AFRL-RY-WP-TR-2023-0021

A TENSORFLOW TO REAL-TIME MACHINE LEARNING (RTML) COMPILER

Miesko Lis
The University of British Columbia

JULY 2023
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

//Signature//

CHRISTOPHER A. BOZADA
Program Manager
Aerospace Components and Subsystems Division

//Signature//

GENE M. WILKINS, Lt Col, USAF
Deputy Chief
Aerospace Components and Subsystems Division
Sensors Directorate

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE July 2023	**2. REPORT TYPE** Final	**3. DATES COVERED** START DATE 6 January 2020	END DATE 28 February 2022

4. TITLE AND SUBTITLE
A TENSORFLOW TO REAL-TIME MACHINE LEARNING (RTML) COMPILER

5a. CONTRACT NUMBER FA8650-20-2-7007	**5b. GRANT NUMBER** N/A	**5c. PROGRAM ELEMENT NUMBER** 61101E
5d. PROJECT NUMBER 1000	**5e. TASK NUMBER** N/A	**5f. WORK UNIT NUMBER** Y21A

6. AUTHOR(S)
Miesko Lis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The University of British Columbia Vancouver, BC V6T 1Z4, Canada	**8. PERFORMING ORGANIZATIONREPORT NUMBER**

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command, United States Air Forces	Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RYDI	**11. SPONSOR/MONITOR'S REPORT NUMBER(S)** AFRL-RY-WP-TR-2023-0021

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-19-1-7996. 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 the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), or the U.S. Government. Report contains color.

14. ABSTRACT
This project developed techniques for compiling deep learning models to silicon hardware with a computation mapping and schedule. The key results are (i) a novel technique to effect mapping and scheduling of deep learning model computations on hardware, and (ii) a proof-of-concept energy-efficient hardware implementation capable of both training and inference tasks.

15. SUBJECT TERMS
machine learning integrated circuit, machine learning algorithms, real-time machine learning, neuroscience-inspired architectures, non-neural ML architectures, and generative adversarial learning techniques

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**C. THIS PAGE** Unclassified	SAR	36

19a. NAME OF RESPONSIBLE PERSON Christopher Bozada	**19b. PHONE NUMBER** *(Include area code)* N/A

Table of Contents

List of Figures

Figure **Page**

List of Tables

1 SUMMARY

Deep learning techniques have become ubiquitous and indispensable in many scientific and industrial applications. Compared to previous techniques, however, applying deep learning requires significant computational effort and processing memory, conditions especially challenging for low-power applications. This need has motivated silicon chips specifically designed to accelerate the matrix computations that are the core of deep learning operations.

For the most power-limited applications, however, more power efficiency is required. This can be achieved by tailoring the silicon chip hardware configuration and the how the computation is mapped to the hardware to the specific deep learning model being employed. Because there is a vast number of possibilities both in the hardware configuration and how a specific deep learning computations is mapped to it, automated tools are required to determine efficient solutions.

This document is a progress report on a project to develop techniques for compiling deep learning models to silicon hardware together with a computation mapping and schedule. The key results are (i) a novel technique to effect mapping and scheduling of deep learning model computations on hardware, and (ii) a proof-of-concept energy-efficient hardware implementation capable of both training and inference tasks.

The mapping technique divides the computation of a specific deep learning model into tiles that can be scheduled on a hardware accelerator. This is the key step in generating efficient hardware configurations specific to models, as efficient mapping alone can increase energy efficiency by one to two orders of magnitude. The technique describes outperforms prior art in efficiency and time-to-solution and does not rely on opaque parameters that require different settings for each deep learning model and even each layer of a single model.

The proof-of-concept hardware implementation supports both inference and training of deep learning models. It achieves efficiency through supporting weight sparsity, both during inference and during training. Measurements from a tapeout-ready GDSII of a 3mm^2 device in the Global Foundries 12nm process indicate that inference using ResNet50 takes 21ms and expends 8.9mJ per image, while training of ResNet50 takes 52ms and expends 15.8mJ per image. The proof-of-concept chip has been taped out and is currently being fabricated.

2 INTRODUCTION

In the past decade, deep neural networks have revolutionized many areas of science and industry. In tasks like image recognition, they surpassed the error rate of the human visual cortex seven years ago [1, 2], and has since only improved. Similar successes have been reported on tasks related to natural language processing, ranging from early sequence-based model successes [3, 4, 5] to giant models such as GPT-3 [6] based on the attention mechanism [7].

However, this advance has come at the cost of dramatic computation requirements: indeed, the deep learning "revolution" has only been possible because of the advent of graphics processing units (GPUs) that could be programmed for non-graphics tasks using languages like CUDA or OpenCL. Still, GPUs are primarily optimized for graphics processing, with many components that (e.g., texture units) and are less efficient than ideal for deep learning tasks. This has motivated a large number of hardware accelerators specific to deep learning inference [8, 9, 10, etc.], which are significantly more efficient than even GPUs.

In search for more efficiency, researchers discovered that deep learning models are overparametrized, and most parameters (model weights) can be set to zero without significant effect on accuracy. Since GPUs and early hardware accelerators could not take advantage of this, a new generation of "sparse" hardware accelerators was created, enabling another of efficiency [11, 12, 13, 14, 15, etc.]. While nearly all of these have focused on inference only, algorithmic and hardware innovations have also allowed sparse hardware accelerators for training deep neural networks [16].

However, these accelerators are still general-purpose to an extent: while they are optimized for deep learning workloads, they can execute any such workload, and are not customized to *specific* deep learning models. Potentially, customizing chips — and, more importantly, how workloads are executed on them — this can unlock another other of magnitude in performance and energy efficiency.

The purpose of the project reported on in this document is therefore to develop techniques that allow directly mapping specific deep neural network models to a hardware architecture together with a mapping and scheduling of exactly how deep learning workloads execute on the hardware.

2.1 DNN Accelerator Architectures

Figure 1 shows the architecture of a typical DNN accelerator [9, 17, 18, 19, 20, 21, 22, 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.

Accelerators also commonly include a larger memory shared among the PEs (L2-level), as well as a large off-chip DRAM. The PEs and L2 are typically interconnected via one simple on-chip interconnect per datatype [9].

Most accelerators are variants of this architecture pattern, because it can amortize the cost of retrieving data from memory by *reusing* it several times. The architecture here supports three kinds of reuse. First, operands may be reused *spatially*, that is, broadcast to a subset of (or all) PEs. L1 memories support short-term *temporal* reuse of operands *within* each PE, and L2 memories support temporal reuse *across* multiple PEs.

