Optimizing Tensor Contractions for Embedded Devices with Racetrack and DRAM Memories

ASIF ALI KHAN and NORMAN A. RINK, Technische Universität Dresden, Germany
FAZAL HAMEED, Institute of Space Technology, Islamabad, Pakistan
JERONIMO CASTRILLON, Technische Universität Dresden, Germany

Tensor contraction is a fundamental operation in many algorithms with a plethora of applications ranging from quantum chemistry over fluid dynamics and image processing to machine learning. The performance of tensor computations critically depends on the efficient utilization of on-chip/off-chip memories. In the context of low-power embedded devices, efficient management of the memory space becomes even more crucial, in order to meet energy constraints. This work aims at investigating strategies for performance- and energy-efficient tensor contractions on embedded systems, using *racetrack memory* (RTM)-based *scratch-pad memory* (SPM) and DRAM-based off-chip memory. Compiler optimizations such as the loop access order and data layout transformations paired with architectural optimizations such as prefetching and preshifting are employed to reduce the shifting overhead in RTMs. Optimizations for off-chip memory such as memory access order, data mapping and the choice of a suitable memory access granularity are employed to reduce the contention in the off-chip memory. Experimental results demonstrate that the proposed optimizations improve the SPM performance and energy consumption by 32% and 73%, respectively, compared to an iso-capacity SRAM. The overall DRAM dynamic energy consumption improvements due to memory optimizations amount to 80%.

CCS Concepts: • **Hardware → Emerging architectures**; • **Computer systems organization → Embedded systems**; • **Software and its engineering →** *Source code generation*;

Additional Key Words and Phrases: Compiler optimization, data transformation, tensors, tensor contraction, matrix multiplication, racetrack memory, preshifting, prefetching, embedded systems, DRAM mapping

ACM Reference format:
Asif Ali Khan, Norman A. Rink, Fazal Hameed, and Jeronimo Castrillon. 2020. Optimizing Tensor Contractions for Embedded Devices with Racetrack and DRAM Memories. *ACM Trans. Embed. Comput. Syst.* 19, 6, Article 44 (September 2020), 26 pages.
https://doi.org/10.1145/3396235

1 INTRODUCTION

Tensors are multi-dimensional data structures that generalize matrices. Consequently, tensor contraction generalizes the operation of matrix multiplication. The abstractions offered by tensors

This work was partially funded by the German Research Council (DFG) through the TraceSymm project CA 1602/4-1 and the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed).
Authors' addresses: A. A. Khan, N. A. Rink, and J. Castrillon, Technische Universität Dresden, 01069, Dresden, Germany; emails: {asif_ali.khan, norman.rink, jeronimo.castrillon}@tu-dresden.de; F. Hameed, Institute of Space Technology, 44000, Islamabad, Pakistan; email: fazal.hameed@ist.edu.pk.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1539-9087/2020/09-ART44 $15.00
https://doi.org/10.1145/3396235

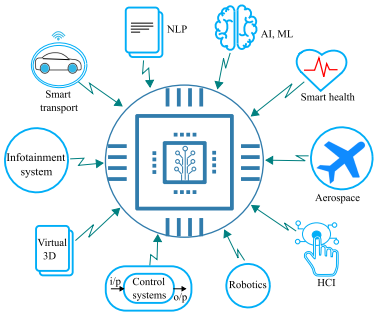

Fig. 1. Applications domains for embedded systems in the Internet of Things.

and their operations are central to many algorithms in modern application domains such as signal and media processing, computer vision, and machine learning. Recent years have seen a surge in the emergence of new programming languages and frameworks specifically designed for the handling of tensor-based computations in these application domains [1, 5, 30, 52], also targeting heterogeneous platforms, e.g., [11] and [29]. In the age of the *Internet of Things*, media processing, computer vision, and machine learning are key application domains for embedded devices, which enable ubiquitous computing in environments that call for extremely low energy footprint and tiny form factors. Examples of such environments are wearables and autonomous vehicles or aircraft, where tensor processing on the device allows for efficient inference in intelligent applications, cf. Figure 1.

The typical constraints on size, power, and energy consumption in the embedded domain make the design of systems for processing large multi-dimensional tensors especially challenging. Particular pressure is put on the design of the memory subsystem, which must accommodate large tensorial data structures within the given constraints. This pushes traditional approaches and technologies to their limits. For example, as was already observed in the mid-2000s, traditional SRAM-based memory is power hungry and suffers from severe leakage power consumption that is responsible for up to 33.7% of the total memory energy consumption [22, 23]. Similarly, the data mapping and the memory access order, if not managed properly, not only degrade performance but also exacerbate energy consumption.

A radically new approach to the design of memory hierarchy is to use a combination of DRAM and NVM memories to exploit their relative benefits while avoiding their disadvantages. One particularly promising NVM technology is the spin-orbitronics-based racetrack memory (RTM), which is more reliable and has lower read/write latency than alternative NVM technologies [6, 43, 44]. Moreover, RTM is very energy-efficient, which is why it is particularly interesting for deployment in embedded devices. This article extends the work in [28] and proposes data layouts and architecture support for optimizing the important tensor contraction operation for RTM-based *scratch-pad* memory (SPM) in conjunction with DRAM-based off-chip memory.

Unlike conventional memories, a single memory cell in RTM stores data in a tape-like magnetic nanowire called *track*. Each track is equipped with a read/write port, and accessing data on a track requires shifting and aligning it to the port position. If the programmer or compiler does not manage data layout judiciously, additional shifts become necessary which degrade performance and energy efficiency. Similarly, to reduce contention in the off-chip DRAM, it is important to devise contention-aware techniques that take into account the underlying DRAM architecture and the tensor data.

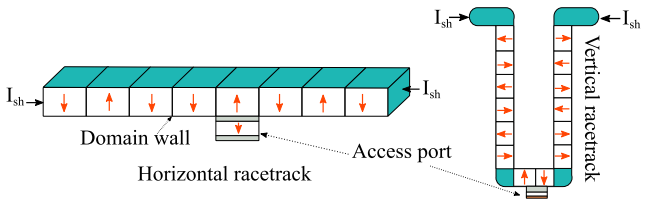

Fig. 2. RTM horizontal and vertical placement.

Specifically, this article makes the following contributions.

(1) For tensors that fit entirely into the SPM, we derive a data layout that reduces the number of shifts necessary for a tensor contraction to the absolute minimum.
(2) We discuss how contractions of large tensors are handled by processing tiles of the tensors in SPM. We show how, in the presence of tiling, the number of shifts can also be reduced to the bare minimum by switching the data layout when bringing new tiles into the SPM.
(3) We present an optimized mapping of tensor data to off-chip DRAM that saves energy by reducing the number of DRAM operations compared to a conventional tensor mapping.
(4) We propose a contention-aware memory access schedule that reduces the impact of different contentions (i.e., read-write interference and row buffer conflict) in DRAM memory by efficiently overlapping computation with memory access that simultaneously improves performance and energy efficiency.
(5) We investigate the impact of large memory access granularity on performance and energy consumption. We found that a large memory access granularity saves energy compared to a smaller one when using our contention-aware memory access schedule and layout.

The rest of this article is organized as follows: Section 2 gives a brief overview of the RTM technology, the SPM layout, off-chip memory, and the tensor contraction operation. Section 3 discusses how various SPM data layouts impact the overall shifting overhead in RTM and presents the best data layout for tensor contraction. Section 4 explains the proposed contention-aware memory layout and efficient scheduler for the off-chip memory. Section 5 and Section 6 provide the evaluation results and comparison with the state-of-the-art. Section 7 discusses the state of the art and Section 8 concludes the article.

2 BACKGROUND

This section briefly explains the working principle and architecture of racetrack and DRAM memories. In addition, it provides background on the tensor contraction operation and layout of scratchpad memories.

2.1 Racetrack Memory

Racetrack memories have evolved significantly over the last decade. Unlike in conventional memories, a single cell in RTM is a magnetic nano-wire (track) that can have up to 100 magnetic *domains* where each domain represents a bit. Domains in a nano-wire are separated by magnetic domain walls (DWs). The track can be placed vertically (3D) or horizontally (2D) on the surface of a silicon wafer as shown in Figure 2. While the vertical placement of tracks achieves the storage density of today's magnetic disk drives, it faces several design challenges. In the horizontal configuration, the

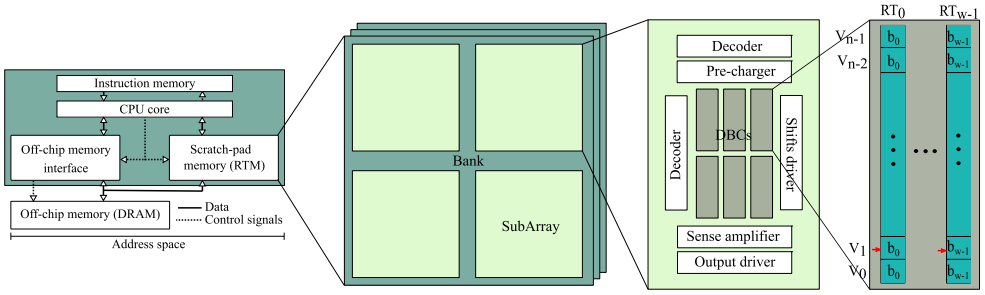

Fig. 3. System and scratch-pad memory architecture.

cell size can be much smaller than the smallest memory cell today. With state-of-the-art materials, the RTM cell size can be 1.5 F^2 compared to 120–200 F^2 in SRAM and 4–8 F^2 in DRAM [37, 53].

