fully utilize the PE grid. As TL can be extremely slow, we terminate it after one hour for each layer and take the best mapping found.

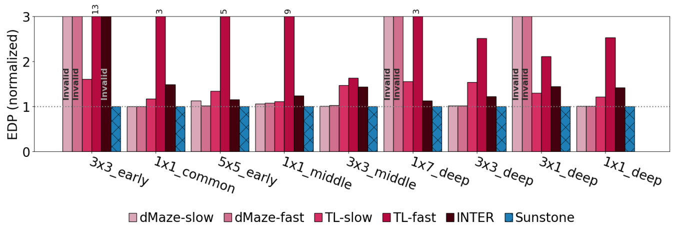


(a) normalized EDP

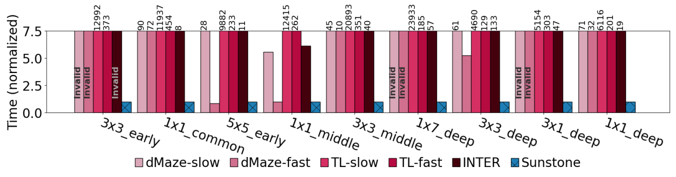


(b) Time-to-solution on non-DNN workloads

Fig. 6: Non-DNN workloads on the conventional accelerator



(a) EDP



(b) Time to solution

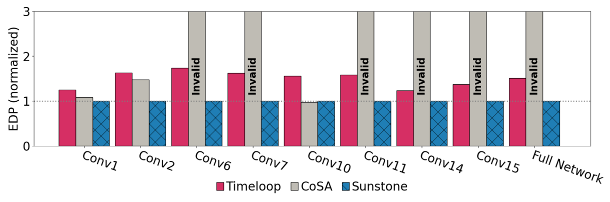
Fig. 7: Weight update (batch 16) of Inception v3 layers; invalid = no mapping meets the minimum utilization constraints, no mapping can use the preset unrolling, or the returned mapping does not correspond to the original computation.

B. *Comparison to prior works*

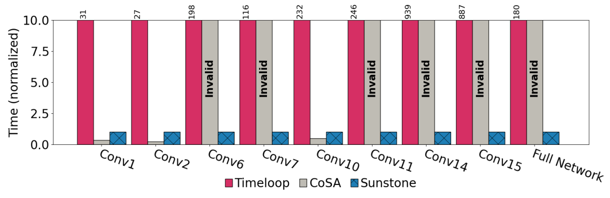
1) Non-DNN workloads: This section demonstrates Sunstone on non-DNN tensor workloads from table II. We evaluate *MTTKRP*, *TTMc*, and *SDDMM* with ranks 32, 8, and 512, respectively, on the conventional architectures of Table IV. We further assume that each PE has a datapath that can entirely consume every operand and produce one partial output in each cycle when operating at line rate.

Fig. 6a and 6b show that Sunstone outperforms TL in both solution EDP and time-to-solution. TL constructs a huge optimization space and uses no pruning methods to shrink it. As a result, it cannot search the whole space in a reasonable time. It may still have suboptimal solutions even after a long time and searching many configurations, as evident by Fig. 6a.

2) Convolution on a conventional accelerator: Fig. 7 shows the evaluation results for some of the convolution layers of Inception V3 network [55]. Overall, Sunstone is much faster



(a) EDP



(b) Time to solution

Fig. 8: Inference (batch 16) of ResNet-18 layers; invalid = some tiles of the returned dataflow do not fit in their memories.

than the prior art and produces mappings with better or equal EDP than them.

dMaze returns invalid mappings on several layers. That is because its minimum utilization conditions do not generalize well to different workloads (e.g., lighter, early convolution layers that do not utilize 50% of the L2 memory). Furthermore, unlike Sunstone, dMaze appears to assume that convolutions are always symmetric, so we were not able to use it to obtain valid mappings for asymmetric convolution layers *1 × 7 _deep* and *3 × 1 _deep* (Fig. 7a).

Timeloop, as discussed in Section V-B1, suffers from its huge optimization space and undirected random search approach. For example, some of its solutions utilize less than 20% of total L1 buffers capacity. Such solutions could have been excluded from the space via the **Tiling Principle** from Section III-A.

Finally, mappings from INTER have poor EDP on several layers. We found that often this is due to the highly restrictive strategy of only considering input and output channels for unrolling. While it helps to reduce the optimization space significantly, it sometimes excludes better mappings. For example, INTER's solutions sometimes reuse *ofmap* both temporally and spatially. This goes against our **Unrolling Principle** (Section III-B) that states for a memory level, if the number of accesses for one of the tensors is optimized with temporal reuse, the focus of spatial reuse should be on the other tensors.

