Efficient Tensor Slicing for Multicore NPUs using Memory Burst Modeling

Rafael Sousa
IC-UNICAMP
Campinas, Brazil
rafael.sousa@ic.unicamp.br

Byungmin Jung and Jaehwa Kwak
LGE San Jose Labs
San Jose, USA
{jesse.jung, jaehwa.kwak}@lge.com

Michael Frank
MagiCore
Santa Clara, USA
michael@magicore.com

Guido Araujo
IC-UNICAMP
Campinas, Brazil
guido@unicamp.br

***Abstract*—Although code generation for Convolution Neural Network (CNN) models has been extensively studied, performing efficient data slicing and parallelization for highly-constrained Multicore Neural Processor Units (NPUs) is still a challenging problem. Given the size of convolutions' input/output tensors and the small footprint of NPU on-chip memories, minimizing memory transactions while maximizing parallelism and MAC utilization are central to any effective solution. This paper proposes a TensorFlow XLA/LLVM compiler optimization pass for Multicore NPUs, called Tensor Slicing Optimization (TSO), which: (a) maximizes convolution parallelism and memory usage across NPU cores; and (b) reduces data transfers between host and NPU on-chip memories by using DRAM memory burst time estimates to guide tensor slicing. To evaluate the proposed approach, a set of experiments was performed using the Neuro-Morphic Processor (NMP), a multicore NPU containing 32 RISC-V cores extended with novel CNN instructions. Experimental results show that TSO is capable of identifying the best tensor slicing that minimizes execution time for a set of CNN models. Speed-ups of up to 21.7% result when comparing the TSO burst-based technique to a no-burst data slicing approach.**

***Index Terms*—burst-based model, convolutional neural network, NPU, mapping strategies**

I. Introduction

Deep Learning using Convolutional Neural Network (CNN) has become a significant architecture model technique that considerably increases the accuracy on many modern AI applications. The steady increase in the adoption of CNNs is driven mostly by applications in the Computer Vision domain where it addresses problems like Object Recognition [1]–[3], Object Detection [4], [5], and Video Classification [6], [7]. Other areas, like Speech Recognition and Natural Language Processing (NLP) have also benefited from the application of CNN models [8], [9].

Followed by its accuracy improvements, the size and complexity of state-of-the-art CNNs have also grown significantly. For instance, LeNet-5 [10], a model that recognizes handwritten digits, has less than 1 Million parameters, while more complex models, like InceptionV3 [11] which classifies thousands of different object categories, has more than 23 million parameters. Such increase in the model complexity and parameters size, not only demands more computational power but also produces a significant increase in the data movement between host (off-chip) and the AI accelerator (on-chip) memories thus considerably impacting energy-consumption [12] and memory traffic.

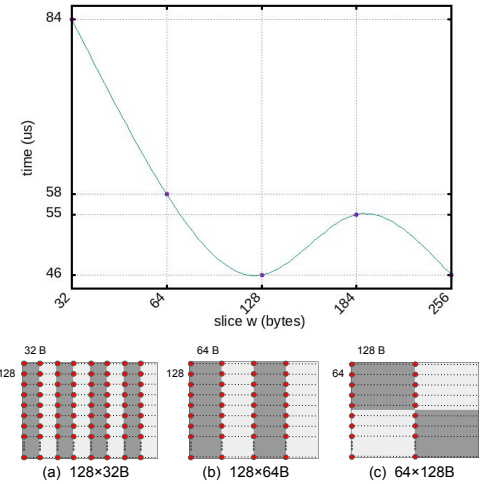


Fig. 1: Memory access with different slice shapes.

It is well-known that convolution is the most expensive operation of a CNN, accounting for the largest share of a CNN execution. Given the size of its tensor inputs and the wide variety of configuration parameters (e.g., kernel size, stride, etc), selecting the best data mapping which maximizes convolution parallelism while minimizing memory transactions is a key factor to the performance of any AI accelerator. This is particularly critical for multicore Neural Processing Units (NPUs), which have stringent (on-chip) memory constraints and need to achieve large inference throughput.

To achieve that, Convolution input tensors and weights need to be divided into slices that fit into NPU on-chip memories. Slices are brought from (slow) external DRAM to (fast) on-chip memories. Input tensors and weight slices are then used to perform Convolution, one set of slices at a time. Depending on how the slice shapes and sizes are selected, the convolution execution time can drastically change. As an example, consider Figure 1, which shows the time taken by a Convolution when using slices of different shapes. In that example, the input tensor is a single channel with 128×128 16-bit fixed-point elements (row-major) computed over a single kernel of size 1×1. In the figure, slices are represented as light/dark gray areas, and each red dot represents a (128B) *memory burst* access to the DRAM. Accessing time in a DRAM can be divided into two components: (a) CAS latency, which is the time taken to

read the first byte of a memory burst from the DRAM Row Buffer; and (b) Access latency, which is the time taken to read the following bytes of the burst. For example, reading the first byte from a 128B burst of a typical DDR3 memory takes $\sim$ 14ns, the same time it takes to read all the remaining 127 bytes of that burst. Depending on how data is sliced, memory bursts can have an enormous impact on execution time. For example, in Figure 1 the Convolution can be divided into: (a) 8 $128 \times 32B$ slices resulting in 1024 bursts (red dots) and an execution time of 84us; (b) 4 $128 \times 64B$ slices corresponding to 512 bursts and a reduced 58us execution time; and (c) 4 $64 \times 128B$ slices which require 256 bursts and 46us execution time, a 45% reduction in the convolution time when comparing to the slicing in (a). In Figure 1, slicing (c) is represented by the smallest memory access time at w = 128B. From that point on, as the width (w) of the slice continues to increase, memory access time worsens and then improves again at the next memory burst alignment (w = 256).

Although memory access coalescing is a common problem in GPU code generation, it has not been explored in the context of multicore NPU parallelism. This paper proposes a compiler optimization for multicore NPUs, called Tensor Slicing Optimization (TSO), which has two goals: (a) to maximize the parallelization of convolutions across the memories of the available NPU cores; and (b) to reduce data transfers between host and the cores' on-chip memory. This is achieved by modeling, at compile time, the memory utilization of the various NPU cores in the search for the best input/output tensor slicing which minimizes data transfers between the host and the NPU cores' memories. To evaluate this approach, a set of experiments was performed using the NeuroMorphic Processor (NMP), a multicore NPU containing 32 cores, and the TensorFlow XLA LLVM compiling toolchain.

This paper is divided as follows. Section II provides a background review. Section III describes details about the NMP accelerator. Section IV shows how to map Convolution Layers on NMP using the TSO algorithm. Section V analyzes the works related to this paper. Section VI shows the experimental results, and finally, Section VII concludes the work.