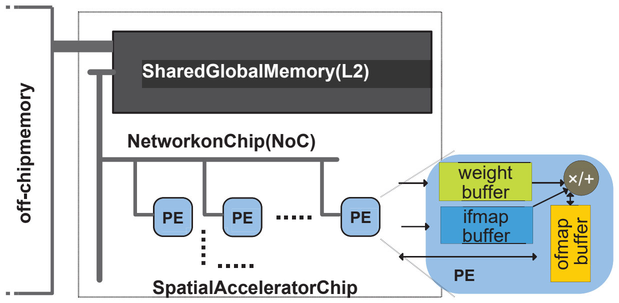


Figure 1: A Generic Spatial Accelerator Architecture with Per-PE (L1) Memories and a Shared On-chip L2 Memory

The desired computation (e.g., DNN inference) is tiled and mapped to these structures to maximize hardware utilization and minimize energy, through a process called *dataflow mapping* and described below.

The flow described in this report generates hardware for variants of this accelerator architecture and is able to take advantage of all three reuse opportunities.

2.2 Dataflow Mapping

While the design space for efficient hardware architectures is limited by the need to provide reuse opportunities, how computation — even of a single layer — is mapped to the accelerator is critical to performance and energy-efficiency. This step is known as dataflow mapping, as forms the heart of the compilation flow.

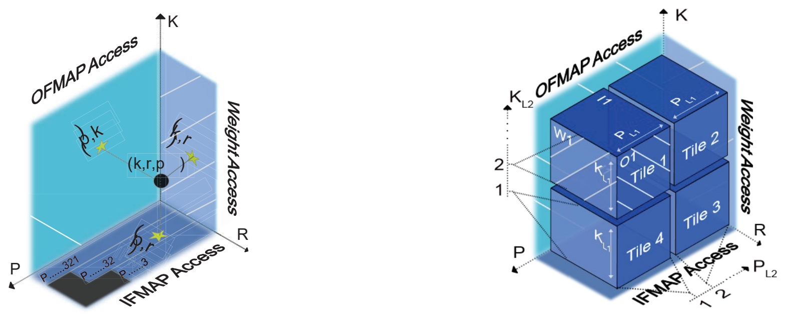
Dataflow mapping consists of tiling, loop ordering, and spatial unrolling. To explain each with a concrete example, we consider the convolution of a 1D tensor *ifmap* with K 1D filters with length R, defined by their *weights*:

$$ofmap[k, p] = \sum_{r} ifmap[p + r] \times weight[k, r] \qquad (1)$$

Typically, this is expressed as a nested loop [23]:

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

This defines a 3D *operation space* of $K \times P \times R$ MAC operations shown in Figure 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

Figure 2: Operation Space of a 1D Convolution. *Ifmap* is Shifted, Replicated, and Padded by 0 Along the Sliding Window Dimension *R*

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

Figure 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 following pseudocode, where the K dimension is divided into K_{L2} equal tiles of size K_{L1}, the P dimension into P_{L2} equal tiles of size P_{L1}:

```
1:  for $k_2 \leftarrow [0, K_{L2})$ do
2:      for $p_2 \leftarrow [0, P_{L2})$ do
3:          for $k_1 \leftarrow [0, K_{L1})$ do
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:                  $ofmap[k, p] \leftarrow 0$
8:                  for $r \leftarrow [0, R)$ do
9:                      $ofmap[k, p] \mathrel{+}= ifmap[p + r] \times weight[k, r]$
10:                 end for
11:             end for
12:         end for
```

```
13:        end for
14: end for
```

In addition to the intra-tile reuse described above, some tensor regions can be further temporally reused over multiple tiles. In Figure 2b, for example, region $W1$ can remain in L1 if tile 3 is processed in the same PE right after tile 4 (as it is in the pseudocode).

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 section 2.2 (order $P_{L2}K_{L2}K_{L1}P_{L1}R$), tile 2 will be processed right after tile 4, reusing the *I1* region of *ifmap* in L1.

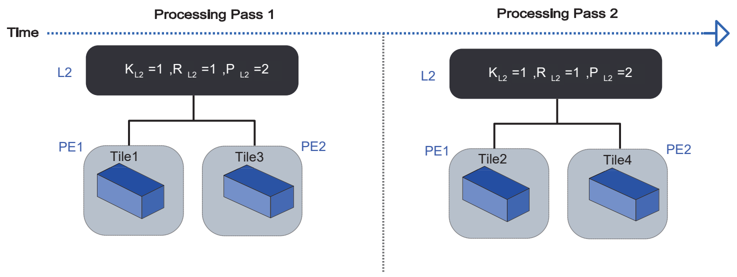


Figure 3: 1D Convolution Operation Space After Tiling

P dimension is divided into P_{L2} tiles of size P_{L1}. Similarly, K dimension is tiled into K_{L2} tiles of size K_{L1}.

Finally, loops can be *spatially* unrolled so that different tiles are assigned to different PEs. For example, in Figure 3 the K dimension is unrolled spatially across two PEs, so tiles 1 and 3 will be computed by PEs 1 and 2 in the first step, and then tiles 2 and 4 in the next. This allows inter-tile *spatial reuse*: 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 operations such as convolution. For example, the third layer of ResNet [24] yields 2.6×10^{10} possible configurations for execution on an accelerator architecture like the one in section 2.1, even with batch size 1 and only considering equally-sized tiles.

3 METHODS, PROCEDURES, AND DESIGN

The compilation flow developed as part of this project consists of key two steps: generation of a hardware description for an accelerator that will efficiently execute the input DNN model, and mapping/scheduling of the computations required in the input model to the generated hardware. This section describes the salient aspects of each step.

3.1 Hardware Architecture Template

The architecture template used to generate the hardware is capable of executing both inference and training of deep neural network models, with sparsity in one of the inputs (weights for inference and training forward pass, and activations during the training backpropagation passes).

3.1.1 Overall Chip Architecture

The overall architecture pattern is illustrated in fig. 4. It consists of four components: a decompression/compression module that converts compressed data (weights, input activations, etc.) retrieved from off-chip (e.g., from off-chip DRAM) to a decompressed form suitable for distributing among the processing elements; a multi-banked global buffer (GLB) shared among all processing elements that facilitates reuse of data fetched from off-chip storage; a simple network-on-chip (NOC) that connects the GLB banks with processing elements; and a 2D array of processing elements (PEs) that perform the required computations.

Depending on the input model and imposed hardware constraints (e.g., silicon area), the exact configuration of the elements in the generated hardware (e.g., number of GLB banks, number of PEs, etc.) varies.

3.1.2 Processing Elements Arrangement and Design

The processing elements are arranged in a single-instruction-multiple-data (SIMD) architecture that permits reuse of one input (typically the sparse input, such as sparse weights during inference) through broadcasting it to all processing elements in the SIMD group. This arrangement supports sparsity in one dimension, which allows efficient hardware utilization in both inference and training tasks.