2.2 Scratch-Pad Memory

Scratch-pad memory is a faster on-chip memory, usually based on SRAM. Compared to hardware-managed on-chip caches, the SPMs, which are managed by software (i.e., by the programmer or compiler), offer a number of advantages. SPMs have relatively simple architecture and do not require the complex peripheral circuitry of caches; saving both area and energy. SPMs do not need any tag comparison, making access to the on-chip memory faster. Particularly in the embedded domain, SPMs perform better than caches because embedded applications often have regular memory access patterns. With SPMs, it is very easy to efficiently choreograph the data movement between the on-chip and off-chip memories. This also enables better predictability of the application timings, a key feature of embedded systems.

Figure 3 shows a typical embedded system architecture with the address space partitioned between the off-chip memory and the SPM. Typically, the off-chip memory is accessed via cache. However, in this work, we are only interested in the data layout in SPM and the data movement between the off-chip memory and SPM. Therefore, we drop the on-chip cache from our design consideration. We assume that scalar variables can be stored in registers and only focus on the tensor layouts in SPM. SPMs have been successfully used already in the design of accelerators for machine learning, e.g., in [10].

Figure 3 also shows the detailed SPM architecture. Since the typical SRAM-based SPMs have small capacity [10], we consider a comparable 48 KiB SPM which is divided into three banks. Each bank stores one tensor and is made up of 64 *domain wall block clusters* (DBCs). A DBC is a group of w tracks with each track storing n domains. Similar to [56], we assume that each w-bit value is stored in an interleaved fashion across the w tracks of a DBC and that the tracks in DBC can be moved together in a lock-step fashion. For this work, we assume one access port per track (pointing to location 0 initially), *dynamic* port access policy and *lazy* port update policy [6]. Further, we consider w equals 32 and n to be 64. This implies that each bank in the SPM can store a 64×64 tensor. Larger tensors can be partitioned into *tiles*, as explained in Section 3.4.

2.3 Off-Chip Memory (DRAM)

A typical DRAM memory is a hierarchical pyramid of structures composed of channels, banks, rows, columns, and cells. Each (channel, bank, row, column) consists of many (banks, rows, columns, cells), respectively. Each bank contains a *row buffer* that keeps the recently accessed row from the bank. The DRAM controller issues one or many commands to read/write data to/from DRAM memory. A *precharge* (PRE) command is required to make the bank ready for a new

access. This command is not issued in the scenario when the bank is already in the *precharged* state. An *activate* (ACT) command is required to fetch the relevant row to the row buffer of the DRAM bank. A DRAM *row buffer hit* occurs when the requested row resides in the row buffer. In this scenario, the controller does not issue the *activate* command. A DRAM *row buffer conflict* occurs when a request is made to a row Row_i of the bank while another row Row_j resides in the row buffer. Finally, a *read/write* command is issued to access the requested data from the row buffer when the desired row exists in the row buffer. A DRAM row buffer conflict causes high DRAM energy consumption compared to a row buffer hit. This is due to the fact that the DRAM controller issues more commands (precharge, activate, and read/write) to service a request with a row buffer conflict compared to a row buffer hit (i.e., read/write) request.

Read-write interference occurs in DRAM due to read-to-write and write-to-read penalties in terms of latency and energy. Read-to-write latency is the minimum latency between a read request and write request which is incurred to change the mode of the DRAM channel pins from read to write state. Therefore, the DRAM channel will be idle during this time. Write-to-read latency is the latency incurred to change the mode of the DRAM channel from the write to read state plus an additional latency required to correctly update the data in the row buffer. The write-to-read latency is higher compared to read-to-write latency which makes the DRAM channel idle for a longer duration.

2.4 Tensor Contraction

Tensors are multi-dimensional data structures. Special cases of tensors are vectors (1-dimensional tensors) and matrices (2-dimensional tensors). Matrix-vector and matrix-matrix multiplication are low-dimensional instances of the more general operation of tensor contraction. To introduce tensor contractions, let us consider the example of a 5-dimensional tensor A and a 3-dimensional tensor B. Five indices are required to access an entry in A, and the entry at indices i_1, i_2, i_3, i_4, i_5 is denoted as $A_{i_1 i_2 i_3 i_4 i_5}$. Analogously, $B_{i_6 i_7 i_8}$ is an entry in the tensor B, at indices i_6, i_7, i_8. Each index can take values in a fixed integer domain, say $i_\alpha \in \{1, \ldots, M_\alpha\}$ for $\alpha = 1, \ldots, 8$. The M_α are the *dimensions* of the tensors A and B. That is, A has dimensions M_1, M_2, M_3, M_4, M_5, and B has dimensions M_6, M_7, M_8. An example contraction of A and B along two dimensions is the following *sum-of-products* that yields a tensor C,

$$C_{j_1 j_2 j_3 j_4} = \sum_{n=1}^{M_5} \sum_{m=1}^{M_2} A_{j_1 m j_2 j_3 n} \cdot B_{j_4 m n}. \tag{1}$$

Here the contraction is over the dimensions indexed with m and n. For this contraction to make sense, certain dimensions of A and B must match. Specifically, $M_2 = M_7$ and $M_5 = M_8$ must hold. In other words, the pairs of dimensions that are indexed with m and n, respectively, must match. The tensor C that results from the contraction in Equation (1) then is 4-dimensional, with dimensions M_1, M_3, M_4, M_6.

Equation (1) can be rearranged to emphasize that tensor contraction is indeed a generalized version of matrix multiplication. To this end, let $\tilde{A}$, $\tilde{B}$ be tensors that are obtained from A, B by permuting indices as follows:

$$\tilde{A}_{i_1 i_3 i_4 i_2 i_5} = A_{i_1 i_2 i_3 i_4 i_5},$$
$$\tilde{B}_{i_7 i_8 i_6} = B_{i_6 i_7 i_8}.$$

The same tensor C as in Equation (1) is obtained by contracting $\tilde{A}$ and $\tilde{B}$ as follows:

$$C_{j_1 j_2 j_3 j_4} = \sum_{n=1}^{M_5} \sum_{m=1}^{M_2} \tilde{A}_{j_1 j_2 j_3 m n} \cdot \tilde{B}_{m n j_4}. \tag{2}$$

If indices are further arranged into groups k_1, k_3, l such that $k_1 = (j_1\, j_2\, j_3)$, $k_3 = (j_4)$, and $l = (m\, n)$, then C can be written as

$$C_{k_1 k_3} = \sum_{l=1}^{M_2 \cdot M_5} \tilde{A}_{k_1 l} \cdot \tilde{B}_{l k_3}. \tag{3}$$

Equation (3) is readily recognized as matrix multiplication.

Reorganizing the tensor contraction from Equation (1) into the form of matrix multiplication is a standard trick that is commonly referred to as TTGT, e.g., [50]. The key problem with TTGT is that the reorganization of the original tensors A, B into $\tilde{A}$, $\tilde{B}$ requires costly transposition operations, i.e., costly changes of data layout. Moreover, the need for the new tensors $\tilde{A}$, $\tilde{B}$ in TTGT doubles the memory footprint of tensor contraction. In the presence of SPM, the copying of tensors to the SPM is necessary anyway before the contraction operation itself can be carried out. This offers an opportunity for hiding the latency of transposition, provided transfers between off-chip memory and the SPM have uniform latency and can be carried out with a stride.[1]

3 SPM LAYOUT FOR MINIMAL SHIFTING

In this section, we explain the impact that data layout and access order in RTM-based SPM have on the shifting overhead. We move from a naive layout to an optimized layout by successively removing unnecessary shifts that do not do any useful work. To process large tensors in the SPM, they must be broken up into tiles. Switching between tiles generally comes with a latency but also offers further opportunities for reducing the number of shifts by overlapping data transfers and computation, and for latency hiding by prefetching.

3.1 Overview

The operation we implement for SPM is tensor contraction in the form specified by Equation (3). If the dimensions of tensors $\tilde{A}$, $\tilde{B}$ are very small, these tensors can fit entirely in the SPM. We focus on this situation in Sections 3.2 and 3.3, deriving an optimized data layout and access order for a minimal number of shifts.

However, in the relevant application domains of media processing and machine learning, tensors are typically large to begin with. Even if one starts out with moderately sized tensors, after grouping dimensions as in the derivation of Equation (3), the resulting matrices $\tilde{A}_{k_1 l}$, and $\tilde{B}_{l k_3}$ will have large dimensions. To still carry out tensor contraction with a fixed-size SPM, the tensors involved must be *tiled* [39] (or *blocked* [2]).

We assume that the SPM can fit three quadratic $n \times n$-matrices. Then, the tensors $\tilde{A}$, $\tilde{B}$, and C must be divided into tiles of size $n \times n$. To ease the discussion of tiling, we introduce new labels for the dimensions of $\tilde{A}$, $\tilde{B}$, and C in Equation (3):

$$\begin{aligned} &\text{dimensions of } \tilde{A}: && N_1,\ N_2 \\ &\text{dimensions of } \tilde{B}: && N_2,\ N_3 \\ &\text{dimensions of } C: && N_1,\ N_3 \end{aligned}$$

[1] One typically speaks of *gather* and *scatter* accesses to memory when referring to reads or writes with a stride.