II. Background

A CNN model can be seen as a directed acyclic graph composed of multiple layers of operations in which a set of input channels (e.g., images) is processed. Among all possible layers that compose a CNN, Conv-layers account for more than 90% of a model execution time [13], and generates a large amount of data movements. This is particularly critical on architectures with small on-chip memories, like NPUs that are used to perform inference on mobile and embedded devices.

As shown in Figure 2, a Conv-layer uses N input feature maps (*IFM*) of size H (height) $\times$ L (width) and a set of pre-trained weights. The weight set (KS) is a set of M multidimensional (e.g., 3-D) arrays kernels/filters of size $N \times K \times K$. Each filter slides over the IFMs performing a 3-D convolution with a stride factor of S. After sliding over the entire input image (IFMs), an $R \times C$ output feature map

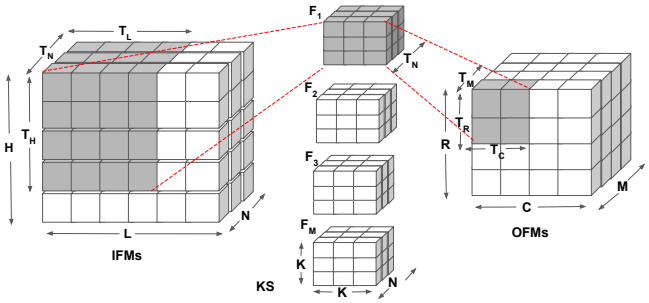


Fig. 2: A Tiled Convolutional Layer.

(*OFM*) is generated. A set of M OFMs results after applying all M filters in KS to the IFMs.

Different Conv-layers in a CNN usually have different kernel sizes, different numbers of IFM and OFM with distinct sizes, and variable strides. For instance, the first Conv-layer from Inception-V3 [11] has $3 \times 299 \times 299$ elements in its input. If we consider the data type of the input as 16-bits fixed-point, the size of the input becomes 524KB. This is impractical to store at once into the constrained on-chip memories available on typical NPUs, particularly those used in mobile devices. Because of that, data tiling is a mandatory task in the computation of a Conv-layer.

Data tiling of a Conv-layer consists in partitioning its IFM and OFM data maps into small tiles and dividing the filters in KS so that each IN, OUT and W tiles fit together at the same time into the NPU on-chip memories. Figure 2 shows how the IFM and OFM data maps are respectively divided into $IN = (T_N, T_H, T_L)$ tiles, and $OUT = (T_M, T_R, T_C)$ tiles. Notice that the dimensions T_H and T_L of the IFM can thus be computed using the dimensions T_R and T_C of the OFM thru Equation 1.

$$\begin{aligned} T_H &= (T_R - 1)S + K \\ T_L &= (T_C - 1)S + K \end{aligned} \tag{1}$$

where K and S are the kernel size and stride, respectively. Moreover, if the filters/kernels in KS do not fit into their respective NPU on-chip memories, KS is also partitioned into $W = (T_M, T_N, K, K)$ tiles, where T_M is the number of filters and T_N the number of channels to be loaded from each filter.

Different tile shapes can be explored when partitioning a convolution to execute on an NPU. Each tile shape leads to different memory accesses and usage of the resources available on the NPU. Besides that, different scheduling strategies can be explored, each one with a specific memory access pattern that leads to different data-reuse (more details in Section IV).

III. The NMP Architecture

This paper uses the NeuroMorphic Processor (NMP) by LG Electronics (LGE) as a compiling target. The key idea behind the NMP architecture is to use RISC-V ISA Extensions to design relevant CNN instructions like Conv-layers, FC-layers, Pooling layers, Element Wise operations, etc.

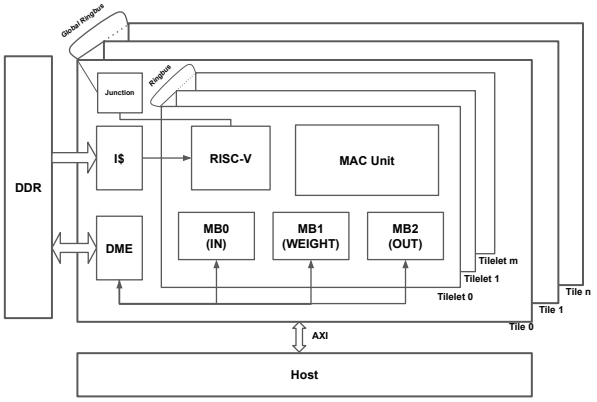


Fig. 3: NMP Architecture.

The NMP architecture (Figure 3) is a multicore NPU that contains an ARM57 processor that works as a host for a set of multiple Tile (TLE) processors, containing each a set of Tilelet (TLT) cores. Each TLT has one RISC-V core, three on-chip (scratchpad) memories, namely MB0, MB1 and MB2, which respectively store the IN, W and OUT tiles from the IFM, KS and OFM data maps. Besides that, each TLT is also equipped with a MAC acceleration unit to execute CNN operations. The MAC unit execution is triggered by the RISC-V core and is capable of executing 8- and 16-bits fixed-point operations. If set to execute with 8-bits fixed-point data, the unit can compute 16 MACs in parallel; otherwise, only 8 MACs are available to operate on 16-bit fixed-point operands simultaneously.

The data transfers between NMP and host happens through a Data Movement Engine (DME) module, as shown in Figure 3. The host communicates with the NMP through an AXI interface and data can be shared between TLTs of different TLEs by using a Global Ringbus. The TLTs of a TLE also have their own Ringbus to communicate data between them. The instructions executed by the RISC-V cores are fetched from the host memory and stored into a cache instruction, which is shared between the TLTs of the same TLE.

To execute any computation on NMP, the model is first compiled from a *protobuf* file using the TF-XLA compiler. The execution of the compiled model starts on the RISC-V cores of each TLT, each one of them executing independently of each other. To invoke any computation on the MAC units, one of the following extended instructions are run by the RISCV-core: nmp_conv2d, nmp_veop, nmp_pool, nmp_activation and nmp_percept. To load data from the DRAM to the TLT on-chip memory, the RISC-V has to execute a data movement instructions (nmp_load and nmp_load3d) so that the DME is invoked to do the job. Data is brought back from each TLT on-chip memory to the host DRAM by executing the RISC-V extended nmp_store and nmp_store3d instructions. Despite the fact that each one of the TLTs of a single TLE executes its computation independently of the others, NMP has support to a special *multicast* load instruction which executes on a single TLT but enables all TLTs of that TLE to load the same data from the DRAM.