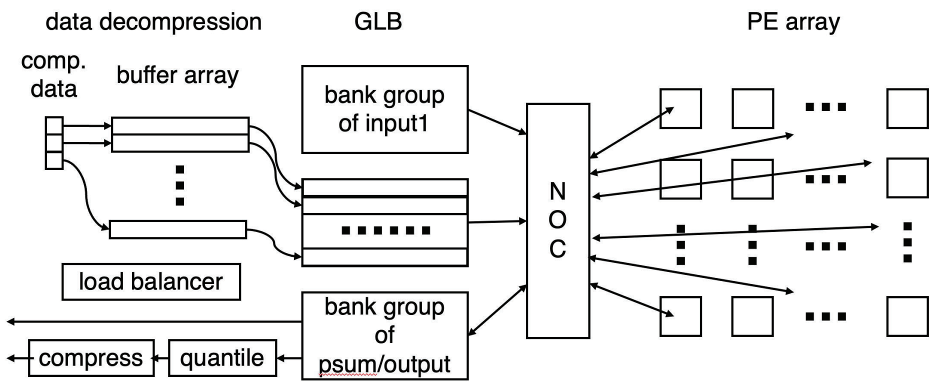


Figure 4: Overall Hardware Architecture Template

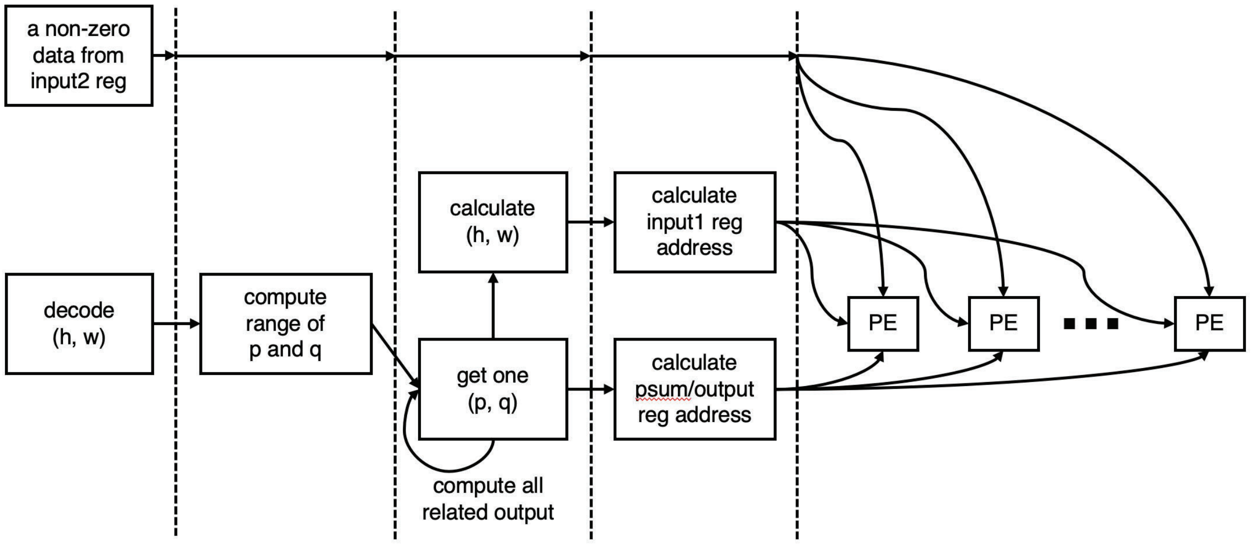

a non-zero data from input2 reg
decode (h, w)
compute range of p and q
calculate (h, w)
get one (p, q)
compute all related output
calculate input1 reg address
calculate psum/output reg address
PE
PE
PE

Figure 5: SIMD Structure, Illustrated on Inference / Training Forward Pass
Training backward and weight-update passes reuse the same hardware, with input1/input2 and coordinate computations suitably changed.

Figure 5 illustrates the SIMD arrangement, using a configuration during a convolutional layer inference task. During other tasks (e.g., training back-propagation), the hardware and computation remain the same, but the computed coordinates change: for example, for the weight-update pass, the output activation map coordinates p and q become the output weight filter coordinates r and s, and so forth.

Each SIMD unit (one row of the 2D array) operates on the same sparse input (during inference and training forward pass, this is the sparse weights), which is broadcast to all processing elements in the SIMD unit as shown in the figure. Each SIMD unit includes a shared SRAM that holds this data (input2 in the figure) in a format where non-zero elements are contiguous; this avoids any computation on ineffectual (zero-valued) weights and saves an order of magnitude in energy over a design where all weights are used for computation.

In contrast, the other two data elements required by the processing element (here, input1 and output) are computed separately in each PE.Figure 6 illustrates the pipelined architecture of each processing element. The processing element has two local SRAMs (in this configuration, one used for input1 and the other for psum), with the psum memory being updated in read-modify-write manner.

Again, the figure illustrates computation in for backward and weight-update passes during training, the same architecture is used, with only the coordinates being computed changing to match the problem at hand.

The hardware architecture described here enables both temporal reuse (through on-chip memories) and broadcast-type spatial reuse (via SIMD techniques). To take advantage of this, however, the computation of interest (such as deep neural network inference) must be carefully mapped and scheduled to execute on the hardware. This critical component of the compilation flow is described in the next section (section 3.2).

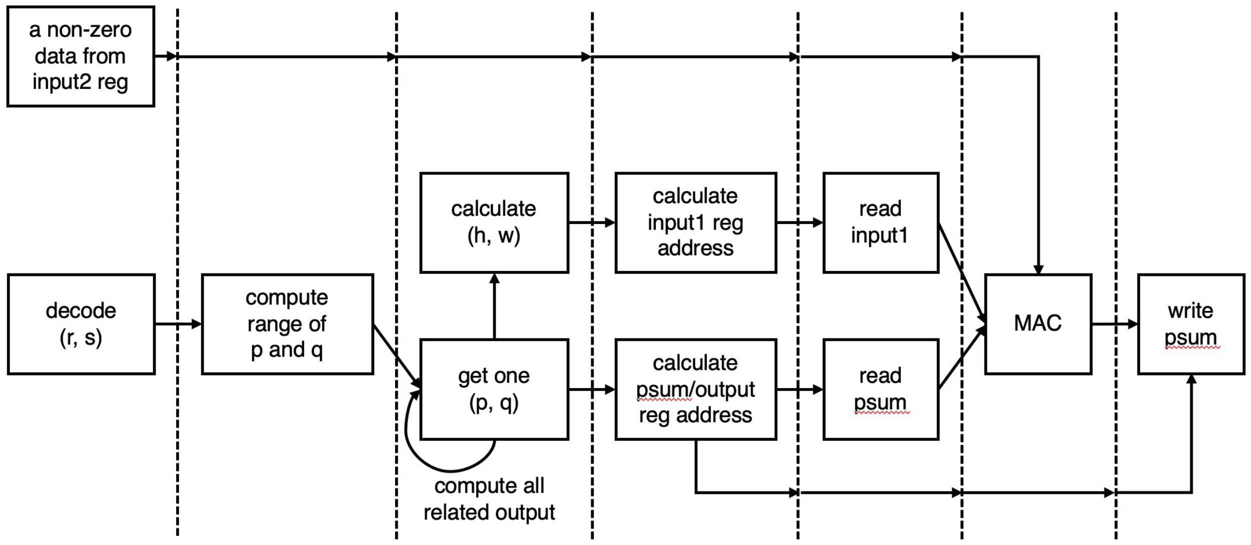

a non-zero data from input2 reg
decode (r, s)
compute range of p and q
calculate (h, w)
get one (p, q)
compute all related output
calculate input1 reg address
calculate psum/output reg address
read input1
read psum
MAC
write psum