3) Convolution on Simba-like accelerator: Fig. 8 shows the evaluation results for the Simba-like accelerator of Table IV. Besides Timeloop and CoSA, the other prior tools do not support optimizing the dataflow for such an architecture. Fig. 8b shows that CoSA can finish the scheduling faster than Sunstone. However, this comes at the cost that most of the mappings returned by it were invalid in our experiments.

Specifically, for the invalid mappings, one or more tiles did not fit in their designated memories of the accelerator. This is because CoSA approximates the non-linear dataflow optimization problem as linear to use linear optimizers such as [22]. Furthermore, for the layers that CoSA returns a valid mapping, solutions are mostly suboptimal compared to Sunstone, such as Conv2 with 1.48× worse EDP.

The Timeloop mapper, on the other hand, cannot even be invoked without providing some constraints on the optimization space. We used some of the constraints provided by the tool for this architecture on their GitHub page [42]. This helped Timeloop to find mappings, although still up to 900× slower than Sunstone as shown in Fig. 8b.

Furthermore, the provided constraints are manually designed and empirical-based. Therefore, they do not necessarily lead to optimal solutions; as shown in Fig. 8a, ResNet-18 achieves an overall 1.5× worse EDP when scheduled with Timeloop compared to Sunstone. We found that this is mainly due to the energy inefficiency of Timeloop mappings, as their latencies were equal to that of Sunstone.

TABLE VI: Effect of optimization order on optimization size and dataflow

Inter-level order	Intra-level order	Space size	EDP ($\times 10^{11}$)
bottom-up	unrolling → tiling → ordering	99350	4.8
bottom-up	tiling → unrolling → ordering	121050	4.8
bottom-up	ordering → tiling → unrolling	108298	4.8
top-down	unrolling → tiling → ordering	7391620	4.6

C. Order of optimization

Next, we study the effect of the order in which dataflow can be optimized on both EDP and optimization space. While we described our optimization process in Section III as finding the best tile candidates for each loop order and then finding the best unrolling for each order-tile pair, in general these can be done in any other order. That is, for example, one can find the best tilings, then the best unrolling candidates for each tile, and finally try all the loop orders on each tiling-unrolling candidate to find the best combination or vice versa.

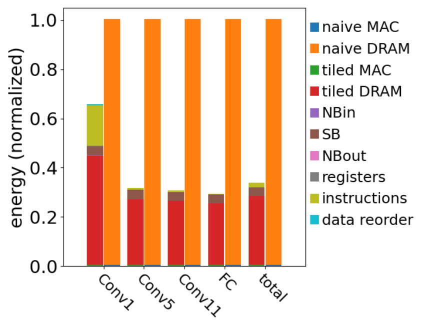
Moreover, since accelerators have multiple levels of memory and spatial tiling, the optimization can be done by starting at the lowest level (i.e., the one after the MACs) and going up level-by-level (bottom-up), or vice versa (top-down).

We examined optimizing ResNet-18 [24] convolution layers for an Eyeriss-like [10] accelerator in different orders (Table VI). Within a level, changing the order of unrolling, tiling, and loop order optimization does not significantly affect performance, which suggests that pruning techniques do not significantly depend on the order of optimization.

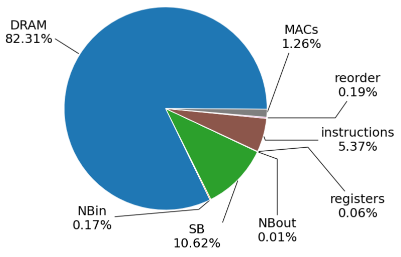
Across memory levels, the bottom-up approach examines an order of magnitude less dataflow than top-down approaches. This is because of two reasons. First, since on-chip memory of accelerators can hold large factors of problem dimensions, the optimization space is still large after the first optimization level of top-down approach. Second, alpha-beta-pruning can prune many more candidates when optimization is bottom-up, because the approximated energy after the first-level of optimization is much closer to the final energy as most of the accesses are to the lowest memory levels when reuse is high.

D. Overheads

Tiling and unrolling can introduce overheads. First, the number of instructions can increase as the code becomes



(a) Normalized energy of a naive (left) versus dataflow optimized (right) execution.