The NMP architecture used in this work is composed of 4 TLEs, each containing 8 TLTs (RISC-V + MAC unit). Each TLT has three on-chip memories of size 8KB each. With an operating frequency of @1GHz, NMP (all the 32TLTs together) has a theoretical performance peak of either 512- or 256-GMACs/sec when executing 8- or 16-bit fixed-point, respectively. The DRAM memory is a DDR3 that operates on a 1066MHz clock rate (DDR3-2133 – 17GB/s). For this edition of the architecture, NMP does not enable MAC/LOAD overlap. For future NMP-architectures, we anticipate the addition of extra on-chip (dual-port) memories that will allow the compiler to software-pipeline MAC/LOADs.

IV. NMP Mapping Strategies

Data transfer performance is known to be one of the most important goals when compiling CNNs to accelerators [12]. There are different ways to transfer data from the host to an accelerator memory during the execution of a Conv-layer. If not done properly, the number of data movements may considerably increase, thus impacting performance and energy consumption. This paper proposes a search space exploration optimization algorithm called *Tensor Slicing Optimization* (TSO) that seeks to identify the best TLE/TLT data partitioning that minimizes the number of memory transfers during the execution of Conv-layers.

A. TSO Algorithm

TSO works by exhaustively exploring the solution space in the search for the best convolution tiling/scheduling strategy that minimizes execution time. It first slices the input tensor of the convolution (IFMs) and its corresponding filters (KS) among the TLE processors of the NMP so that each TLE computes a different slice of the Conv-layer's output. After that, each TLE slice is further partitioned into multiple tiles so as to distribute the computation among the TLT cores of the corresponding TLE processor. These two steps of the TSO algorithm are detailed below. Before moving further please consider from now on that every mention to *slice* refers to a TLE data partitioning and every mention to *tile* refers to a TLT partitioning.

Initially (refer to Algorithm 1), TSO takes as input the set of convolutions of the model ($CONVS$) and the number of TLEs ($\#TLE$) and TLTs ($\#TLT$) of the architecture (line 1). It then iterates over all convolutions (line 2) and initializes a map which stores the best TLE slice and TLT tile for that specific convolution (lines 4). Then for all possible TLE slicing strategies p available in $PART_{TLE}$ (line 6, see Subsection IV-B for details), the algorithm uses a call to function $TLESlicing$ to divide the convolution IFMs and KS data across the TLEs. $TLESlicing$ returns tuple $slice = (TLE_R, TLE_W)$, where TLE_R refers to the part of the OFM (rows) that is generated from the slice of the IFMs designated to the TLE processor, and TLE_W a subset of the KS filters that will run on that TLE processor.

Remember that each TLE processor in NMP has a set of TLT cores, and thus for each TLE slice produced in line 8, the slice data needs to be divided among its corresponding TLT

Algorithm 1 Select best TLE/TLT mapping

```
1: function TSO(CONVS, #TLE, #TLT)
2:     for each conv ∈ CONVS do
3:         ▷ Let conv = (IFM, KS, OFM)
4:         map[conv].bestTile.time ← ∞
5:         ▷ Let PART_TLE = {KS, KS&OFM, OFM}
6:         for each p ∈ PART_TLE do
7:             ▷ Let slice = (TLE_R, TLE_W)
8:             slice ← TLESLICING(p, conv, #TLE)
9:             ▷ Let PART_TLT = {IS, OS, WS}
10:            for each q ∈ PART_TLT do
11:                ▷ Let tile = (IN, W, OUT, time, schedule)
12:                tile ← TLTTILING(q, conv, slice, #TLT)
13:                if tile.time < map[conv].bestTile.time then
14:                    map[conv].bestSlice ← slice
15:                    map[conv].bestTile ← tile
16:    return map
```

Algorithm 2 TLE Slicing

```
1: function TLESLICING(p, conv, #TLE)
2:     rows ← conv.R
3:     filters ← conv.M
4:     if p = KS then
5:         TLE_W = ⌈filters/#TLE⌉
6:         TLE_R = rows
7:     if p = KS&OFM then
8:         TLE_W = ⌈filters/(#TLE/2)⌉
9:         TLE_R = ⌈rows/(#TLE/2)⌉
10:    if p = OFM then
11:        TLE_W = filters
12:        TLE_R = ⌈rows/#TLE⌉
13:    return (TLE_R, TLE_W)
```

cores. Hence, for each TLT scheduling strategy q (line 10, see Subsection IV-D for details), TSO computes the best TLT tile for the current TLE data slice using a call to $TLTTiling$ (line 12). This function takes as input the TLT scheduling strategy q, the convolution data ($conv$), the current TLE $slice$, and the number of TLTs ($\#TLT$). It then determines the best tiling of the TLE data among the TLT cores. The $TLTTiling$ function returns tuple $tile = (IN, W, OUT, time, schedule)$, where IN, OUT are the tiles of the IFM and OFM data maps assigned to the TLTs of that TLE, and W is a tile that contains a subset of the filter in KS.

The tuple also returns an estimate of the time taken to compute the convolution using that specific combination of TLE slice and TLT tile for the best possible scheduling ($schedule$) strategy (see Subsection IV-E for details). To achieve that, it takes into consideration the cost to load the IN and W tiles from DRAM into the (on-chip) TLT memories MB0 (IN) and MB1 (W), respectively, and the time to store the OUT from the MB2 TLT (OUT) memory back to the host DRAM. Moreover, $time$ also includes the time taken by each evaluated partitioning to run on the MAC Unit using the various scheduling alternatives (see Subsection IV-D for details).

After returning from $TLTTiling$, TSO compares (line 13) the estimated $time$ for the evaluated partitioning with the best time ($map[conv].bestTile.time$) found so far for that specific convolution. It then stores it into the appropriate map entry (i.e. $map[conv]$) the corresponding TLE slice (line 14) and TLT tile (line 15). Finally, the map containing the best slices/tiles for each convolution is then returned (line 16), so it can be used later by the code generator to synthesize and schedule the code for the TLT cores.

B. TLE Partitioning

The first step in the TSO optimization is to divide the filters in KS among the TLEs and define which part of the OFM (rows) the selected filters will compute. This is done according to the partitioning set defined in Algorithm 1 – $PART_{TLE} = \{KS, KS\&OFM, OFM\}$, where KS, KS&OFM and OFM are partitioning strategies computed by Algorithm 2.

KS partitioning – In the first partitioning scheme (line 4), only the convolution filters in KS are divided into slices among the TLEs (line 5). In terms of data replication, all the $R \times C$ elements of an OFM have to be computed by the TLE, which requires loading the entire IFMs at runtime on each TLE. This partitioning scheme usually works well on the last Conv-layers of a CNN model, given the increase in the number of filters as well as in their channels' depth. Dividing the filters may reduce the number of data transfers between DRAM and the NMP.