Figure 6: Architecture Operation for Inference and Training Forward Pass
Training backward and weight-update passes reuse the same hardware, with input1/input2 and coordinate computations suitably changed.

3.2 Dataflow Scheduling and Mapping Algorithm

Figure 7 shows the overall flow of the mapping and scheduling algorithm developed in this project. The tool first accepts a description of the target architecture as well as the description of the tensor workload ❶ (section 3.2.1), and infers which tensors can be reused across which dimensions ❷ (section 3.2.2). Next, the algorithm determines a set of promising loop orderings ❸ (section 3.2.3), and a set of viable tiling candidates for L[1] ❹ (section 3.2.7). If per-PE memories end at L11, we

[1] The spatial unrolling step may occur later in the sequence if the PE has multiple private memory levels.

then find a set of spatial unrolling candidates (i.e., dimensions that will be spatially unrolled across the PE array) for each tiling candidate ❺ (section 3.2.8).

Next, we estimate the energy of each tiling candidate and their respective unrolling candidates, and apply a variant of alpha-beta pruning to remove suboptimal solutions ❻ (section 3.2.10).

Finally, we repeat steps ❹ and ❻ at each enclosing memory level (L2, L3, ...) ❼ (section 3.2.9) for each solution candidate, using the inter-tile traversal ordering of the prior level as the intra-tile order for the current level (e.g., intra-tile traversal at L2 equals the inter-tile order at L1).

In the rest of this section, we explain each of these steps in detail. As the running example, we will use a slightly more complex version of the 1D convolution example from section 2, where the *ifmap* vector has C channels and the *ofmap* vector has K channels:

$$ofmap[k, p] = \sum_{c} \sum_{r} ifmap[c, p + r] \times weight[c, k, r]. \quad (1) \qquad (2)$$

3.2.1 Target Workloads and Representation

The algorithm targets matrix algebra workloads that consist of nested loops with no inter-loop dependencies, i.e., loops that can be freely reordered; this includes deep neural network workloads, but also includes other tensor operations. 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* [25], *TTMC* [26] (which are bottleneck kernels in tensor decompositions), and various tensor contraction workloads that permeate the optimization domain [27, 28, 29, 30, 31, 32]. In contrast, many prior works such as [33, 34, 35] hardcode assumptions about how operands interact, and so only support optimization of a small subset of these workloads (mostly conv and fully connected layers).

Table 1: Inferred Reuse of Each Tensor in eq. (1)

tensor	indexed by	full reuse	partial reuse
ofmap	*k*, *p*	*cr*	–
ifmap	*c*, *p*, *r*	*k*	*p*, *r*
weight	*c*, *k*, *r*	*p*	–

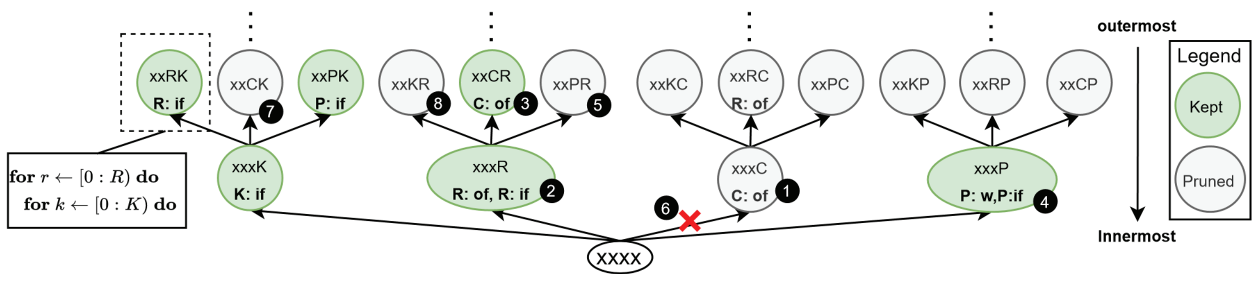
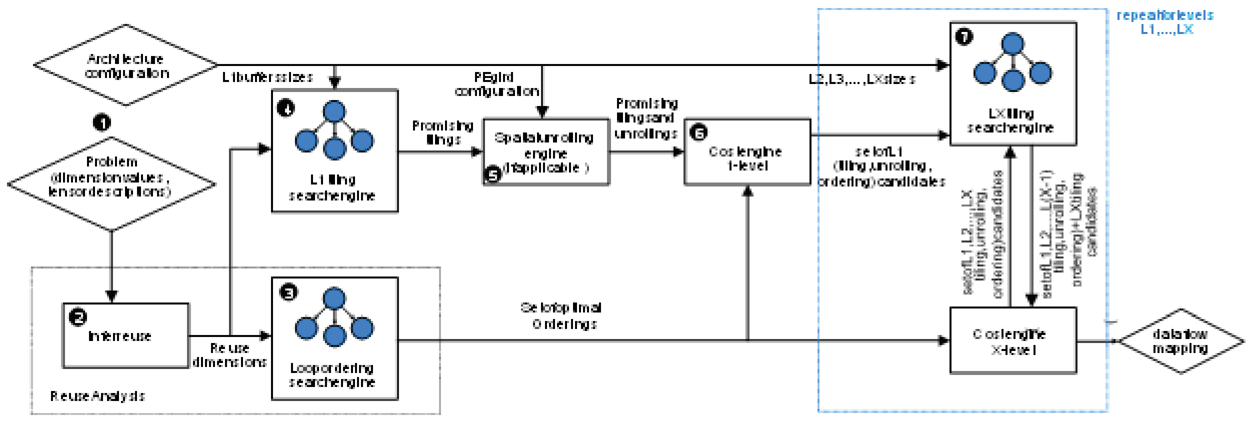