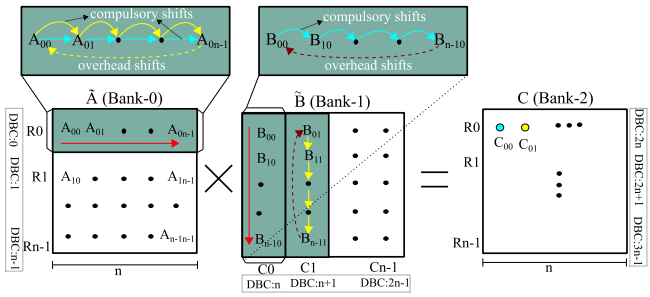

Fig. 4. Tensor contraction with a naive memory layout.

We further assume that n evenly divides these dimensions, i.e., that there are natural numbers T_1, T_2, T_3 such that $N_1 = T_1 \cdot n$, $N_2 = T_2 \cdot n$, and $N_3 = T_3 \cdot n$. If this is not the case initially, one can always pad $\tilde{A}$, $\tilde{B}$, and C with rows or columns of zeros, which does not affect the result of tensor contraction.[2] The tensor C now consists of $T_1 \times T_3$ tiles, $\tilde{A}$ of $T_1 \times T_2$ tiles, and $\tilde{B}$ of $T_2 \times T_3$ tiles, and the tiled version of Equation (3) is:

$$C_{(t_1 \cdot n+k_1)(t_3 \cdot n+k_3)} = \sum_{t=0}^{T_2-1} \sum_{l=1}^{n} \tilde{A}_{(t_1 \cdot n+k_1)(t \cdot n+l)} \cdot \tilde{B}_{(t \cdot n+l)(t_3 \cdot n+k_3)}. \tag{4}$$

For a fixed value of t (in the outer summation), the inner summation (over l) can now be carried out inside the SPM. When the inner summation for fixed t has been completed, new tiles of $\tilde{A}$ and $\tilde{B}$ must be brought into the SPM. Specifically, the tiles for the next value of t, i.e., $t + 1$, are needed. The tile of C stays in the SPM and accumulates the results of the inner summations for each fixed $t = 0, \ldots, (T_2 - 1)$. The tile of C is written back to off-chip memory only after all summations over t and l have been completed. At this point, the evaluation of tensor contraction moves on to the next entry in the rows or columns of tiles of C.

As we will see in Section 3.2, a sizeable portion of the shifts in tensor contraction may be spent on resetting access ports of DBCs to their initial positions for processing again a row of $\tilde{A}$ or a column of $\tilde{B}$ that has previously been traversed in computing an entry of C. While Section 3.3 discusses how the portion of these shifts can be reduced, Section 3.4 demonstrates how unnecessary shifts can be fully eliminated in tiled tensor contraction. Section 3.5 explains that although *prefetching* parts of the next tiles cannot further reduce the number of shifts, it can hide latencies in the full tensor contraction operation. The same statement applies to *preshifting*, cf. Section 3.6.

3.2 Naive SPM Layout

In a naive layout, the tensors $\tilde{A}$, $\tilde{B}$, and C are stored in RTM in their order of access. Specifically, tensor $\tilde{A}$ is accessed row-wise and is stored in the RTM with each DBC storing one row. Similarly, tensor $\tilde{B}$ is accessed column-wise and is stored column-wise in DBCs. The resultant tensor C is computed and stored row-wise. Figure 4 sketches this layout, which is assumed to be the starting point for the tensor contraction operation. All access ports of all DBCs are aligned with the first entries in rows (for $\tilde{A}$ and C) or the first entries in columns (for $\tilde{B}$).

To compute the entry C_{00} in the resultant tensor C, the first row of $\tilde{A}$ (stored in DBC-0) is multiplied with the first column of $\tilde{B}$ (stored in DBC-n). More explicitly, $\tilde{A}_{00}$ is multiplied with $\tilde{B}_{00}$ and both DBCs are shifted once so that the access ports point to next elements $\tilde{A}_{01}$ and $\tilde{B}_{10}$,

[2] This is because contraction is a *linear* operation.

respectively. Next, $\tilde{A}_{01}$ and $\tilde{B}_{10}$ are multiplied and the DBCs are shifted once again. This continues until $\tilde{A}_{0(n-1)}$ and $\tilde{B}_{(n-1)0}$ are reached and multiplied. The blue arrows in Figure 4 demonstrate this process that results in the entry C_{00} of the tensor C, which is marked by a blue dot. At this point, each of DBC-0 and DBC-n have been shifted $n-1$ times, resulting in a total number of $2(n-1)$ shifts. These shifts cannot be avoided as they are required to access the entries in the first row of $\tilde{A}$ and the first column of $\tilde{B}$. Hence, we refer to these shifts as *compulsory shifts*.

The access ports of both DBC-0 and DBC-n now point to locations $n-1$. Before computing C_{01}, DBC-0 needs to be shifted $n-1$ times in order to align its access port to location 0, i.e., to the entry $\tilde{A}_{00}$. These shifts do not perform any useful work, and we call them *overhead shifts*. With these overhead shifts, the total amount of shifts increases to $2(n-1)+(n-1)$. The exact same process is repeated to compute the remaining $n-1$ elements in the first row of tensor C. After computing the last element (C_{0n-1}) in the first row of C, the port position of DBC-0 is restored to position 0. Thus, the total amount of shifts required for computing $R0$ in C is

$$\text{Shifts}'_{R0} = 2n(n-1) + n(n-1), \tag{5}$$

with the second term in the expression on the right-hand side representing the overhead shifts.

After computing the first row of C, the access ports of all DBCs of tensor $\tilde{B}$ point to location $n-1$. They must be shifted back to location 0 before the computation of the next row of C can start. This incurs $n(n-1)$ overhead shifts. The updated sum of the total number of shifts then becomes

$$\text{Shifts}_{R0} = \underbrace{2n(n-1)}_{\text{compulsory shifts}} + \underbrace{n(n-1) + n(n-1)}_{\text{overhead shifts}}. \tag{6}$$

Computing each of the remaining $n-1$ rows of C incurs the same amount of shifts, leading to the total number of shifts required for contracting the $n \times n$ tensors $\tilde{A}$, $\tilde{B}$,

$$\text{Total shifts}' = n \cdot \Big(\underbrace{2n(n-1)}_{\text{compulsory shifts}} + \underbrace{2n(n-1)}_{\text{overhead shifts}} \Big). \tag{7}$$

For writing the entries of C, which result from the computations, $n(n-1)$ compulsory shifts are needed. The same amount of overhead shifts is required to reset the port position to location 0 in all DBCs for tensor C. Adding these to Equation (7) and expanding yields

$$\text{Total shifts (naive)} = \underbrace{2n^3 - n^2 - n}_{\text{compulsory shifts}} + \underbrace{2n^3 - n^2 - n}_{\text{overhead shifts}} \tag{8}$$

From Equation (8), it is clear that half of the total number of shifts are overhead shifts. Thus, avoiding the overhead shifts can improve the memory system's performance by as much as 2×.

3.3 Optimized SPM Layout

The large proportion of overhead shifts in the naive layout of tensors in the RTM occur due to the uni-directional accesses of the tensors' entries: rows of $\tilde{A}$ are always accessed from left-to-right and columns of $\tilde{B}$ from top-to-bottom. In this section, we eventually fully eliminate the overhead shifts by laying out tensors in the RTM so that bi-directional accesses become possible.

First, instead of always accessing $R0$ of $\tilde{A}$ from left to right to compute a new entry in the first row of C, we can access $R0$ in a back and forth manner, and thus completely avoid the overhead shifts for $R0$. Specifically, after computing C_{00}, the access port of DBC-0 is not reset to location 0. Instead, C_{01} is computed by accessing the elements of $R0$ (in $\tilde{A}$) in the reverse order. For this to

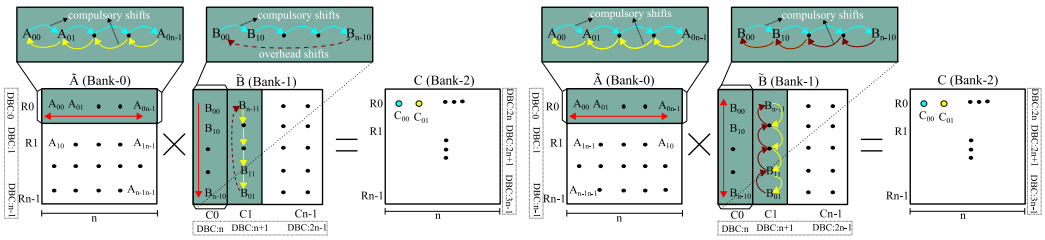

Fig. 5. Tensor contraction with the partially optimized and optimized memory layouts (note the layouts and access orders of $R1$ in $\tilde{A}$ and $C1$ in $\tilde{B}$).

produce the correct result, the column $C1$ of $\tilde{B}$ must be stored in reverse order in DBC-(n+1), as depicted in Figure 5(a). Note that this way of computing C_{01} relies on the associativity of addition.[3]

The same procedure works for the computations of all elements of C, provided the columns of $\tilde{B}$ are stored in DBC-n to DBC-(2n-1) with alternating directions. Since the rows of $\tilde{A}$ are now accessed in a back-and-forth manner, no overhead shifts are incurred for accessing $\tilde{A}$. However, the DBCs that store the columns of $\tilde{B}$ must be fully reset after computing each row of C, leading to a total of $n(n-1)$ overhead shifts per row of C. The numbers of compulsory and overhead shifts required for accesses to C are the same as in the naive layout. Thus, the total number of shifts for the alternating layout of columns of $\tilde{B}$ is

$$\text{Total shifts (partial-opt)} = \underbrace{2n^3 - n^2 - n}_{\text{compulsory shifts}} + \underbrace{n^3 - n}_{\text{overhead shifts}}, \tag{9}$$

which one arrives at by subtracting the $n^2(n-1)$ overhead shifts for resetting the rows of $\tilde{A}$ from the right-hand side of Equation (8).