KS and OFM partitioning – The second partitioning scheme (line 7) divides both the filters and the OFM rows among the TLEs. For the NMP used in this work (#TLE = 4), it slices the OFM rows into two sets as well as the filters in KS, which are then combined to generate one slice for each TLE. This TLE partitioning scheme reduces the data transfer over the IFMs, compared to the first TLE partitioning scheme, but increases the loads over the filters, given that more filters are assigned to the slices. This scheme usually works better in the middle layers of a CNN, where both the KS and IFMs data have similar sizes.

OFM partitioning – Finally, the third partitioning only divides the OFM rows among the TLEs (line 12). Given that the filters in KS have to be loaded by each TLE, this partitioning scheme usually works better on the first Conv-layers, since the IFMs are bigger when compared to the filters in KS.

Since the NMP board used to collect the experiments for this paper does not have a global shared-buffer (shared among the TLEs), we have not considered this feature in designing TSO. However, TSO can be easily extended to consider a global shared-buffer since different slices of the IFMs/KS from different TLEs may be the same.

C. TLT Partitioning

After choosing a TLE partitioning scheme, the workload of each TLE is divided among their corresponding TLTs by means of a call to function $TLTTiling$ in line 12 of Algorithm 1. $TLTTiling$ takes as input the TLT scheduling strategy (q), the convolution data ($conv$), the TLE slice ($slice$) resulting in line 8 of Algorithm 1 and the number of TLTs at each TLE ($\#TLT$). It then produces as output the

Algorithm 3 TLT Tiling

```
1: function TLTTILING(q,conv,slice,#TLT)
2:     ▷ Let tile = (IN, W, OUT, time, schedule)
3:     bestTile.time ← ∞
4:     for T_R ← 1 to slice.TLE_R do
5:         for T_C ← 1 to conv.C do
6:             for T_N ← 1 to conv.N do
7:                 T_M ← GETFILTERS(T_R, T_C, q, slice.TLE_W, #TLT)
8:                 W ← GENTILE_W(T_M, T_N, conv, q)
9:                 OUT ← GENTILE_OUT(T_M, T_R, T_C)
10:                IN ← GENTILE_IN(T_N, T_R, T_C, conv)
11:                (time, schedule) ← CALCTIME(IN, W, OUT, q)
12:                if time < bestTile.time then
13:                    bestTile ← (IN, W, OUT, time, schedule)
14:    return bestTile
```

tuple $(IN, W, OUT, time, schedule)$ which will be used to generate code for the TLTs.

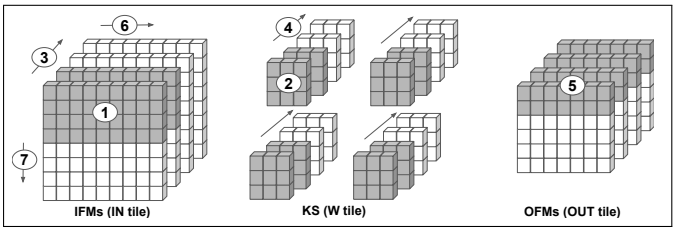
Initially (refer to Algorithm 3), $TLTTiling$ initializes variable $bestTile.time$ with infinity as it will store the smallest (estimated) execution time of all possible tiles visited by the function. To achieve that, a sequence of three nested loops (lines 4-6) generate the values T_R, T_C and T_N that are used to explore all possible IN, W and OUT tiles shapes that can be formed from a TLE slice. But before computing the IN and OUT tiles for that TLE slice, the convolution filters in TLE_W need to be divided among the various TLTs. This is done in line 7, which also determines the maximum number of filters (T_M) that can fit into the MB1 (W) memory of a TLT, and in line 8, which generates the corresponding W tile. In the case of an unbalanced filter partitioning, the remaining filters are spread among the TLTs which have the lowest IDs. This is followed by calling functions to generate the OUT tile ($GenTILE_{OUT}$ in line 9) and IN tile ($GenTILE_{IN}$ in line 10). These two functions also check if the tiles OUT and IN respectively fit into memories MB2 (OUT) and MB0 (IN) of a TLT, as shown in Equation 2, where type stands for either 8- or 16-bit fixed-point. The functions between lines 8-10 also calculate the number of times the IN, OUT and W tiles have to loaded/stored from/to the DRAM to cover all the workload of a TLE slice (more details in Subsection IV-D).

$$\begin{cases} IN = T_N \times T_H \times T_L \times type \leq MB0 \\ W = T_M \times T_N \times K \times K \times type \leq MB1 \\ OUT = T_M \times T_R \times T_C \times type \leq MB2 \end{cases} \quad (2)$$

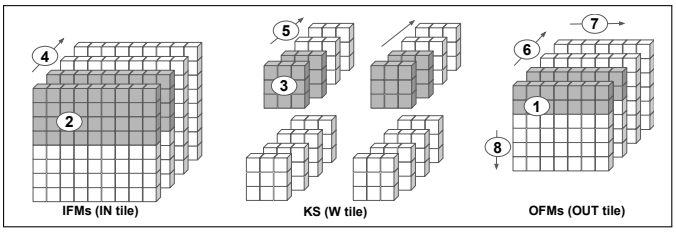
Tiles IN, W, OUT and the tiling strategy q, are then passed to function $CalcTime$ (line 11), so it can estimate the best $schedule$ and $time$ to compute the TLE slices using the generated tiles (more details in Subsection IV-E). Finally, the algorithm tests if the $time$ computed for the current tiling is smaller than the $bestTile.time$ seen so far, and if so, it updates the $bestTile$.

D. Scheduling

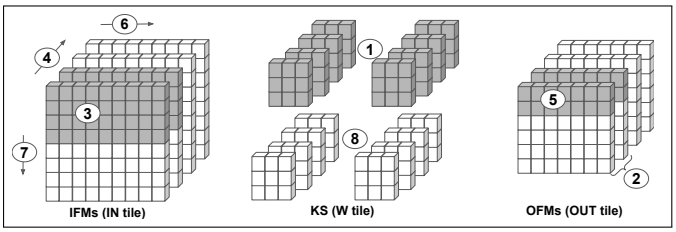
With the data divided among the TLEs/TLTs, different scheduling strategies – Input Stationary (IS), Output Stationary (OS) and Weight Stationary (WS) – may be used by the TLT cores to execute the convolution, each providing a different memory access pattern ($schedule$). Given that the data-transfers of the mapping strategies presented herein can be determined statically, we compute the number of accesses to the DRAM required by each one of them, according to their data-flow patterns, so as to determine the one that can result in the best data reuse.