(b) Energy breakdown of executing ResNet-18 on a DianNao-like accelerator

Fig. 9: Tiling and unrolling overhead analysis

complex. Second, the data might need to be reordered in the memory according to the target dataflow so that the tiles of each operand can be read consecutively and in a burst. We define a processing pass as loading several data tiles into the on-chip memory, processing them, and storing the result. The reordering is needed because many accelerators encode the memory address and size of the operands needed for a processing pass. Having the tiles required for a processing pass in adjacent addresses can minimize the number of instructions needed to load/store them.

To analyze these overheads, we built an in-house simulator of an accelerator similar to DianNao [9], which describes its instruction set architecture (ISA). We also built a compiler that can generate DianNao-like instructions and compiled the layers of ResNet-18 with it.

We were able to process all the layers using 4.1 million of 256-bit DianNao-style instructions with our simulator. The number of instructions for each layer was much less than the number of operations for that layer, thanks to the SIMD nature of tensor workloads that DianNao ISA can capture. Specifically, instructions are needed every time a data transfer from/to off-chip memory is needed, but on-chip data can be processed without instructions using FSM controllers. We refer the readers to [9] for architectural details of DianNao and its ISA.

Our goal is to analyze the benefits of tiling and unrolling. We compare the energy needed for a dataflow-optimized execution of ResNet-18 with a naive case where data is streamed from DRAM. We used a similar method to Section V-A for modeling the access/operation energy of each component in the accelerator and used our simulator to get the event counts. We assumed the instructions are accessed from the expensive DRAM because the energy can only be improved if a dedicated memory is used for the instructions. The resulting energies are shown in Fig. 9a for some representative layers and the total execution.

The naive approach only spends energy on MACs and accessing DRAM. However, the inherent reuse in the workload is not captured. On the other hand, the tiled and unrolled execution leverages the reuse and swaps many of the DRAM accesses with cheaper accesses to on-chip buffers for weights (*SB*), ifmap (*NBin*) and ofmap (*NBout*). Consequently, the overall execution with tiling and unrolling is 2.9× more energy efficient, even though the overheads for instructions and data reordering exist. The overhead for instructions and reordering is only 5% and 0.2% of the overall execution, respectively. Finally, Fig. 9b shows the breakdown of energy when ResNet-18 layers are executed on a DianNao-like accelerator.

VI. Related Work

Transforming nested loops to achieve better performance is backed by decades of research. Seminal works such as [39, 62, 63] introduce the idea and explain how to apply it automatically. However, deciding among all possible transformations requires searching a vast space, and our work focuses on significantly reducing it.

Many prior works have focused on mapping tensor workloads to 2D accelerators. Timeloop [43] uses a random search with user-configurable termination criteria; as a result, it's not restricted to convolution, but is also far slower than other tools.

dMazeRunner [14], Marvel [7], and ZigZag [40] focus on DNNs and use directed search, introducing heuristics to prune the search space and relying on user-specified thresholds like utilization factors. Interstellar [68] also relies on empirically-driven heuristics. Sunstone also employs directed search, but eschews arbitrary knobs and user-specified hyperparameters in favor of rules based on algebraic analysis.

NN-baton [56] and Simba [46] target the high communication latency of multi-chip accelerators and propose novel dataflow at the package level to solve this. Their solutions are orthogonal to our on-chip optimization techniques and can be combined.

Others [4, 18, 36, 57] use polyhedral compilation techniques. These either target different platforms (e.g., GPU [18, 57] or CPU [37]), or are too generalized [4, 36] and suffer from inaccurate cost models which can lead to suboptimal solutions [5, 37].

Finally, another set of prior work tackles the mapping problem by attempting to approximate mapping in terms of well-known optimization problems, and using black-box optimizers. Gamma [32] and Ansor [70] use genetic algorithms. Ansor also introduces a method to generate a rich optimization space automatically. This can be combined with Sunstone to auto-generate a space that is both comprehensive and compact. CoSA [28] uses mixed-integer programming, Mindmappings [25] uses gradient descent, TVM [8] uses gradient tree boosting, and ConfuciuX [31] uses reinforcement learning. However, these approximations often don't capture parts of the problem and yield poor solutions (see Section V-B3).

VII. Conclusion

In this paper, we analyzed the reuse equations for executing tensor algebra on spatial accelerators and highlighted principles that can reduce the optimization space significantly. We designed and built a fast optimizer based on these, which supports dataflow optimization for various tensor workloads and modern accelerator architectures. Finally, we showed that our approach outperforms prior work by up to 800× in optimization time and up to 1.9× in EDP.