The vast majority of overhead shifts in the previously discussed alternating column layout of $\tilde{B}$ occurs when the computation of one row of C has been completed and one advances to the next row. At this point, all access ports for the DBCs that store columns of $\tilde{B}$ point to the last entry in each column. To compute the next row of C, the next row of $\tilde{A}$, say $R1$, must be multiplied into the columns of $\tilde{B}$. The access port for DBC-1 points to the first entry in $R1$ of $\tilde{A}$, which necessitates that the access ports for the columns of $\tilde{B}$ (DBC-n to DBC-(2n-1)) be reset to point at the first entry of the columns. However, this resetting of DBC-n to DBC-(2n-1) can be avoided, if the next row of $\tilde{A}$ is stored in reverse order. Then, multiplication of $R1$ into a column of $\tilde{B}$ can be carried out in a backwards fashion. This alternating row layout for $\tilde{A}$ is depicted in Figure 5(b), in combination with the alternating column layout of $\tilde{B}$. The total number of shifts is now comprised of the compulsory shifts and only those $n(n-1)$ overhead shifts that are needed to reset the DBCs for the rows of C after the full contraction operation has been completed, i.e.,

$$\text{Total shifts (opt)} = \underbrace{2n^3 - n^2 - n}_{\text{compulsory shifts}} + \underbrace{n^2 - n}_{\text{overhead shifts}}. \tag{10}$$

Note, in particular, that no overhead shifts are required to reset the DBCs for $\tilde{A}$, $\tilde{B}$ after completing the full tensor contraction. Since the rows of $\tilde{A}$ and the columns of $\tilde{B}$ are traversed in a back and forth manner, the access ports for their DBCs point back to the first entries in the rows of

[3]For floating-point numbers, associativity of addition is typically also assumed when aggressive compiler optimizations are enabled with *fast-math* compiler flags.

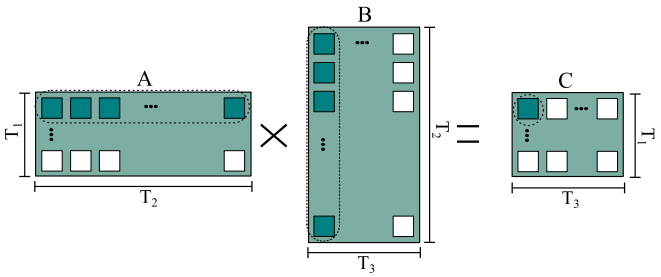

Fig. 6. Tile-wise tensor contractions (tile-size: $n \times n$).

$\tilde{A}$ and columns of $\tilde{B}$, respectively, exactly when the computation of the last entry in C has been completed. This reasoning relies on n being even. In practice, n is actually a power of two, for efficient utilization of address bits.

By comparing Equation (10) with the corresponding equation for the naive layout, i.e., Equation (8), we see that the alternating row and column layout asymptotically cuts the total number of shifts necessary to implement tensor contraction in half.

3.4 Contraction of Large Tensors

We now use the optimized layout from the previous section to optimize the number of shifts needed for contracting large tensors that must be processed in the SPM tile by tile, as explained in Section 3.1. Equation (3) says that each pair of tiles from $\tilde{A}$ and $\tilde{B}$ is contracted exactly as discussed in the previous sections, where it was assumed that $\tilde{A}$ and $\tilde{B}$ fit entirely into the SPM. Equation (3) also says that each tile of C is computed by accumulating the results of contracting a row of tiles of $\tilde{A}$ with a column of tiles of $\tilde{B}$. This is depicted by Figure 6, where T_1, T_2, T_3 are the respective numbers of tiles in each dimension, as in Section 3.1.

Based on Equation (10), the overall number of shifts needed to contract all tiles of $\tilde{A}$ with all tiles of $\tilde{B}$ is

$$\text{Shifts}'_{\text{tiled}} = T_1 T_2 T_3 \cdot \left\{ (2n^3 - n^2 - n) + (n^2 - n) \right\}. \tag{11}$$

This accounts for resetting the access ports of the DBCs that hold a tile of C after the contraction of each pair of tiles of $\tilde{A}, \tilde{B}$. What is not yet accounted for are the number of shifts needed to bring new tiles into the SPM.

To copy a new tile of $\tilde{A}$ or $\tilde{B}$ into the SPM, $n(n-1)$ compulsory shifts are required. The same number of shifts is needed to reset the access ports for the newly copied tile. The computation of each new tile of C must start with a zero-initialized tile. This initialization requires again $n(n-1)$ compulsory shifts and $n(n-1)$ overhead shifts. After the computation of a tile of C has completed, the tile must be copied back to off-chip memory, incurring once again $n(n-1)$ compulsory shifts and $n(n-1)$ overhead shifts. Bearing in mind that the tensor C consists of $T_1 T_3$ tiles, adding all of these shifts to Equation (11) yields

$$\text{Total shifts}_{\text{tiled}} = \begin{cases} T_1 T_2 T_3 \cdot (2n^3 - n^2 - n) \\ +T_1 T_2 T_3 \cdot 2n(n-1) \\ +T_1 T_3 \cdot 2n(n-1) \end{cases} \text{compulsory shifts} \\ \begin{cases} +T_1 T_2 T_3 \cdot (n^2 - n) \\ +T_1 T_2 T_3 \cdot 2n(n-1) \\ +T_1 T_3 \cdot 2n(n-1) \end{cases} \text{overhead shifts}$$

Although the number of overhead shifts only grows quadratically with n, for a fixed n they can still accumulate to a noticeable number. We eliminate them by judiciously laying out tiles that are newly brought into the SPM. Instead of restoring the positions of access ports to location 0 before and after loading/writing each tile, the rows and columns of tiles are loaded and processed in a back-and-forth manner, completely analogous to our discussion in Section 3.3. This completely removes the shifting overhead caused by tiling. Furthermore, the initialization of a tile of C with zeros can take place at the same time as the writing back to off-chip memory of the previously computed tile. Thus, the final total number of shifts required for tiled tensor contraction in the RTM-based SPM is

$$\begin{aligned}\text{Total shifts (opt)}_{\text{tiled}} &= T_1T_2T_3 \cdot \{2n^3 + n^2 - 3n\} \\ &\quad + T_1T_3 \cdot \{n^2 - n\}. \end{aligned} \tag{12}$$

3.5 Hiding Tile-Switch Latency with Prefetching

For large tensors, as soon as the result of contracting the current tiles of $\tilde{A}$ and $\tilde{B}$ has been computed, these tiles need to be replaced, requiring $2n^2$ off-chip reads. In addition, after every T_2 tiles, the contents of the resultant tile of C must also be written back to the off-chip memory, incurring another n^2 off-chip writes. For the access latencies, let us assume that the off-chip access latency, including the data transfer, is t_{off} and both the off-chip memory and the SPM are read/write symmetric. The *tile-switch* latency then becomes

$$\text{Tile-switch latency} = \beta + \begin{cases} 2n^2 \times t_{\text{off}}, & \text{every tile,} \\ 3n^2 \times t_{\text{off}}, & \text{after every } T_2 \text{ tiles,} \end{cases} \tag{13}$$

where β represents the transfer initiation cost. Since the off-chip latency t_{off} is significantly higher than the access latency of the SPM, the tile-switch latency contributes significantly to the total latency and can thus pose a serious performance problem. The value of β and t_{off} are not fixed and depends upon memory layout, memory access schedule, and memory access granularity (cf. Section 4).

To reduce the impact of the off-chip latency on the embedded system's performance, we can use compiler-guided prefetching to overlap the off-chip access latency with the computation latency. Specifically, as soon as the computation of the first row in the resultant tile has been completed, the first row of $\tilde{A}$ can already be replaced with the elements of the new tile. This replacement can happen while the processing unit operates on the next row of $\tilde{A}$. Thus, the load latency of $\tilde{A}$ can be overlapped with the computation latency. Since every element in the resultant tensor requires n scalar multiplications and $n - 1$ additions, computation of the entire row of the resultant tile provides sufficient time for accessing n elements from the off-chip memory.

When the computation of the last row of the resultant tensor C starts, some of the rows in the next tile of $\tilde{A}$ have already been loaded into the SPM. The compiler can then start prefetching the remaining rows of $\tilde{A}$ and the columns of the next tile of $\tilde{B}$. One new column of $\tilde{B}$ can be loaded into the SPM after the computation of each entry in the last row of C. After computing the last entry in the resultant tile of C, the processing unit can immediately start multiplying the first row in the next tile of $\tilde{A}$ with the first column in the next tile of $\tilde{B}$. This way, the significant tile-switch latency can be reduced by overlapping it with computations. In Section 4, we explain how data from the off-chip memory can be efficiently accessed to improve performance and energy.