(a) Input Stationary



(b) Output Stationary



(c) Weight Stationary

Fig. 4: Scheduling Strategies used on NMP.

Input Stationary (IS) – is a scheduling strategy that focuses on reusing the IN tiles. Figure 4a shows the execution flow of IS. The first step (① in Figure 4a) is to load the IN tile from the DRAM into the NMP MB0 on-chip memory; then, the W tile is also loaded ② from the DRAM into MB1. To make full reuse of the IN tile, the W tile has to include all the filters designated to the TLT – even if just a small part of each one of them. With the IN and W tiles already loaded, the MAC Unit executes the convolution on them. The result is stored into the MB2 (OUT) memory which, at this point, only contains a partial sum of the Convolution – the final result of the OUT tile is only generated after computing all elements through the depth of the IFMs. To do that, multiple IN ③ and W tiles ④ may be required to be loaded while going through the channel (depth) direction. After computing and accumulating the results, the OUT tile is ready to be stored into the DRAM ⑤. After that, a new IN tile is loaded, going first on the width ⑥ and then on the height ⑦ directions of the IFM – for each one of them, the same W tiles are reloaded again and again.

Given the access pattern performed by IS when load-

ing/storing data from/to the DRAM, one can use Equation 3 to identify, for each tile (IN, W and OUT tile), the number of times it is required to load/store each one of them to cover the entire computation of a Conv-layer over the TLEs/TLTs. The α_{in} and α_w symbols denote the number of times the TLEs/TLTs have to load the IN and W tiles from the DRAM to compute an entire Conv-layer. The α_{out} stands for the number of times the OUT tiles are stored to the DRAM.

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \\ \alpha_w = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \end{cases} \quad (3)$$

Output Stationary (OS) – is a mapping strategy that prioritizes the generation of the OUT tiles, no matter if the same IN and W tiles have to be loaded multiple times from the DRAM into their respective on-chip memories. Figure 4b shows the execution flow of OS. First, based on the OUT tile ① dimensions, the corresponding IN ② and W ③ tiles are loaded from the DRAM into their respective on-chip memories to compute a convolution on them using the TLT's MAC Unit. Given that typically the on-chip memories have not enough space to accommodate all the required input data, multiple IN ④ and W ⑤ tiles have to be loaded using the channel (depth) direction. After finishing the computation of an OUT tile, it is stored into the DRAM and a new OUT tile starts to be computed using the channels' (depth) direction ⑥. After that, the other OUT tiles are computed by following first the OFM's width ⑦ and then its height ⑧. The number of times each IN, W and OUT tile have to be loaded/stored from/to the DRAM is defined in Equation 4.

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_w = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{M}{T_M} \right\rceil \end{cases} \quad (4)$$

Weight Stationary (WS) – is a mapping strategy that focuses on loading the filters from the DRAM only once and reuse them as the convolution tiles are computed. Figure 4c shows the execution flow of WS. First, the T_M filters in W tile are loaded from the DRAM into the MB1 on-chip memory ①. In this strategy, each loaded filter includes all its N channels. The loaded filters are then reused until the resulting OFMs are computed ②. Prior to executing the nmp_conv2d instruction, an IN tile is loaded from the DRAM ③. Multiples loads of an IN tile along the channels' depth may be required ④, each computing and storing partial results that will later be accumulated to form the final OUT tile ⑤, so it can be stored back to the DRAM. This is followed by loading other IN tiles in sequence over the width ⑥ and then over the height ⑦. This proceeds until all the OFMs of the respective filters in W tile are computed. After that, a new W tile with other filters may be required to be loaded ⑧ to compute their corresponding OFMs – at this point, for each iteration, the same IN tiles are again loaded. Equation 5 defines the number of required data transfers to/from the DRAM to cover the entire Conv-layer.

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_w = \#TLEs \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{M}{T_M} \right\rceil \end{cases} \quad (5)$$

E. Estimating Time

In order to decide which slicing strategy is the best among those discussed in the sections above, for each convolution TSO combines multiple solutions from the search space $<PART_{TLE}, PART_{TLT}, T_M, T_N, T_R, T_C>$, and estimates the time taken by each valid combination to select the one which provides the best performance. This estimate has the following components, listed in increasing order of their contribution to the convolution execution: (a) the time required to run the RISC-V instructions at each TLT; (b) the time needed to perform the MAC unit operations on the slices; and (c) the time required to load/store data between the DRAM and the NPU on-chip memories. An estimate for the execution time is calculated by function *CalcTime* (defined in Algorithm 3 – line 11), which sums the time of each component of the execution according to Equation 6.

$$T_{CONV} = T_{MAC} + T_{DRAM} + T_{SW} \quad (6)$$

where T_{MAC}, T_{DRAM} and T_{SW} stands for the time taken by the MAC Unit, the time taken to transfer data between the NPU's on-chip memories and DRAM and the time taken to execute the RISC-V instructions, respectively. Since T_{SW} is not significant (usually less than 5% of the total execution), we will not cover it in detail in this paper.

The time T_{MAC} is calculated according to the number of Multiply-accumulate operations (MAC operations) of a Conv-layer. In the first step (see Equation 7), it is identified the number of MAC operations required to compute a channel over the IFMs, which is then divided by the number of MAC operations that a MAC unit can execute at each cycle. After that, the other feature maps are then considered to compose the estimated time of the entire tile ($Tile_{MAC}$). Given the time taken of a single tile, it is then possible to estimate the total time required to compute the entire Conv-layer's workload, which is distributed over all the TLTs cores ($\#TLEs \times \#TLTs$) in NMP (see Equation 8).

$$Tile_{MAC} = T_N \times T_M \times \left(\left\lceil \frac{T_R \times T_C \times K \times K}{\#MACs} \right\rceil \right) \times \frac{1}{Freq} \quad (7)$$

$$T_{MAC} = \frac{1}{\#TLTs} \times \left\lceil \frac{M}{T_M} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times Tile_{MAC} \quad (8)$$

Algorithm 4 Estimate the time taken by Data Transfer

```
1: function CalcDataTransfer(tile, conv, arch)
2:     type ← arch.type
3:     BW ← arch.BW
4:     if model = TSO-burst then
5:         nbursts ← CalBurstCount(tile, conv, type)
6:         cas_latency ← nbursts * CAS
7:         tile_size ← GetTileSize(tile, type)
8:         transfer_time ← tile_size/BW
9:         total_time ← transfer_time + CAS_latency
10:    else
11:        ▷ TSO-noburst
12:        tile_size ← GetTileSize(tile, type)
13:        total_time ← tile_size/BW
14:    return total_time
```