Acknowledgments

The authors are grateful to the anonymous reviewers for insightful feedback and helpful suggestions. This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Project Agency (DARPA) under agreement number FA8650-20-2-7007, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) under award number NETGP 485577-15. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL), Defense Advanced Research Project Agency (DARPA), the U.S. Government, the Natural Sciences and Engineering Research Council of Canada (NSERC), or the Government of Canada.

References

[1] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 662–673, 2018.

[2] Woody Austin, Grey Ballard, and Tamara G Kolda. Parallel tensor compression for large-scale scientific data. In *2016 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 912–922. IEEE, 2016.

[3] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 940–953, 2020.

[4] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149, 2015.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO 2019, page 193–205. IEEE Press, 2019.

[6] Xuan Bi, Annie Qu, and Xiaotong Shen. Multilayer tensor factorization with applications to recommender systems. *Annals of Statistics*, 46, 11 2017.

[7] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for dnn operators on spatial accelerators. *arXiv preprint arXiv:2002.07752*, 2020.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[9] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 269–284, New York, NY, USA, 2014. Association for Computing Machinery.

[10] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[11] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2016.

[12] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[13] Andrzej Cichocki, Danilo Mandic, Anh-Huy Phan, Cesar Caiafa, Guoxu Zhou, Qibin Zhao, and Lieven Lathauwer. Tensor decompositions for signal processing applications from two-way to multiway component analysis. *Signal Processing Magazine, IEEE*, 32, 03 2014.

[14] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–27, 2019.

[15] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[16] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems*, 27, 2014.

[17] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ISCA '15, page 92–104, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: Dsl for linear algebra and neural net computations on gpus. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 42–51, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Evgeny Frolov and Ivan V. Oseledets. Tensor methods and recommender systems. *CoRR*, abs/1603.06038, 2016.

[20] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '19, page 807–820, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Deepak Ghimire, Dayoung Kil, and Seong-heum Kim. A survey on efficient convolutional neural networks and hardware acceleration. *Electronics*, 11(6), 2022.

[22] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[23] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network, 2016.

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

[25] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 943–958, 2021.

[26] Yunxiang Hu, Yuhao Liu, and Zhuovuan Liu. A survey on convolutional neural network accelerators: Gpu, fpga and asic. In *2022 14th International Conference on Computer Research and Development (ICCRD)*, pages 100–107, 2022.

[27] Qijing Huang, Grace Dinh, and miheer vaidya. Ucb-bar/cosa: A scheduler for spatial dnn accelerators that generate high-performance schedules in one shot using mixed integer programming (mip). https://github.com/ucb-bar/cosa.

[28] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.

[29] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866, 2014.

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636. IEEE, 2020.

[32] Sheng-Chun Kao and Tushar Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Jean Kossaifi, Aran Khanna, Zachary Lipton, Tommaso Furlanello, and Anima Anandkumar. Tensor contraction layers for parsimonious deep nets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR)*, pages 26–32, 2017.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[35] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[36] Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22, 12 2012.

[37] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, apr 2021.

[38] Ye Liu and Michael K. Ng. Deep neural network compression by tucker decomposition with nonlinear response. *Knowledge-Based Systems*, 241:108171, 2022.

[39] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, jul 1996.

[40] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators. *IEEE Transactions on Computers*, 70(8):1160–1174, 2021.

[41] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[42] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation.

[43] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[44] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks, 2017.

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

[46] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108. IEEE, 2014.

[48] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, 2017.

[49] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, 2017.

[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[51] Age K Smilde, Paul Geladi, and Rasmus Bro. *Multi-way analysis: applications in the chemical sciences*. John Wiley & Sons, 2005.

[52] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[53] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702, 2020.

[54] Jian-Tao Sun, Hua-Jun Zeng, Huan Liu, Yuchang Lu, and Zheng Chen. Cubesvd: a novel approach to personalized web search. In *Proceedings of the 14th international conference on World Wide Web (WWW '05)*, pages 382–390, 2005.

[55] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 2818–2826, 2016.

[56] Zhanhong Tan, Hongyu Cai, Runpei Dong, and Kaisheng Ma. Nn-baton: Dnn workload orchestration and chiplet granularity exploration for multichip accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1013–1026, 2021.

[57] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

[58] M Alex O Vasilescu and Demetri Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *European conference on computer vision (ECCV)*, pages 447–460. Springer, 2002.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[60] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J. Dally, Joel Emer, Stephen W. Keckler, and Brucek Khailany. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[61] Sasindu Wijeratne, Rajgopal Kannan, and Viktor K. Prasanna. Reconfigurable low-latency memory system for sparse matricized tensor times khatri-rao product on FPGA. *CoRR*, abs/2109.08874, 2021.