Figure 7: The stages in the Scheduling and Mapping Algorithm



Architecture configuration
L1 buffer sizes
PE grid configuration
L2,L3,...,LX sizes
repeat for levels L1,...,LX
1
Problem (dimension values, tensor descriptions)
4 L1 tiling search engine
Promising tilings
5 Spatial unrolling engine (if applicable)
Promising tilings and unrollings
6 Cost engine 1-level
set of L1 (tiling, unrolling, ordering) candidates
7 LX tiling search engine
set of L1, L2, ..., LX tiling, unrolling, ordering) candidates
set of L1, L2, ..., L(X-1) tiling, unrolling, ordering)+LX tiling candidates
2 Infer reuse
Reuse dimensions
3 Loop ordering search engine
Set of optimal Orderings
Cost engine X-level
dataflow mapping
Reuse Analysis

Figure 8: Representing the Order Space of the 1D Convolution Problem as a trie, and how it can be Pruned

3.2.2 Inferring Reuse

From this problem description, the algorithm infers reuse as follows. First, observe that the location accessed in each tensor can change only when the problem dimensions that index this tensor (the tensor's *indexing* dimensions) change. In contrast, when any of the tensor's *non-indexing* dimensions change, the location in the tensor stays the same. For example, in eq. (1), 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.

A second kind of reuse arises when a tensor index is an arithmetic combination of problem dimensions, such as the $p+r$ index of *ifmap* in eq. (1). This is because $p+r$ can have the same value for multiple values of p and r. Typically this involves less reuse than is due to non-indexing dimensions — e.g., in eq. (1), *ifmap[0]* from tile 1 is not reused but *ifmap[1]* and *ifmap[2]* are — so we refer to this as *partial reuse*.

In general, inferring this reuse requires algebraic analysis on the indexing expression [36**?**]. The algorithm described here, however, only considers the common case where the partial reuse is due to arithmetic combinations of multiple problem dimensions and which includes all deep learning workloads. Table 1 shows the reuse dimensions inferred from eq. (1).

3.2.3 Loop order (inter-tile reuse)

Recall that the order in which loops are nested determines the inter-tile reuse. In this section, we show how the algorithm examines the potential orders and rejects those that exhibit strictly worse reuse.

3.2.4 Insights

The method presented here relies on the following observations, which we explain by discussing the traversal between L1 tiles (i.e., ordering at the L2 level) in the running example (eq. (1)). Below, we have isolated the L2 level of an example dataflow for this computation:

```
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
5:                 $L1$ tile computation
6:             end for
7:         end for
8:     end for
9: end for
```

Observe that while 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 (section 3.2.4) is inside the K loop (section 3.2.4) — that is, within each iteration of the K loop, there are multiple iterations of the C loop that replace the *ifmap* tensor in L1 and prevent reuse between K iterations. In the general case, this observation becomes:

Observation 1

For a non-indexing dimension of an operand to actually reuse the operand, it must either be the inner-most loop, or the loops inside it must be limited to the other non-indexing dimensions of that same operand.

Next, note that while the innermost loops R (section 3.2.4) and C (section 3.2.4) lead to the reuse of the *ofmap* tensor, reordering the loops *above* them (i.e., section 3.2.4 and section 3.2.4) does not impact the number of accesses to *weight* and *ifmap* even though these tensors could potentially be reused across P and K respectively. This is because C and R index *weight* and *ifmap*, and they load different tiles of these tensors within each iteration of P and K, destroying any potential reuse. The general case is:

Observation 2

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.

Lastly, observe that C (section 3.2.4) and R (section 3.2.4) are full-reuse dimensions of *ofmap*, while R is also a partial-reuse dimension for *ifmap*. If we make R the innermost loop, we fully reuse *ofmap* across R and C and we partially reuse *ifmap* across R. On the other hand, if C is inner-most (i.e., section 3.2.4 and section 3.2.4 are swapped), the partial reuse of *ifmap* will be destroyed because C is an indexing dimension of *ifmap*. In general:

Observation 3

Even when considering only the inner loops that reuse the same tensor, certain ordering of these inner loops may lead to less reuse than others.

3.2.5 Representation

To take advantage of these observations, the mapping algorithm represents the loop ordering search space by a trie, illustrated in fig. 8 on the running example from eq. (1).

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 (represented 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 due to Observation 1: 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.

3.2.6 Pruning

Once the trie has been constructed, some nodes can be pruned as strictly worse, relying on the two rules below. (Our implementation in fact never generates these nodes, but we discuss the concepts in terms of pruning for clarity.)

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 (Observation 2). For example, xxCK ❼ is pruned away because C reuses *ofmap*, but *ofmap* reuse has already been destroyed by the inner K loop. Based on the same argument, xxKR ❽ and xxPR ❺ will also be pruned.

Second, if two child nodes either: (i) lead to reusing the same tensor α from the same dimensions A and B (although 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 additionally reuses another tensor β, then one of the children can be pruned (Observation 3). Figure 8 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.

The mapping algorithm is conservative and only prunes orderings that provably lead to suboptimal reuse or reuse that is already being offered by another ordering. For example, in fig. 8, both xxRK and xxPK are kept: even though both reuse *ifmap*, the partial reuse is offered by different dimensions, so one could lead to more reuse than the other depending on the P and R bounds.

Removing provably-worse orderings significantly prunes the ordering space. For example, for batched convolution, the set of potential orderings is pruned from 7! =5040 to 10 orderings.

3.2.7 L1 Tile Size Optimization

Next, algebraic analysis selects efficient L1 memory tile configurations, which in turn determines both the intra-L1 reuse and what can be reused at higher levels. We again explain this in terms of pruning, although the actual algorithm never actually generates the suboptimal configurations.
Insights To determine suboptimal L1 tilings, let us consider the following dataflow for eq. (1):

```
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
```

```
4:          for $p_1 \leftarrow [0, P_{L1})$ do
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
9:                  end for
10:             end for
11:         end for
12:       end for
13:     end for
14:   end for
15: end for
```

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*, and the total number of L1 tile iterations is $P_{L2}\times K_{L2}\times C_{L2}$. Thus, to execute the full workload, the total number of L2 memory accesses would be #*passes* × *tile_size*, broken down as:

$$ifmap\ K_{L2}\times P_{L2}\times C_{L2}(P_{L1}+R-1)\times C_{L1} \tag{2}$$

$$= K_{L2}\times C\times P_{L2}(P_{L1}+R-1)\ weight$$

$$:K_{L2}\times P_{L2}\times C_{L2}(C_{L1}\times K_{L1}\times R)$$

$$= \underbrace{C\times K\times R}_{\text{problem dimensions}} \times P_{L2} \tag{3}$$

(4)

$$ofmap\ :K_{L2}\times P_{L2}\times C_{L}{}^{*}2(P\text{reused}_{L1}\times K_{L1})$$

reused

:

$$= P\times K\times C_{L}{}^{*}2 \qquad = P\times K$$

Here, *ofmap* is reused C_{L2} times — that is, *ofmap* remains in L1 between L1 tiles — because that is the innermost L2 loop. The total L2 access count is the sum of eqs. (2) to (4):

$$\text{L2 accesses} = K_{L2}\times C\times P_{L2}(P_{L1}+R-1)$$

$$+C \times K \times R \times P_{L2} + P \times K \tag{5}$$

For best L1 reuse, our task is to minimize this, under the constraint that the L1 tiles of all datatypes fit in the L1 memories — e.g., $R_{L1} \times C_{L1} \times K_{L1}$ must not exceed the L1 weight buffer.