To evaluate the data transfer between the on-chip memories and DRAM, TSO uses two approaches to estimate the time taken to load/store the IN, W and OUT tiles. They are: (a) TSO-burst, a burst-based model that estimates the tile's DRAM transfer time by determining the number of memory bursts and the total CAS and access time it takes; and (b) TSO-noburst, a data volume-based model, which estimates the DRAM transfer time solely based on the size of the tile's data and the memory bandwidth. Algorithm 4 describes how the DRAM transfer time is computed using both approaches. For the sake of explanation of Algorithm 4 assume a DDR3 memory with CAS latency (us), BW bandwidth (Bytes/sec) and $BURST$ size in bytes (e.g. 128B)

In order to estimate the time taken by data transfers (see Equation 9) in both approaches (TSO-burst & TSO-noburst), the number of memory accesses each tile requires (α_{in}, α_w and α_{out}) is combined with a call to function *CalcDataTransfer* (defined in Algorithm 4 – more details below). The estimated time of each tile is then composed to form T_{DRAM}.

$$\begin{aligned} T_{DRAM} = \alpha_{in} * CalcDataTransfer(IN, conv, arch) + \\ \alpha_w * CalcDataTransfer(W, conv, arch) + \\ \alpha_{out} * CalcDataTransfer(OUT, conv, arch) \end{aligned} \quad (9)$$

Burst-based data transfer (TSO-burst) – The key idea behind TSO-burst is to determine the number of bursts taken by each access to a tile row to use it to determine an estimate for the DRAM access time of the tile. For instance, consider an IN tile containing T_N (channels) $\times$ T_H (rows) $\times$ T_L (columns) where each entry has 16-bit (2B) elements. Given that the channel is laid out in row-major, loading the first element of a row takes time CAS, while loading the remaining elements $\sim (2 \times (T_L - 1))/BW$. Thus, an IN row takes $CAS+ \sim (T_L - 1)/BW$ to load. This is true if the size of the row ($2 \times T_L$) is smaller than $BURST$ bytes. Otherwise, other memory bursts may occur when loading the row, and additional CAS penalties will impact the time.

Algorithm 4 is used to estimate the execution time when convolution $conv$ is divided into tiles $tile$ on architecture $arch$. Initially, the tile data size $type$ (e.g. 16-bit fixed-point) (line 2) and the DRAM memory bandwidth BW (line 3) are extracted from the $arch$ data structure. Next (line 4), the algorithm selects the memory transfer model (e.g., TSO-burst) and uses a call to function *CalBurstCount* (line 5) to determine the number of bursts ($nburts$) required to load all the T_H rows of an (IN) tile. Then, the impact of the CAS latency is computed into $cas\_latency$ (line 6) and the size of the tile ($tile\_size$) is determined in line 7 by calling function *GetTileSize*. The time to transfer all the data in a tile ($transfer\_time$) is then determined (line 8), and finally, the total time to load the tile is computed (line 9) and returned (line 14).

TSO-burst does not make any assumptions about the external DRAM or memory-controller designs, besides the existence of burst-based accesses typically found in these memories. The memory-controller found in the NMP board follows the ARM-bus protocol. Besides CAS-latency, other DRAM parameters (e.g., Trcd/Trp/Tras) could also be included to improve the precision of data transfer modeling. Nevertheless, since CAS-latency is the most relevant of these DRAM parameters Algorithm 4 focused only on it.

Volume-based data transfer (TSO-noburst) – This approach is typically used by all previous works which address this problem. As shown in lines 12-13 of Algorithm 4, it estimates the tile time by considering only the time to transfer the tile data (line 12) and not the impact of the CAS latency of the tile's memory bursts.

V. Related Works

Maestro [14] and Timeloop [15] uses analytical modeling that evaluates different mapping configurations – dataflow strategy, data-reuse, tile size, etc; to estimate the runtime for different configurations. While Maestro designs some annotations to classify the loops either as temporal or spatial, Timeloop analysis the nested loops to apply the transformations on them. For both, given a DNN layer (e.g., a Convolution and its information), hardware configuration (number of PEs, on-chip memories size, etc), the dataflow strategy, these approaches estimate the runtime performance, energy and power. Given the easy use of Maestro, different solutions have adopted its annotations to estimate computation [16], [17]. As an example, Marvel [16] uses the Maestro notations and has for main goal the reduction of the search space by decoupling the analysis of the cost model of the accesses to the on-chip/off-chip sub-spaces. Timeloop and Maestro model Spatial DNN Accelerators, i.e., FPGA-based architectures, in which the inner-loops of a Convolution are unrolled and then synthesized into PE array (MAC units) which run in a synchronized fashion to leverage on data-sharing between them through inter-PE communication. Similar to Marvel and Timeloop, our work also performs cost modeling and design space exploration, but contrary to them, we model execution on multicore NPU architectures and not on FPGA designs.

Tu *et al.* [18] and Hu *et al.* [19] proposed an FPGA-based accelerator capable of reconfiguring its resources to increase data reuse. They used the concept of Input Stationary (IS), Weight Stationary (WS), and Output Stationary (OS). Besides that, they propose a novel approach called Hybrid Stationary (HS) that leverages on these concepts to find an optimal configuration for each Conv-layer. Although their work has

some similarity to ours, instead of mapping the operations to an array of PEs, we consider an architecture (NMP) with multiple cores where each core has an accelerator which runs independently of the others. Besides that, our search space exploration algorithm considers different tile shapes based on memory bursts, and not just square shapes that fit into the hardware topology.

To select different tile sizes, loop order, unroll factor, etc, TVM [20] uses a machine-learning cost model, which does not require hardware information, and periodically learns from previous predictions to search for an improved partitioning. To the best of our knowledge, and from the available public literature, TVM has not shown any results for multicore NPUs like NMP, generating code only for FPGAs, embedded CPUs and server CPUs.

To improve data reuse, some works use polyhedral-based optimization techniques [21], [22]. Ma *et al.* [23] describes a performance model that implements Output Stationary (OS). Chen *et al.* [24] describes an approach called Row Stationary (RS) that minimizes data movement by exploiting data reuse through inter-PE communication. The tile selection on those works is usually selected from the use of a roofline-based model [25], [26]. Compared to our work, their roofline model only considers the number of memory accesses without taking memory burst into consideration. Stoutchinin *et al.* [27], on the other hand, uses a technique called reuse distance, which aims to identify the memory footprint which is required to accommodate the Convolution's data into the on-chip memory, which varies up to 512KB. His work only considers data reuse over the on-chip memories without taking into consideration DRAM accesses.