[62] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, PLDI '91, page 30–44, New York, NY, USA, 1991. Association for Computing Machinery.

[63] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, page 655–664, New York, NY, USA, 1989. Association for Computing Machinery.

[64] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[65] Yannan (Nellie) Wu, Angshuman Parashar, Joel Emer, and Po-An Tsai. Accelergy-project/timeloop-accelergy-exercises: Exercises for exploring the fibertree, timeloop and accelergy tools. https://github.com/Accelergy-Project/timeloop-accelergy-exercises.

[66] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. Hasco: Towards agile hardware and software co-design for tensor computation, 2021.

[67] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. Procrustes: a dataflow and accelerator for sparse deep neural network training, 2020.

[68] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–383, 2020.

[69] Kaiqi Zhang, Xiyuan Zhang, and Zheng Zhang. Tucker tensor decomposition on fpga. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[70] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

Appendix

This appendix provides an abstract reuse analysis for tiling a tensor workload on a two-level memory hierarchy, extending the tiling principle discussion in Section III-A.

Assume a tensor algebra problem $P(OP_1, OP_2)$, with two tensor operands OP_1 and OP_2, and n dimensions represented with the set $D = \{d_1, \ldots, d_n\}$. Further, assume that $A \subset D$, represented as $A = \{d_1, \ldots, d_x\}$, is the set of non-indexing dimensions for OP_1. There will also be a $B \subset D$, the set of indexing dimensions for operand OP_1, represented as $B = \{d_{x+1}, \ldots, d_n\}$. We represent the sets of indexing and non-indexing dimensions for OP_2 as A' and B', respectively.

First, we consider the case that $A = B'$ and $B = A'$. In other words, the indexing dimensions for OP_1 are the non-indexing dimensions for OP_2 and vice-versa. Let $[d_y]$ represent the value of dimension d_y. Next, consider the following two-level tiling algorithm for P:

```
for $i_n \leftarrow [0, [\,dn\,]_{L2})$ do
    ...
    for $i_x \leftarrow [0, [\,dx\,]_{L2})$ do                 L2 tile
        ...
          for $i_1 \leftarrow [0, [\,d1\,]_{L2})$ do
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            for $j_n \leftarrow [0, [\,dn\,]_{L1})$ do           L1 tile
                ...
                for $j_1 \leftarrow [0, [\,d1\,]_{L1})$ do
                    computation
```

Here, $[d_y]_{L2}$ represents the L2-tile factor for the dimension d_y. From this, we can observe that OP_1 is being reused by the loops over $[d1]_{L2}$ to $[dx]_{L2}$ and derive the number of L2 accesses $\overline{OP}$ for each operand as follows:

$$\overline{OP}_1 = \prod_{i=x+1}^{n} [di]_{L2} \times \prod_{i=x+1}^{n} [di]_{L1} = \prod_{i=x+1}^{n} [di] \tag{8}$$

$$\overline{OP}_2 = \prod_{i=1}^{n} [di]_{L2} \times \prod_{i=1}^{x} [di]_{L1} = \prod_{i=x+1}^{n} [di]_{L2} \times \prod_{i=1}^{x} [di] \tag{9}$$

Observe that $\overline{OP}_1 + \overline{OP}_2$ consists of the product of full problem dimensions, which we cannot change, and $\prod_{i=x+1}^{n} [di]_{L2}$, the product of L2-tile factors for the indexing dimensions of the reused operand OP_1. Therefore, to minimize the total sum, we should decrease the L2-tile factors for the indexing dimensions of OP_1. Since $[di] = [di]_{L2} \times [di]_{L1}$, increasing these dimensions in the L1-tile will do the job.

Next, we consider the case where $B \cap B' \neq \emptyset$. In other words, OP_1 and OP_2 will have one or more common indexing dimensions. In this case, if $B' = \{d_1, \ldots, d_x\}$, not all the loops from B' can reuse OP_1 even if they are consecutive innermost loops. However, as long as $A - A' \neq \emptyset$, there will still be a x' such that $A - A' = \{d_1, \ldots, d_{x'}\} \subset D$ and the loops from this set can reuse OP_1 when they are consecutively innermost. Therefore, we make the same analysis as the previous case and aim to decrease the L2-tile factors for the dimensions in $\{d_{x'+1}, \ldots, d_n\}$.