3.6 Overlapping Shift and Compute Latency with Preshifting

In Section 3.3, we described an optimized memory layout and access order that incurs zero overhead shifts. In Section 3.5, we introduced prefetching to completely hide the tile-switch latency

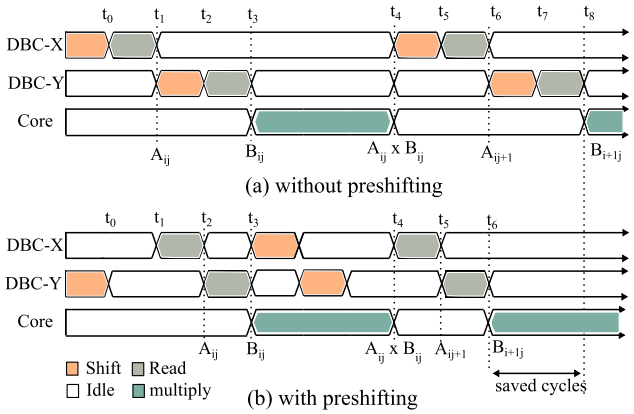

Fig. 7. Overlapping shift latency with computation (DBC X and Y store the elements of $\tilde{A}$ and $\tilde{B}$, respectively).

(for off-chip memory accesses) by overlapping the loading of tiles with the computation process. In this section, we explain how preshifting optimizes the access latency of the on-chip RTM-based SPM.

Typically, SRAM-based SPMs have a fixed access latency of one cycle. Since RTMs are sequential in nature, even with the best memory layout, the DBCs in RTM-based SPM must be shifted once before the next entry can be accessed. This shifting typically takes one cycle, and another cycle is needed to read out the next entry. Hence, the access latency of the RTM-based SPM is 2 cycles.

Fortunately, in the case of tensor contractions, the access pattern is known and the compiler can accurately determine the next memory location to be accessed. We take advantage of this and completely hide the shift latency by *preshifting*, an operation that aligns the access ports of the active DBCs with the memory locations to be accessed next. For instance, when the processing unit is busy multiplying $\tilde{A}_{00}$ with $\tilde{B}_{00}$, both DBCs storing the current row and column are preshifted to point to the next entries, i.e., $\tilde{A}_{01}$ and $\tilde{B}_{10}$. The next memory request made by the program will ask for these entries, and the ports will already be aligned to positions of $\tilde{A}_{01}$ and $\tilde{B}_{10}$ in their respective DBCs. This effectively hides the shift overhead and halves the SPM access latency, as illustrated in Figure 7. Note that this does not interfere with the prefetching operation which affects different DBCs.

3.7 Code Generation for Tensor Contractions

The memory layout and access order that we have identified to reduce the number of shifts in tensor contractions can be automatically generated by a compiler. This includes the appropriate handling of tiling, and even the prefetching and preshifting operations. The major complication in getting a compiler to automatically generate efficient code for tensor contractions is the detection of contractions in the program source code. For programs written in a general-purpose language, this is a non-trivial task: the way in which loop nests and multi-dimensional tensor accesses are structured may obscure the true nature of a tensor operation.

Previous work has suggested methods for detecting matrix multiplication and, more recently, tensor contraction in programs written in general-purpose programming languages. For the Fortran programming language, this is described in [36]. A suggestion for detecting tensor contractions in general-purpose languages has been made in [16], relying on polyhedral methods for the analysis of loop nests [15]. To the best of our knowledge, no assessment exists of how effective

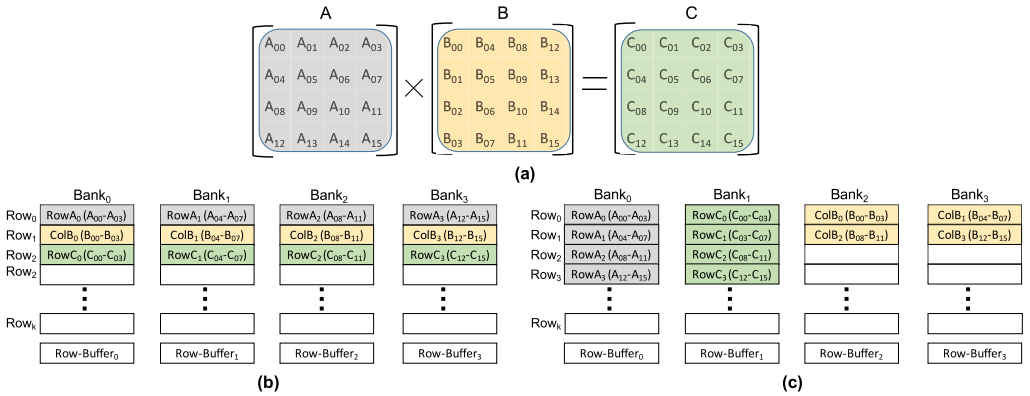

Fig. 8. (a) Illustrative example, (b) naive memory layout, (c) contention-aware memory layout.

the described detection techniques are in detecting contractions in real application domains such as signal and media processing, computer vision, and machine learning.

Domain-specific languages (DSL), on the other hand, offer an alternative approach that makes the nature of domain-specific operations, such as tensor contraction, obvious to the compiler or, more generally, to any code analysis. This is achieved by making tensor contraction a primitive operation of the language, as is the case in virtually all DSLs that are in wide-spread use in the area of machine learning [1, 5, 45]. In the form of MATLAB/Simulink, DSLs are also commonly used in the signal-processing domain. Note that the method for detecting matrix multiplication in [36] is also applicable to MATLAB programs. New DSLs for signal processing [46, 49] have recently been developed, in particular also for embedded applications [32].

In the area of scientific computing, DSLs for tensor operations have been in use for some time, e.g., [4]. Continued interest and recent new developments in this area show that DSLs for tensors are a practically relevant approach to increasing programmer productivity and application performance [30, 47].

4 OFF-CHIP LAYOUT AND ACCESSED ORDER FOR IMPROVED PERFORMANCE AND ENERGY CONSUMPTION

This section describes how an optimized SPM access data from the off-chip DRAM in a performance and energy-friendly way. The proposed optimizations for the off-chip DRAM include: a contention-aware memory layout, an intelligent memory access schedule, and the choice of a suitable memory access granularity. Further, we discuss the impact of these optimizations on the overall energy consumption of the DRAM memory.

4.1 Contention-Aware Memory Layout

To explain our contention-aware memory layout, we consider an illustrative example showing $A \times B = C$ in Figure 8(a). For the given example, we make the following assumptions. Each tensor consists of 16 elements (i.e., A_{00} to A_{15} for tensor A and B_{00} to B_{15} for tensor B) and each tensor fits into its corresponding SPM bank. The off-chip DRAM has 4-banks while each bank has its row-buffer (see Figure 8(b)). For instance, the Row-Buffer$_0$ in Figure 8(b) can accomodate any Row$_i$ of Bank$_0$. We further assume that each DRAM row stores four elements. For instance, Row$_0$ of Bank$_0$ in Figure 8(b) accomodates the first row of tensor A and so on. We start with a naive layout in Figure 8(b) where tensors are mapped to different banks in a row/column interleaved fashion. In

Table 1. Comparing Naive Memory Layout (Figure 8(b)) and Contention-Aware Optimized Memory Layout (Figure 8(c)) for Sequence of Read Commands to Compute the Result of C_{00} Using the (a) Naive Schedule and the (b) Contention-Aware Schedule

Naïve-schedule	Naïve-Mem-layout	CA-Mem-layout
Load A_{00}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load B_{00}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$
Load A_{01}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load B_{01}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$
Load A_{02}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load B_{02}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$
Load A_{03}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load B_{03}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$

(a)

CA-schedule	Naïve-Mem-layout	CA-Mem-layout
Load A_{00}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load A_{01}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load A_{02}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load A_{03}	Read Row_0 of $Bank_0$	Read Row_0 of $Bank_0$
Load B_{00}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$
Load B_{01}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$
Load B_{02}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$
Load B_{03}	Read Row_1 of $Bank_0$	Read Row_0 of $Bank_2$

(b)

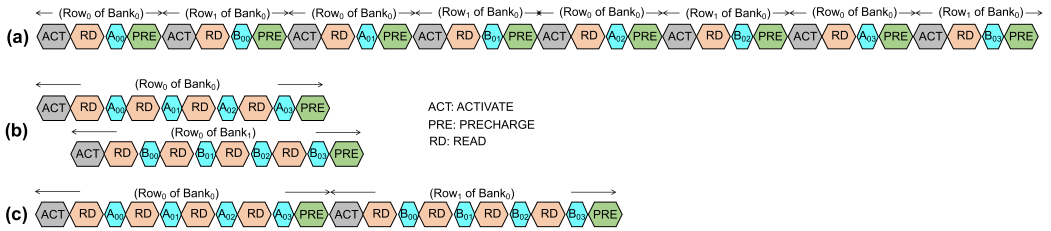

Fig. 9. Sequence of commands for (a) naive memory layout using naive memory access schedule, (b) contention-aware memory layout using naive memory access schedule, and (c) naive memory layout using contention-aware memory access schedule; to compute C_{00}.

this layout, the entries of tensors A, B, and C are stored in row-major, column-major, and row-major order, respectively.