VI. Experimental Results

In order to validate the TSO approach, a set of experiments was executed on an NMP board equipped with 4 TLEs, each having 8 TLTs. Each TLT contains three 8KB MB on-chip memories (MB0–MB2). To evaluate TSO, we used 5 CNN image classification models: InceptionV3 [11], LeNet [10], MobileNetV2 [28], ResNet-50 [2], and SqueezeNet [29]. We have also applied TSO to an object detection application - a YOLO-based model [30]. All the models were compiled by our TF-XLA compiler using a quantization pass set to 16-bit fixed-point. The accuracy achieved by each of the image classification models on NMP is shown in Table I.

TSO-burst vs TSO-noburst

We compared TSO with the burst-based modeling activated (TSO-burst) and without it (TSO-noburst), as the latter is a common approach found in most previous works. The speedup of TSO-burst over TSO-noburst ranges from 7.4%, for YOLO, up to 21.7%, for InceptionV3, as shown in Figure 5. The main improvements from using TSO-burst come when the IFMs are divided into IN tiles. This happens because TSO tends to select larger tiles on the width (row-major) direction. By selecting larger tiles, TSO minimizes the number of required bursts, thus reducing the impact of the CAS latency on the memory

Model	Google (FP)		Platform	Top-1	Top-5
	Top-1	Top-5			
InceptionV3	77.9%	93.8%	CPU (FP)	74.5%	93.7%
			NMP	74.4%	92.3%
LeNet	99.9% (Keras)	100% (Keras)	CPU (FP)	98.5%	100%
			NMP	98.5%	100%
MobileNetV2	71.8%	90.6%	CPU (FP)	70.2%	90.2%
			NMP	70.3%	90.4%
ResNet-50	75.2%	92.2%	CPU (FP)	71.7%	90.5%
			NMP	71.3%	90%
SqueezeNet	49%	72.9%	CPU (FP)	48.3%	72.3%
			NMP	48.7%	71.6%

TABLE I: Model accuracy on CPU (FP32) and NMP (16-bit fixed point).

Model	TSO (us)		TLE partitioning fixed (us)			TLT scheduling fixed (us)		
	Burst	No Burst	KS	KS& OFM	OFM	IS	OS	WS
InceptionV3	72686	88478	82370	81367	91408	73706	93731	101641
LeNet	199	231	202	224	231	199	233	228
MobileNetV2	14030	15470	17765	16696	17571	14084	17387	15155
ResNet-50	55927	62375	62077	63118	78844	59714	71905	77535
SqueezeNet	12504	13713	16579	14134	12993	12908	15840	13452
YOLO	53271	57225	56591	55713	63550	68530	58592	55375

TABLE II: Model Execution times for TSO (burst and noburst) and fixed strategies.

Fig. 5: TSO-burst speedup over TSO-noburst.

access time. By prioritizing bursts on the width direction, TSO maximizes the usage of the bursts, as it improves memory access coalescing. For the case of TSO-noburst, the tiles are selected so as to reduce the number of bytes loaded from the DRAM to NMP. This approach is adopted by most solutions that have been proposed so far in the literature [18], [19], [22], [25]. Contrary to those, the TSO-burst technique proposed in this paper takes into consideration DRAM access coalescing to estimate the time taken to LOAD/STORE data from memory, thus resulting in better partitioning and improved performance.

As an example, consider the 5th Conv-layer of InceptionV3, which has 80 IFMs of size 73x73 each, and 192 filters of size 80x3x3. The shape of the IN tile selected by TSO-burst has size 14x4x73 ($T_N \times T_H \times T_L$). Since the IN tile of the TSO-burst takes the whole width L (73) of a channel, it results in a sequence of 584 bytes aligned sequentially on the DRAM ($4 \times 73 \times 2B$), which requires 5 memory bursts for each tile's channel. In total, when considering the 14 channels of that tile, the 5th Conv-layer requires a total of 70 memory bursts. On the other hand, TSO-noburst selects a tile of size 16x11x20, which corresponds to only 20 bytes aligned on the DRAM, thus resulting in one memory burst for each row. Given that

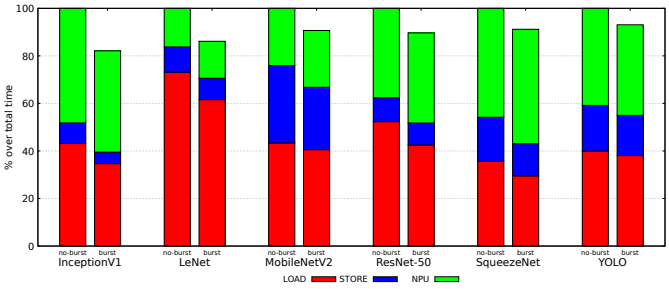


Fig. 6: TSO-burst vs TSO-noburst: Execution breakdown.

the IN tile has 16 channels, each with 11 rows, it requires a total of 176 memory bursts, which is more than double what is needed to load the TSO-burst tile. For that specific 5th Conv-layer, TSO-burst reduces the tile execution time by 28%.

The solution provided by TSO-burst aims to reduce the total time taken for data transfer operations. To illustrate that, refer to Figure 6, which shows for each model two bars representing the breakdown of the percentage of computation time spent in LOAD, STORE, and MAC operations with respect to the total execution time. For the TSO-burst's bars, the percentage of the execution time is calculated with respect to the TSO-noburst total time. As shown in Figure 6, when TSO-noburst is used, the percentage of the LOAD+STORE transfer time ranges from 51.83%, for InceptionV3, up to 83.80% for LeNet. On the other hand, when TSO-burst is used to model memory access during TF-XLA compilation, the time taken by LOAD+STORE operations decreases from 7.08% to 23.71% for YOLO and InceptionV3, respectively.