What are the degrees of freedom here? Equations (2) to (4) involve either full problem dimensions (e.g., C, K and R in eq. (3)) — which we cannot change — or loop bounds (e.g., P_{L2} in eq. (3)) — which we can select to change L1 tile dimensions. For example, the *ofmap* access count $P \times K$ only includes full problem dimensions (eq. (4)), 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 for the sake of argument 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), because there are fewer *ifmap* and *weight* fetches. We can generalize this observation and use it to prune suboptimal L1 tile sizes:

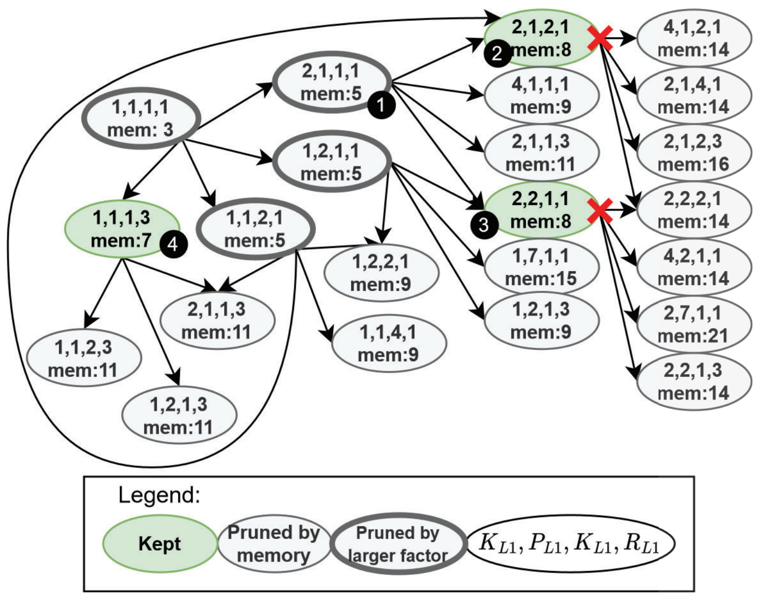


Figure 9: L1 Tile Size Search and Pruning

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

Observation 4

For any given tile T, if any of its L1 dimension can be enlarged while still fitting in the L1, the larger tile will have fewer data accesses; therefore T can be pruned.

Representation To take advantage of observation 4, we again formulate the problem as a search tree, this time rooted at the smallest L1 tile possible (where every dimension is 1). Figure 9 shows this for the running example where the total problem dimensions are P =14, K =4, R =4, C =4 and the unified L1 memory has 8 entries.

Each node is an L1 tile size 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 of its children is a candidate that is identical to the parent node except for one dimension, which is enlarged to the next higher factor of the corresponding problem dimension. For example, node ❶, which represents the L1 tile with C_{L1} =2, P_{L1} =1, K_{L1} =1, R_{L1} =1, while its child ❷ is the same except P_{L1} =2.

Based on observation 4, nodes that have at least one child that still fits 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.

Note that we can only use this method to draw conclusions between a node and its descendants (such as ❶ and ❷). Our pruning rule cannot draw further conclusions about nodes where *different* dimensions have been enlarged: for example, ❷ and ❸ cannot be directly compared without knowing the next-level (e.g., L2) tiling, so both nodes are kept.

Applying this technique, the L1 tile search space for ResNet-18 [24] convolution layers is reduced by up to 80% (vs all valid L1 tile candidates).

Finally, for each remaining L1 tile, we compute the number of memory accesses under each remaining loop ordering (section 3.2.3), and pair the L1 tile with the ordering that leads to the fewest memory accesses.

3.2.8 Spatial unrolling

At this point, we have a set of candidate L1 tiles, each with its optimal L2 loop order. For each of these options, we find the best ways to spatially distribute the work among multiple PEs.

In the running example, this step takes place between the L1 and L2 levels because the L1 memories are private and the L2 is shared among all PEs. If both L1 and L2 were private, we would spatially unroll after L2; indeed, this can occur multiple times if the PEs are clustered with per-cluster storage [37].

Insight Spatially unrolling some loops can reduce the number of L2 accesses *if* the same data is needed by multiple PEs at the same time by broadcasting said data (see section 2).

We continue the running example, now adding a spatial tile in each dimension, so that now $P = P_{L2} \times P_{spatial} \times P_{L1}$, etc.:

```
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** $k_{spatial} \leftarrow [0, K_{spatial})$ **do**
5:                  **for** $p_{spatial} \leftarrow [0, P_{spatial})$ **do**
6:                      **for** $c_{spatial} \leftarrow [0, C_{spatial})$ **do**
7:                          **for** $k_1 \leftarrow [0, K_{L1})$ **do**
8:                              **for** $p_1 \leftarrow [0, P_{L1})$ **do**
9:                                  **for** $c_1 \leftarrow [0, C_{L1})$ **do**
10:                                     **for** $r \leftarrow [0, R)$ **do**
11:                                         compute
12:                                     **end for**
13:                                 **end for**
14:                             **end for**
15:                         **end for**
16:                     **end for**
17:                 **end for**
18:             **end for**
19:         **end for**
20:     **end for**
21: **end for**
```

To account for the spatial unrolling, we expand the equations from section 3.2.7:

$$ifmap : K_{L2} \times P_{L2} \times C_{L2}(P_{spatial} \times P_{L1} + R - 1) \times C_{spatial} \times C_{L1} = K_{L2} \times C \times P_{L2}(P_{L1} + R - 1) \tag{6}$$

$$weight\ K_{L2} \times P_{L2} \times C_{L2}(C_{spatial} \times C_{L1} \times K_{spatial} \times K_{L1} \times R) = C \times K \times R \times P_{L2} \tag{7}$$

$$ofmap : K_{L2} \times P_{L2} \times C^{*}_{L2}(P\text{reused}_{spatial} \times P_{L1} \times K_{spatial} \quad \text{reused} \quad : \quad \times K_{L1}) = P \times K \times C^{*}_{L2} = P \times K \tag{8}$$

Observe that 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 *weights* acceses because *weights* are not indexed by P and can be broadcast to all PEs across which P is unrolled.