To compute the first entry in the resultant tensor, all elements in the first row of A and first column of B need to be referenced. This computation requires read accesses to physical DRAM rows corresponding to elements A_{00} to A_{03} (i.e., $RowA_0$) and B_{00} to B_{03} (i.e., $ColB_0$) for tensors *A* and *B*, respectively, and write accesses to $RowC_0$ for writing the entry C_{00} of tensor *C*. The second column of Table 1(a) shows the sequence of read accesses for the naive memory layout to compute C_{00}.

The DRAM controller translates each memory request to a series of sub-commands (cf. Section 2.3). For the given read access sequence in the naive layout, the memory controller has to issue the sequence of commands for each request. Figure 9(a) shows the sequence of commands to read elements of tensor A and B for the naive layout. For brevity, the write command to C_{00} is not shown. For this read sequence, a total of eight *ACT-PRE* commands is required due to row buffer conflicts between different rows of the same bank, resulting in high DRAM energy consumption. These row buffer conflicts are highlighted in red in the second column of Table 1(a).

In the naive memory layout, the row buffer conflicts occur due to interference between tensor A and B. This is due to the fact that the data of tensor A and B are mapped to different rows of the same bank. To eliminate inter-tensor interference, our contention-aware optimized memory layout maps the data of tensors A, B, and C to disjoint banks. From the implementation perspective, our mechanism requires the following information from the application layer for each tensor: (1) the bank-id of the first bank B_{start}, and (2) the bank-id of the last bank B_{last}. The above information is communicated to the memory allocation unit which in turn will perform bank assignment in a

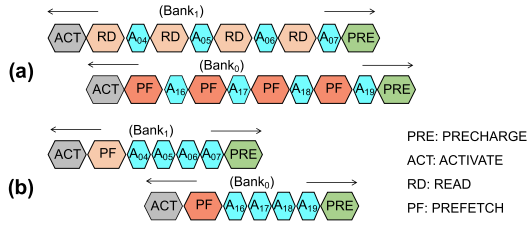

Fig. 10. Interleaving of prefetch and read requests when they belong to different banks using (a) element-level Memory Access Granularity (MAG) and (b) row-level MAG.

round robin fashion among the banks that lies in the range between B_{start} and B_{last}. To enable the application to specify this information, we assume that the system software supports a new variant of the *malloc* system call which includes the above-mentioned two additional parameters. The memory allocation unit in turn will translate virtual pages into physical rows of the off-chip memory.

For the example contention-aware memory layout in Figure 8(c) and the illustrative example in Figure 8(a), the (B_{start}, B_{last}) bank-ids for tensors A, B, and C are ($bank_0$, $bank_0$), ($bank_2$, $bank_3$) and ($bank_1$, $bank_1$), respectively. In this layout, the data of tensor A and C are assigned to $bank_0$ and $bank_1$, respectively. Similarly, for this example, the data of tensor B is assigned to $bank_2$ and $bank_3$ in a round robin manner. The data of tensor B can be assigned to a single bank if it entirely fits into a single bank. In this scenario, the $bank_3$ can be completely turned off for leakage power reduction. However, we do not consider this optimization as we primarily focus on reducing the row buffer conflicts. The third column of Table 1(a) shows the sequence of read accesses for the contention-aware memory layout to compute the results of C_{00}. By eliminating the interference between tensor A and B, the row buffer conflicts in our proposed solution are two in contrast to eight row buffer conflicts in the naive memory layout. Consequently, the contention-aware memory layout results in off-chip DRAM energy saving compared to the naive memory layout. In the results section, we vary these different parameters and tensor sizes and report the impact on performance and energy consumption of the off-chip memory.

4.2 Contention-Aware Memory Access Schedule

This section describes our contention-aware memory access schedule that reduces the number of row buffer conflicts for any memory layout including the naive one. The conflict-aware memory access schedule is shown in the first column of Table 1(b) when computing the result of C_{00}. The proposed schedule reorders the accesses to the elements of tensors A, B, and C in a way that minimizes the row buffer conflicts. In our conflict-aware memory access schedule, the number of *ACT-PRE* commands are curtailed to 2 (Figure 9(c)) compared to 8 when using the naive memory access schedule (Figure 9(a)). Besides, it intelligently interleaves prefetch and read requests to different banks using different granularities (Figure 10) to mitigate the impact of serialization in the contention-aware memory layout (Figure 9(b)).

4.2.1 Loading the First Tile. The major drawback of contention-aware schedule using the naive memory layout is that the computation and memory access cannot be fully overlapped when loading SPM for the first tile. For instance, $RowA_0$ (i.e., A_{00} to A_{03}) and $ColB_0$ (i.e., B_{00} to B_{03}) are serially accessed (i.e., assigned to different rows of the same bank) one after another to compute the result of C_{00} (Figure 9(c)). To reduce the impact of serialization and to efficiently overlap computation with memory fetch for the first tile, our contention-aware memory access schedule makes the following optimizations. First, it fetches all the elements of the first row of tensor A (e.g., $RowA_0$)

followed by loading all columns of tensor B (e.g., $ColB_0$ to $ColB_3$) into SPM. After that, all remaining rows of tensor A (i.e., $RowA_1$ to $RowA_3$) are fetched into SPM. The resulting computation of the elements in the first row of tensor C (i.e., C_{00}, C_{01}, C_{02}, C_{03}) starts immediately with the fetch of the first element in the corresponding rows of tensor B ($ColB_0$, $ColB_1$, $ColB_2$, $ColB_3$), respectively. A row of tensor *C* is scheduled for off-chip memory write after the computation of last entry in that row. The write to a row of tensor *C* is scheduled after loading all elements of tensor A and B in the SPM.

4.2.2 Hiding Off-Chip Latency with Access Overlapping. The contention-aware memory access schedule is based on the following principles to effectively overlap computation with memory access.

(1) To reduce the impact of access serialization, two requests (read or prefetch) are scheduled together if they belong to different banks (cf. Figure 10(a)).
(2) Prefetch requests are scheduled to load the rows of the next tile as discussed in Section 3.5.
(3) Write requests of the current tile that belong to the first half of SPM *C* (e.g., C_{00} to C_{07}) are scheduled before prefech request of the next tile to avoid write starvation.
(4) To reduce the impact of read-write interference (cf. Section 4.2.3), write requests are not scheduled together with read or prefetch requests.
(5) Less-critical write request that belongs to the second half of SPM C (e.g., C_{08} to C_{15}) in the current tile are scheduled after all elements of the next tile in SPM A and B are fetched.

4.2.3 Comparison with FR-FCFS Scheduler. We employ a First-Come-First-Serve (FCFS) access scheduler in the DRAM controller for our contention-aware memory access schedule. The FCFS is simpler to implement compared to state-of-the-art First-Ready-First-Come-First-Serve (FR-FCFS) scheduler [48]. The FR-FCFS prioritizes a request that hits in the row buffer (i.e., the requested row is in the row buffer) over a request that misses in the row buffer. Using a naive memory layout, the drawbacks of the FR-FCFS scheduler compared to our contention-aware memory access schedule are as follows.

First, the FR-FCFS may cause more occurences of read-to-write and write-to-read latency penalties (cf. Section 2.3). The negative impact of these penalties on latency using FR-FCFS scheduler is explained with the following illustrating example. Using the naive memory access schedule, the sequence of commands sent to DRAM controller after reading B_{03} will be write C_{00}, read B_{04}, read B_{05}, read B_{06}, read B_{07}, write C_{01}, read B_{08}, read B_{09}, read B_{10}, read B_{11}, write C_{02}, read B_{12}, read B_{13}, read B_{14}, read B_{15}, write C_{03}. For the above schedule a write-to-read penalty will be incurred between the pairs (C_{00}, B_{04}), (C_{01}, B_{05}), (C_{02}, B_{06}), and (C_{03}, B_{07}). Similarly, a read-to-write penalty will be incurred between pairs (B_{03}, C_{00}), (B_{07}, C_{01}), (B_{11}, C_{02}), (B_{15}, C_{03}). During these times, the DRAM channel will remain idle, thus hurting performance. In our contention-aware memory access schedule, the write requests to off-chip DRAM are only scheduled when the computation of the last element in the relevant row is completed. For instance, the write to C_{00}, C_{01}, C_{02}, and C_{03} are scheduled together after the result of last element (i.e., C_{03}) in $RowC_0$ is available. Furthermore, all write request to a particular row are scheduled together and they are not overlapped with a read or prefetch request. This reduces the number of DRAM channel idle cycles by reducing the occurences of read-to-write and write-to-read penalties. Note that these penalties also exists in contention-aware memory layout when a naive memory access schedule is used. Second, the FR-FCFS may unecessarily prioritize non-critical prefech and write requests (that hit in the row buffer) over critical read request (that miss in the row buffer) which may hurt the performance.