Roofline Model

To evaluate the performance of the code resulting from using TSO on the Conv-layers of each model executed on NMP (TSO-burst), we used the Roofline Model shown in Figure 7. In the graph, the y-axis presents the Multiply-and-accumulate (MAC) throughput (in GMACs/sec) achieved by the architecture and the convolution execution. In the x-axis is the Operational Intensity, which stands for the number of MAC operations executed for each byte that is loaded from the DRAM. The blue lines in the graph represent the theoretical roofs for both the MAC throughput (horizontal line) and DRAM bandwidth (sloped line) that can be respectively achieved by the NMP engine and the memory system. To better evaluate the real performance of the system two additional experiments were undertaken to measure these parameters. This is required given that other architecture components can impact their values. The black lines in the graph represent these measurements. As shown, the measured MAC throughput reaches a roof of 192 GMACs/sec, represented by the horizontal black line. A number of issues can explain this reduction. For example, the NMP device used in this work has single-ported on-chip memories, and thus TLTs that are waiting for data get idle without using its MAC Unit. As for the memory bandwidth (the sloped black line) the measured value is also reduced. This can be explained by the fact that the DRAM bandwidth is constrained by a single DMA engine

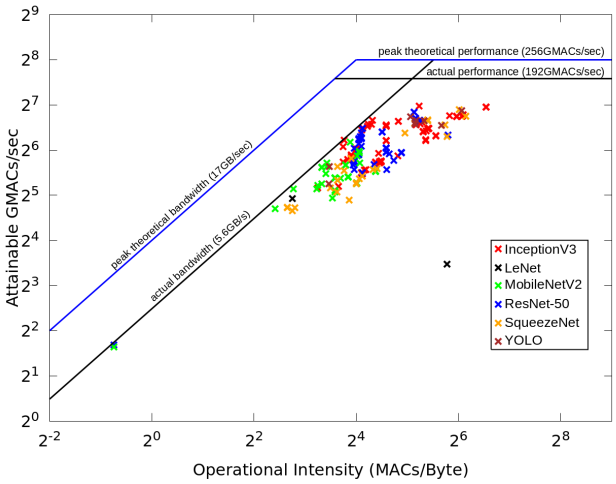


Fig. 7: Roofline model for NMP architecture (TSO-burst).

per TLE which has to simultaneously serve all 8 TLT cores. In order to evaluate the performance resulting from TSO, we plotted one point in Figure 7 for all convolutions in the models. As shown, most of the convolutions reach either the roof limited by the (measured) memory bandwidth (sloped black line), or approach the roof defined by the MAC throughput (horizontal black line). This makes it clear that TSO produces code which approaches the maximum performance of the architecture.

Fixed TLE/TLT Partitioning

Fixed TLE partitioning – in this experiment, the compiler was set to generate code which fixes each TLE slicing strategy described in Subsection IV-B for all Conv-layers of a model. The experiment works as follows. The compiler identifies, for the fixed TLE slicing, the best TLT tiling/scheduling strategy (IS, OS and WS). The result of this experiment is shown in Table II which reports the time of the model compiled with TSO (burst mode) with the fixed TLE slicing. For the *KS* case, TSO achieves a speedup of up to 32.6%, for SqueezeNet. SqueezeNet does not perform well for TLE slicing since most of its Conv-layers have IFMs larger than the size of the filter set (KS). For *KS&OFM*, TSO speedup reaches up to 19%, for MobileNetV2. This TLE strategy usually works better for the Conv-layers that have similar sizes for both IFMs and weights (KS). For *OFM*, TSO speedup is 41.0%, for ResNet-50. For most of ResNet-50's Conv-layers, the size of the weight set KS is larger than the size of the IFM data maps. As a result, TSO outperforms the best fixed TLE slicing strategy.

Fixed TLT partitioning – in this experiment, the compiler is set to generate code which fixes one of the three TLT scheduling strategies (IS, OS, and WS) for all the model's Conv-layers. The compiler applies the fixed scheduling strategy to all possible TLE slicing options (KS, KS&OFM and OFM) to search for the best performance. Table II shows the time of TSO with the fixed TLT strategy. By fixing IS during TF-XLA Code Generation, TSO speedup reaches 28,6% on YOLO. For the YOLO network, IS does not perform well since this network has multiples Conv-layers with IFMs varying

from 102×102 ($H \times L$) to 416×416 – this results in multiples loads of the filters since multiple IN tiles are required. TSO speedup with respect to fixed OS reaches up to 28.6% on ResNet-50. When compared to fixed WS, TSO speedup is 39.8% on InceptionV3. In the case of InceptionV3, when WS is used, multiples W tiles are required to work on the slices, and thus multiples loads of the IFMs become necessary for each W tile, leading to an increase in data transfers. Here again, TSO outperforms the best fixed TLT scheduling strategy.

VII. Conclusion

Given the restricted on-chip memory sizes of NPU architectures, efficient data tiling and scheduling techniques are crucial to minimizing the cost of memory accesses. This paper proposes TSO, an optimization pass for the TF-XLA compiler that identifies the best combination of data tiling, scheduling and MAC operations that minimizes execution of convolutions in CNN models. To achieve that, TSO does a precise modeling of memory burst, achieving a speedup of up to 21.7% for some typical CNN models when comparing to no-burst modeling. TSO also achieves up to 41.0% speedup when compared to a fixed TLE slicing, and 39.8% when compared to a fixed TLT tiling.

Acknowledgment

The authors would like to thank the anonymous reviewers for the insightful comments. This work was supported by LG Electronics and CAPES (Proc. 88887.368695/2019-00).

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[2] 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*, 2016, pp. 770–778.

[3] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *Advances in neural information processing systems*, 2014, pp. 487–495.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[5] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[6] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[7] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun, "Pedestrian detection with unsupervised multi-stage feature learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 3626–3633.

[8] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[9] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[10] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[11] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[12] F. Tu, W. Wu, S. Yin, L. Liu, and S. Wei, "Rana: towards efficient neural acceleration with refresh-optimized embedded dram," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 340–352.

[13] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*. Springer, 2014, pp. 281–290.

[14] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE micro*, vol. 40, no. 3, pp. 20–29, 2020.

[15] 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.

[16] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, "Marvel: A data-centric compiler for dnn operators on spatial accelerators," *arXiv preprint arXiv:2002.07752*, 2020.

[17] S.-C. Kao and T. Krishna, "Gamma: automating the hw mapping of dnn models on accelerators via genetic algorithm," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[18] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 2017.

[19] X. Hu, Y. Zeng, Z. Li, X. Zheng, S. Cai, and X. Xiong, "A resources-efficient configurable accelerator for deep convolutional neural networks," *IEEE Access*, 2019.

[20] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. 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)*, 2018, pp. 578–594.

[21] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 13–19.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[23] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Performance modeling for cnn inference accelerators on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 843–856, 2019.

[24] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[25] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 575–580.

[26] C. Park, S. Park, and C. S. Park, "Roofline-model-based design space exploration for dataflow techniques of cnn accelerators," *IEEE Access*, vol. 8, pp. 172 509–172 523, 2020.

[27] A. Stoutchinin, F. Conti, and L. Benini, "Optimally scheduling cnn convolutions for efficient memory access," *arXiv preprint arXiv:1902.01492*, 2019.

[28] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[29] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[30] S. M. Silva and C. R. Jung, "Real-time brazilian license plate detection and recognition using deep convolutional neural networks," in *2017 30th SIBGRAPI conference on graphics, patterns and images (SIBGRAPI)*. IEEE, 2017, pp. 55–62.