Once again, since $ofmap$ is temporally reused across C at the L1 local buffers, C_{L2} does not affect the total number of L2 accesses (i.e., the sum of eq. (6), eq. (7), and eq. (8)). Therefore, to reduce the total access count, we must reduce some combination of P_{L2} and K_{L2}. This time, however, each candidate tile has P_{L1} and K_{L1} already determined in section 3.2.7, so those cannot change. Instead, we can unroll P and K spatially, i.e., maximize $P_{spatial}$, $K_{spatial}$, or some combination of those. We do not make any conclusion about the combination of the factors that should be unrolled (i.e., it tries all the possible combination for those factors), but rather infers what dimensions should not be unrolled (e.g., C_{L2} in this example). In general:

Observation 5

To maximize the spatial reuse when unrolling dimensions, the *reuse* dimensions of the operand(s) that are *temporally reused* at the PE local buffers should not be spatially unrolled, unless the PE array cannot be utilized otherwise.

Finally, for each L1 tile/ordering pair, we consider each remaining unrolling dimension and enumerate all of its factors, retaining all resulting combinations. With our unrolling technique, we can prune more than 90% of unrolling candidates for ResNet-18 [24] convolution layers and a 14×12 PE array, similar to the one used in [9].

3.2.9 Optimizing the L2 Level and Beyond

At this point, we have a set of candidate L1 tiles, each with its optimal L2 loop ordering and potential spatial unrollings.

For each candidate, we generate the potential L2 tile sizes in the same way as for L1 (section 3.2.7), exploring and pruning the tile space using the same search tree representation; again, each L2 tile is paired with its optimal L3 loop ordering. (The spatial unrolling step is only repeated if the next memory level (L3) is shared among multiple L2s.)

This process repeats until tile candidates have been computed for all on-chip memory levels. Note that this will likely result in multiple L2 tile options for each L1 tile candidate, and so on; we show how to avoid evaluating all of them below.

3.2.10 Dynamic Inter-Level Pruning

Rather than examine all possible combinations of tile sizes (L1, L2 tiles, etc.), we evaluate them in a specific order and employs a variant of alpha-beta pruning [38] to dynamically reject suboptimal tiles.

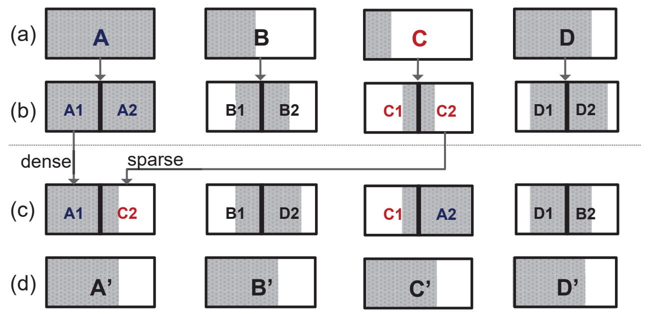


Figure 10: Load Balancing: Work Tiles are Cut in Half (b) and the Halves Rearranged in Dense-Sparse Pairs (c)

To do this, we first assign a cost to each L1 tile, equal to energy incurred from the transactions between the L1 and L2 (the number of transactions obtained from using the equations in section 3.2.7); we refer to this cost as the *L1-cost*. Similarly, we refer to the energy from the transactions between the L2 and DRAM (or L3) as the *L2-cost*, determined by the L2 equivalents of the same equations, and so on.

Next, we also compute an ideal bound for the L2-cost, by assuming each operand is only ever fetched once (cf. section 3.2.7); we call this *L2-ideal*. This number is unrealistic because specific loop orders reuse only subsets of operands; however, we can use it to prune the search space as follows.

We start the search by examining the L1 tile with the lowest L1-cost. We generate and examine examine all of the L2 tile candidates for this L1 tile, select the L2 tile with the minimum L2-cost, and obtain the current-best L1+L2 cost candidate by adding the L1- and L2-costs.

Next, we examine the L1 tile with the next-lowest L1-cost. We compute the ideal L1+L2-cost by adding the actual L1-cost to the L2-ideal cost lower bound. If this cost exceeds the current-best L1+L2 cost, we can reject the current L1 tile, because no possible L2 ordering can have a lower L2 cost than the ideal. Moreover, we can also reject the remaining L1 tiles, because all of them have a higher L1 cost and therefore a higher L1+L2 lower bound.

If, on the other hand, the ideal cost is less than current-best, we evaluate all L2 tile options for this L1 tile, update the current-best L1+L2 cost as before, and continue the process.

Applying this technique on ResNet-18 [24], we reduce the number of searched L1 tiles by 50-80%, while reducing the number of evaluated L2 tiles by 60–90%.

3.2.11 Load Balancing and Dataflow

When model weights are sparse (e.g., inference on pruned models or during a pruning-whiletraining process), load imbalance can arise because tiles assigned to different processing elements can have different sparsity. When this occurs, all processing elements that have finished processing their tiles must wait for the slowest processing element to finish before accepting new tiles. With typical weight sparsity rates (~90%), this load imbalance causes an average execution latency overhead of 50%, in some times exceeding 100% (i.e., more than double the time required to compute the same number of operations if they were perfectly balanced).

To combat this, our flow relies on a load balancing scheme to bring the amounts of work allocated to the processing elements within a small margin of one another. Figure 10 illustrates this load balancing process. First, every work tile (a) is cut into two halves along one of the tile dimensions (b); because sparsity is almost certainly uneven within the tile, the two halves will likely have different densities. Next, the halves are sorted according to density, and half-tiles are matched starting from opposite ends (c): the sparsest half-tile is matched with the densest half-tile, and so on. This ensures that each newly formed tile is as close as possible to the average density across all PE work tiles (d).

4 RESULTS AND DISCUSSION

This section describes a proof-of-concept chip tapeout to verify the compilation flow described in section 3, as well as measurements on the post-layout GDSII form of the chip pre-tapeout. Note that the methods and design procedures developed during this project are described in the previous section (section 3).

4.1 Proof-of-Concept Chip Details

A proof-of-concept chip RTL was generated for tapeout in Global Foundries 12nm technology (GF12). The device architecture is organized as described in section 3.1, including support for sparse inference and training (compression/decompression) and SIMD organization.

The chip has a total silicon area of 3mm^2, and is configured with192 processing elements (PEs) in a 16×12 grid. Because the device supports training as well as inference, all data-related computations (e.g., multiply-and-add units) are performed in floating-point arithmetic in the 16-bit bfloat16 format (1 sign bit, 8-bit exponent, 7-bit mantissa).

However, as we were unable to obtain DRAM IP blocks either through DARPA channels or from the ASIC library provider (Synopsys), the device does not include a DRAM interface. Instead, the chip includes a custom off-chip interface that connects to an FPGA; the FPGA acts as a virtual DRAM device to feed data to the chip as a DRAM interface would.