4.2.4 Efficient Selection of the Memory Access Granularity. The *memory access granularity* (MAG) is defined as the number of bytes that can be read from or written to off-chip memory

Table 2. Configuration Details for SRAM, RTM, and Off-Chip Memory

Technology (SRAM and RTM)	32 nm
SPM size	48 KiB
Number of banks	3
Word size	32 bits (4 B)
Off-chip Memory	DDR2-800
Number of RTM ports per track	1
Number of tracks per DBC in RTM	32
Number of domains per track in RTM	64

using a single memory request. The off-chip energy consumption and performance of tensor contraction for the contention-aware memory layout is largely influenced by MAG as explained in the following. We define two types of MAG, namely element-level and row-level MAG. The processing of any MAG type starts with a row-decoding stage, where the entire row is loaded into the row buffer using an activate (ACT) command. In the subsequent column-decoding stage, the item in the row buffer corresponding to the column address is accessed using a read, a write, or a prefetch command. Considering that each DRAM row contains 4 elements, the number of access commands required for element-level and row-level MAG are 4 and 1, respectively.

Since a larger MAG requires fewer access commands, memory consumes less power for a large MAG compared to a smaller one. A larger MAG also reduces the impact of serialization if two requests belongs to different banks. This observation is highlighted in Figure 10 which shows the timing diagram of element-level and row-level MAG requests. For this figure, we assume that the relevant rows of these requests belongs to different banks. The DRAM memory (e.g., DDR2, DDR3, and DDR4 [17] allows multiple MAGs which provides the capability to access multiple data within the same row with automatic address generation. It is worth mentioning that a typical memory controller uses a small MAG because majority of applications suffer from inefficient bandwidth utilization due to limited data reuse. In these applications, not all data of a DRAM row are needed and the unnecessary data is subsequently discarded, which lowers DRAM power efficiency.

5 SPM LAYOUT EVALUATION

This section describes our experimental setup for both on-chip SPM and off-chip DRAM evaluation. Based on this, we compare the performance and energy consumption of the optimized RTM-based SPM with that of the naive and the SRAM-based SPM.

5.1 Experimental Setup

The architectural simulations are carried out in the racetrack memory simulator RTSim [26]. The configuration details for SRAM- and RTM-based SPM are listed in Table 2. Given that access sequences are independent of data, we synthetically generate memory traces for the naive and optimized layouts and fed them to RTSim for the architectural evaluation. The off-chip memory used is JEDEC-compliant DDR2-800 memory 512 MiB composed of a single channel, single rank, eight banks, 64-bit channel and 2 KiB row size.

The latency, energy and area numbers for iso-capacity SRAM and RTM are extracted from Destiny [38] and are provided in Table 3. These values include the latency incurred and the energy consumed by the row/column decoders, sense amplifiers, multiplexers, write drivers,

Table 3. SRAM and RTM Values for a 48 KiB SPM

Memory type	SRAM	RTM
Leakage power [mW]	160.9	25.3
Write energy [pJ]	38.6	35.4
Read energy [pJ]	58.7	22.5
Shift energy [pJ]	0	18.9
Read latency [ns]	1.24	1.01
Write latency [ns]	1.17	1.38
Shift latency [ns]	0	1.11
Area [mm^2]	0.84	0.24

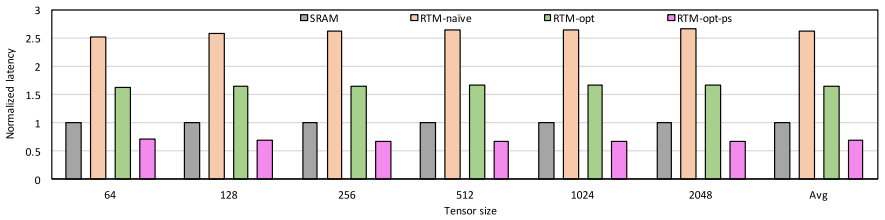

Fig. 11. Latency comparison.

and shift drivers (only for RTM). The DRAM energy consumption results are obtained using the DRAMPower tool [8].

For SPM evaluation, we compare the following configurations:

- *RTM-naive*: The naive RTM-based SPM, cf. Section 3.2.
- *RTM-opt*: The optimized RTM-based SPM, cf. Section 3.3.
- *RTM-opt-preshift* (RTM-opt-ps): RTM-opt with preshifting.
- *SRAM*: Conventional SRAM-based SPM.

We apply prefetching (cf. Section 3.5) on top of all configurations with the assumption of a contention-aware off-chip memory layout (cf. Section 4.1), contention-aware memory access schedule (cf. Section 4.2) and using row level memory access granularity (cf. Section 4.2.4).

5.2 Performance and Energy Evaluation

The main performance and energy consumption results of our evaluation are summarized in Figure 11 and Figure 12, respectively. As depicted, our RTM-opt-preshift improves the average performance by 1.92×, 96% and 32% compared to RTM-naive, RTM-opt, and SRAM, respectively. Likewise, the energy improvement translates to 28%, 10%, and 73%, respectively.

5.2.1 Comparing RTM-naive and RTM-opt. We compare the number of shifts incurred by the naive and the optimized layouts. Our optimized layout (Section 3.3) approximately cuts the number of shifts in half. As a result, the optimized layout reduces the average runtime by 96% and the overall energy consumption by 18% compared to the naive layout. The energy reduction is delivered by simultaneous improvement in both shift and leakage energy (cf. Figure 12). The shift energy gain comes from reducing the number of shifts while the reduction in leakage energy is due to shorter runtime.

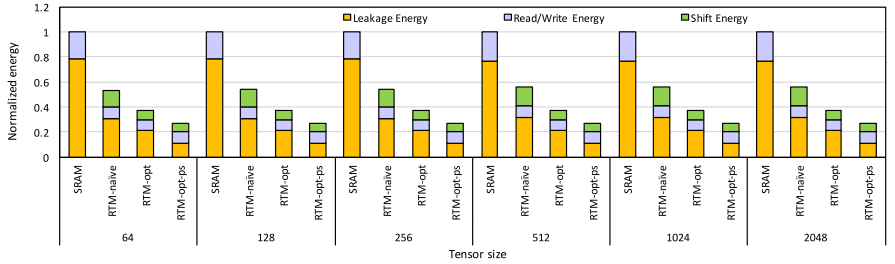

Fig. 12. Overall energy breakdown.

5.2.2 Impact of Preshifting. Although RTM-opt is more efficient in terms of performance and energy consumption compared to RTM-naive, it still suffers from shift-read serialization latency as depicted in Figure 7(a). To completely eliminate this serialization latency, the preshift optimization (Section 3.6) entirely overlaps the shift and the read latency (cf. Figure 7(b)). This improves the average runtime and energy consumption by 96% and 10%, respectively, compared to the RTM-opt configuration. The decrease in the energy consumption comes from the reduced leakage energy which stems from the reduction in runtime.

5.2.3 Comparison with SRAM. The performance comparison with SRAM shows that naively replacing RTM by SRAM for tensor contraction does not provide any benefits in terms of performance, at least for the same capacity. Employing RTM-naive, we witness an average 1.65× runtime degradation compared to SRAM. This runtime degradation is caused by the increased shift cost (cf. Section 5.2.1) and the shift-read serialization latency (cf. Figure 7(a)). Although RTM-opt reduces the shift cost, its average runtime is still 65% worse compared to SRAM. Our combined optimizations (i.e., RTM-opt-preshift), employing the optimized RTM layout and preshifting, reduce the average runtime by 32% compared to SRAM.

The energy results in Figure 12 clearly indicate that each variant of RTM greatly outperforms SRAM in terms of energy consumption. As highlighted, the SRAM leakage energy is the major contributor (i.e., 78%) to the overall energy consumption. The SRAM energy degradation is due to significantly higher leakage power consumed in the larger SRAM cells compared to RTM cells. Finally, since an SRAM cell is significantly larger than an RTM cell, the overall area used by SRAM is 71% larger compared to the iso-capacity RTM, cf. Table 3.

6 OFF-CHIP MEMORY EVALUATION

In this section, we provide a comprehensive evaluation of the off-chip DRAM-based memory in terms of performance and DRAM energy consumption. For evaluation, we compare the following configurations:

- *NaiveSCH-NaiveML-FCFS*: A combination of naive memory access schedule, naive memory layout and First-Come-First-Serve DRAM scheduler.
- *NaiveSCH-NaiveML-FRFCFS*: Similar to above except the DRAM scheduler is First-Ready-First-Come-First-Serve (FR-FCFS) DRAM scheduler [48].
- *CASCH-NaiveML-FCFS*: Our contention-aware memory access schedule (cf. Section 4.2) applied on top of naive memory layout using FCFS DRAM scheduler.
- *CASCH-CAML-FCFS*: Similar to above except the naive memory layout is replaced by contention-aware memory layout (cf. Section 4.1). For this configuration, three banks are dedicated to tensor A and B each, while two banks are assigned to tensor C since we assume an 8-bank DDR2 DRAM memory (cf. Section 5.1).

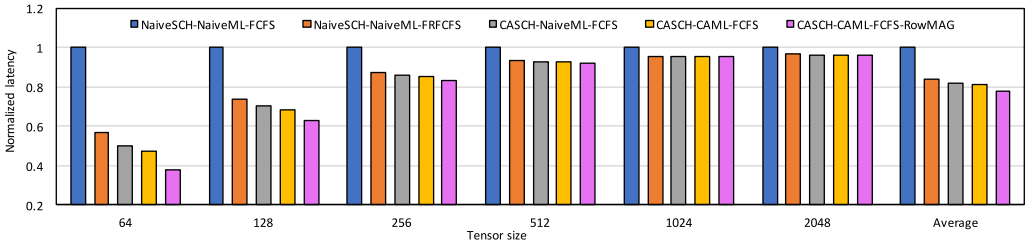

Fig. 13. Latency results for the evaluated configurations.