The device was taped out in early 2022; samples are expected in September 2022.

To efficiently map DNN workloads to the generated hardware, techniques developed in section 3.2 were used.

4.2 Measurements: Inference

Measurements on inference tasks were performed for three benchmarks for the MLPerf suite: ResNet 50 (classification), GNMT (language model), and Mask RCNN (image segmentation). Measurements are reported for both batch processing and single inputs (e.g., single images). All measurements used the pre-layout chip GDSII.

The measurements are reported in table 2. Note that the models perform different tasks and have very different input sizes; thus, the latency and energy numbers cannot be compared between models.

4.3 Measurements: Training

Measurements on training tasks were performed for three benchmarks for the MLPerf suite: ResNet 50 (classification), GNMT (language model), and Mask RCNN (image segmentation).

Because the computation for the different phases has different properties — for example, in CNNs, the forward and backward passes involve convolving a large tensor with a small convolutional kernel to obtain a large tensor, whereas the weight update pass convolves two large tensors to obtain a small one — the mappings obtained from the (see section 3.2) have different batch sizes for the passes.

As with inference measurements, we used the pre-layout chip GDSII.

The measurements are reported in table 3. Note that the models perform different tasks and have very different input sizes; thus, the latency and energy numbers cannot be compared between models.

model	single-inputlatency(ms)	single-inputenergy(mJ)	batchsize	batchlatency(ms)	batchenergy(mJ)
ResNet50	21.1	8.9	12	253	106.5
GNMT	0.3	0.02	12	3.36	0.24
MaskRCNN	1,471.7	672.2	12	17,660	8,066.9

Table2:Latencyandenergy-efficiencyofthreeMLPerfbenchmarksoninferencetasks,bothinsingle-inputmodeandinbatchmode batchsize12).Notethat,inadditiontothemodelsbeingdifferent,themodelsperformdifferenttasksandinputsizesforthemodels arevastlydifferent.

modelpassbatchsizebatchlatency(ms)batchenergy(mJ)
ResNet50forward601130513.9
ResNet50backward601010185.5
ResNet50weightupdate641050267.9
GNMTforward6019.54.21
GNMTbackward6013.64.59
GNMTweightupdate6434.32.18
MaskRCNNforward6087,37639,275
MaskRCNNbackward6084,42017,695
MaskRCNNweightupdate6461,0109278

Table3:Latencyandenergy-efficiencyofthreeMLPerfbenchmarksontrainingtasks.Batchsizeissetto60forforwardandbackward passes,and64forweightupdatepass.Notethat,inadditiontothemodelsbeingdifferent,themodelsperformdifferenttasksand inputsizesforthemodelsarevastlydifferent.

5 CONCLUSIONS

This document reports on the key findings from grant FA8650-20-2-7007 (RTML), an effort to develop techniques to compile DNN models directly to hardware. The flow consists of two components: (i) generating hardware architecture from a template, described in section 3.1, and (ii) mapping of the DNN computation to the generated hardware, described in section 3.2.

As part of the project, a proof-of-concept hardware implementation was generated and taped out in the GF12 process. The chip supports both inference and training of deep learning models, achieving efficiency through supporting sparse computation in both cases.

References

[1] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, “DeepFace: Closing the Gap to Human-Level
Performance in Face Verification,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[2] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level
Performance on ImageNet Classification,” in *IEEE International Conference on Computer Vision (ICCV)*, 2015.

[3] A. Graves, A. R. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[4] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” in *NIPS*, 2014.

[5] W. Xiong, L. Wu, F. Alleva, J. Droppo, X. Huang, and A. Stolcke, “The Microsoft 2017 Conversational Speech Recognition System,” *arXiv:1708.06073*, 2017.

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *ASPLOS*, 2014.

[9] Y.-H. Chen, J. Emer, and V. 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)*, 2016, pp. 367–379.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” in *ISCA*, 2017.

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: efficient inference engine on compressed deep neural network,” in *ISCA*, 2016.

[12] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, “Cambricon-X: An accelerator for sparse neural networks,” in *MICRO*, 2016.

[13] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.

[14] Y.-H. Chen, T.-J. Yang, J. Emer, and V. 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*, 2019.

[15] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks," in *MICRO*, 2019.

[16] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis, "Procrustes: a dataflow and accelerator for sparse deep neural network training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 711–724.

[17] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[18] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1708.04485

[19] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 92–104. [Online]. Available: https://doi.org/10.1145/2749469.2750389

[20] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis, "Procrustes: a dataflow and accelerator for sparse deep neural network training," 2020. [Online]. Available:
https://arxiv.org/abs/2009.10976

[21] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," 2016. [Online]. Available: https://arxiv.org/abs/1602.01528

[22] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. 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)*, 2018, pp. 662–673.

[23] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[24] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 770–778.

[25] A. K. Smilde, P. Geladi, and R. Bro, *Multi-way analysis: applications in the chemical sciences*.
John Wiley & Sons, 2005.

[26] W. Austin, G. Ballard, and T. G. Kolda, “Parallel tensor compression for large-scale scientific
data,” in *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2016, pp. 912–922.

[27] M. A. O. Vasilescu and D. Terzopoulos, “Multilinear analysis of image ensembles: Tensorfaces,” in *European conference on computer vision (ECCV)*. Springer, 2002, pp. 447–460.

[28] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen, “Cubesvd: a novel approach to personalized web search,” in *Proceedings of the 14th international conference on World Wide Web (WWW ’05)*, 2005, pp. 382–390.

[29] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, “Speeding-up convolutional neural networks using fine-tuned cp-decomposition,” *arXiv preprint arXiv:1412.6553*, 2014.

[30] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting linear structure within convolutional networks for efficient evaluation,” *Advances in neural information processing systems*, vol. 27, 2014.

[31] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, “Tensor decomposition for signal processing and machine learning,” *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[32] J. Kossaifi, A. Khanna, Z. Lipton, T. Furlanello, and A. Anandkumar, “Tensor contraction layers for parsimonious deep nets,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR)*, 2017, pp. 26–32.

[33] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzynek, T. Norell, and Y. S. Shao, “Cosa: Scheduling by constrained optimization for spatial accelerators,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 554–566.

[34] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, “Dmazerunner: Executing perfectly nested loops on dataflow accelerators,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–27, 2019.

[35] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. 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)*, 2020, pp. 369–383.

[36] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, ser. PLDI '91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 30–44. [Online]. Available: https://doi.org/10.1145/113445.113449

[37] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. 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)*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, pp. 14–27. [Online]. Available: https://doi.org/10.1145/3352460.3358302

[38] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACRONYM	**DESCRIPTION**
PEs	Processing Elements
MAC	Multiply-and-Accumulate
NOC	Network-on-Chip
GLB	Global Buffer
SIMD	Single-Instruction-Multiple-Data