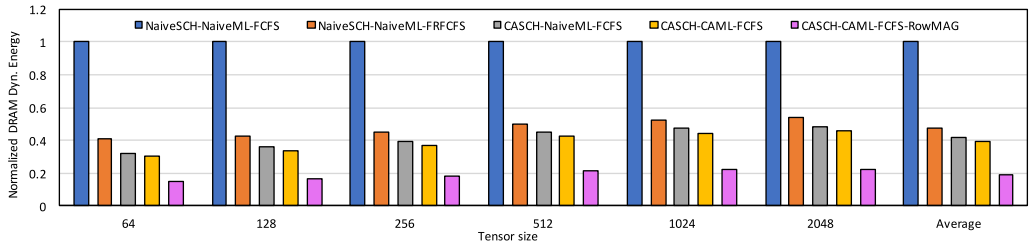

Fig. 14. Off-chip DRAM dynamic energy consumption results.

- *CASCH-CAML-FCFS-RowMAG*: Similar to above except the memory access granularity is equal to the size of DRAM row (i.e., 2 KiB).

Except NaiveML-NaiveSCH-FCFS-RowMAG, the memory access granularity of other configurations is 64-bytes. For all of the above configurations, we assume an RTM-opt-ps configuration which represent optimized RTM-based SPM layout with support of preshifting and prefetching. Figure 13 and Figure 14, respectively, show the latency and DRAM dynamic energy consumption results for all evaluated configurations.

6.1 Impact of Contention-aware Memory Access Schedule

This section qualitatively and quantitatively analyzes a naive (i.e., NaiveSCH-NaiveML-FCFS) and contention-aware (i.e., CASCH-NaiveML-FCFS) memory access schedules applied to a naive memory layout. As depicted, our contention-aware memory access schedule improves the latency by 18.2% and DRAM dynamic energy consumption by 58.6% compared to the naive one. The energy and latency benefits are achieved by reducing the row buffer conflicts while latency reduction is provided through optimizations discussed in Section 4.2.1 and 4.2.2. These optimizations reduces the impact of serialization by effectively overlapping computation with memory access. We also compare our contention-aware memory access schedule with state-of-the-art FR-FCFS DRAM scheduler [48]. Our proposal reduces the impact of read-write interference by 16.7% compared to FR-FCFS scheduler. As a result, our proposal provides improved latency (2.2%) and energy saving (6.2%) compared to FR-FCFS DRAM scheduler.

6.2 Impact of Contention-aware Memory Layout

To show the effectiveness of our contention-aware memory layout (i.e., CASCH-CAML-FCFS) we compare it with a naive memory layout (i.e., CASCH-NaiveML-FCFS). As shown in Figure 13 and Figure 14, the latency and energy behavior of both configurations is almost similar. This implies that the memory access serialization and row buffer conflict problem can be either solved using a contention-aware memory access schedule or a contention-aware memory layout.

As shown in Figure 13, the combination of contention-aware memory layout and access schedule provides noticeable latency reduction for a smaller tensor size compared to the larger one. This is due to the fact that these optimizations reduces the access serialization while loading the SPM for the initial tiles. The access serialization is reduced via request interleaving where the data to disjoint banks are accessed in parallel. For larger tiles, the negative impact of access serialization is primarily offset by prefetching. Therefore, our optimizations does not play a major role in reducing the access latency when the tensor size is large.

6.3 Impact of Memory Access Granularity

This subsection provides an analysis of our joint optimizations by choosing a suitable memory access granularity (MAG) and observing its impact on performance and off-chip DRAM dynamic energy consumption. Figure 13 and Figure 14 shows that the memory requires lower latency and energy when a row-level MAG (i.e., 2 KiB in CASCH-CAML-FCFS-RowMAG) is employed compared to 64-byte MAG (i.e., in CASCH-CAML-FCFS). The energy reduction using row-level MAG is achieved because a larger MAG requires less number of DRAM access commands compared to the smaller one (cf. Figure 10). It is worth mentioning that our access interleaving optimizations in Section 4.2.2 efficiently overlap computations with off-chip memory access for latency reduction. The basic idea is to interleave requests that belong to different banks to be served in parallel.

7 RELATED WORK

This section reviews the relevant literature on tensor and matrix processing, the recent developments in RTM and the state of the art in the utilization of SPM in embedded systems.

7.1 Matrix and Tensor Processing

Matrix multiplication (MM), its applications and optimized implementations have been widely studied for a long time. In numerical linear algebra, MM is a key operation and a major bottleneck in a large class of matrix problems such as the least-square and the eigenvalue problems. By clever algorithm design, the computational complexity of multiplying two $n \times n$-matrices can be reduced from $O(n^3)$ to less than $O(n^{2.376})$ [14, 55]. MM has been implemented on almost all novel and parallel compute platforms [18, 31, 41, 64].

Various linear algebra libraries exist that efficiently implement MM. For instance, the standard *basic linear algebra subprograms* (BLAS) library offers efficient and portable implementations of common operations on matrices and vectors [33]. The *automatically tuned linear algebra software* (ATLAS) library auto-detects the underlying architecture and automatically optimizes algorithms for it [13, 60]. Other work [18, 19] focuses on the partitioning of matrices that best suits the memory hierarchy. All these implementations are optimized for conventional random access memories. The challenges that are introduced by the sequential but energy- and area-efficient RTMs have not been addressed.

The present work even goes one step further: instead of addressing MM in RTMs, we have studied the more general operation of tensor contraction. On conventional platforms, i.e., with traditional random access memory, implementing tensor contraction efficiently has been approached in ways similar to ours [29, 50]. Alternative approaches that avoid transpositions [35] or are based on polyhedral compilation methods [16] have also been explored. It has also recently been demonstrated that, instead of relying on polyhedral methods for the analysis and transformation of loops, meta-programming techniques can be used at least as effectively in optimizing tensor kernels [52], including parallelization for multi-core platforms. Frameworks that attempt to optimize tensor-based computations by auto-tuning, analogous to ATLAS for

computations involving low-dimensional linear algebra, also exist and can target diverse and heterogeneous architectures [11, 54].

7.2 Racetrack Memory

RTMs, being a promising alternative to existing conventional and non-conventional memory technologies, have been explored all across the memory hierarchy including GPU register file [34], lower-cache [62] and last-level-cache levels [56, 63]. In these works, RTM has been reported to be both energy as well as area efficient compared to traditional SRAM and STT-RAM based architectures. Despite being energy and area efficient, RTMs can severely degrade the memory system's performance and energy footprint if the shifting operation is not handled properly. Shifting consumes more than 50% of the RTM energy [63] and can increase the access latency by up-to 26×, in the worst case, compared to the SRAM [57]. Even in our small-size RTM-based SPM, we observed an average 1.33× performance degradation in the naive layout compared to the SRAM.

To mitigate the impact of the shifting overhead, isolated efforts have been made and hardware/software solutions have been proposed. At the architectural front, researchers have proposed techniques such as pre-shifting, data-swapping and re-ordering of the memory requests to minimize the number of shifts [3, 34, 51, 56, 58]. However, these solutions are infeasible in the embedded domain as they require additional hardware that costs area, latency, and energy. Similarly, the software techniques presented in [12, 25, 27], and [40] are not ideal fits to optimize tensors applications. To the best of our knowledge, this is the first work that explores tensors' layout in RTMs for the contraction operation.

7.3 Scratch-Pad and Off-chip Memory

On-chip SPMs have long been used in embedded systems [22, 24]. Compared to caches, SPMs are faster, consume less power and are under the full control of the programmer/compiler. Historically, SRAMs have remained the lone choice of realizing SPMs because of their low access latency. However, with the emergence of NVMs such as STT-RAM [20] and PCM [61], researchers have proposed NVM-based SPMs because they consume less static power and offer higher storage capacity [59]. Nevertheless, these emerging NVMs suffer from higher access latency and endurance.

To minimize the data transfer between the off-chip DRAM and the on-chip SPM, Kandemir et al. [24] first proposed techniques that analyze the application, perform loop and layout transformations and dynamically partition the SPM space in a way that reduces the number of off-chip accesses. To improve the life-time of hybrid SPMs, Hu et al. [21] proposed a dynamic data-allocation algorithm that allocates read-intensive program objects to the PCM-based SPM and write-intensive objects to SRAM. The RTM-based SPMs do not suffer from any of the limitations mentioned above. However, they incur the unique shift operations which, if not handled properly, can severely degrade their performance (cf. Section 7.2). The proposed layout effectively diminishes the amount and impact of RTM shifts in tensor contractions.

Complex and non-linear memory layouts such as recursive layouts [9] and block data layouts [42] for matrix multiplications have been investigated for cache-based architectures to improve the cache hit rate. However, in this article, we explore a simple contention-aware memory layout to reduce the energy consumed in the off-chip DRAM for SPM-based embedded architectures. In the past, several memory access schedulers have been proposed for DRAM. However, in this article, we showed that the widely used FR-FCFS scheduler [48] is not an ideal fit to optimize tensors applications and performs worse compared to our contention-aware scheduler (combined with simple FCFS DRAM scheduler). To the best of our knowledge, this is the first work that explores memory access schedules for joint off-chip DRAM and SPM-based architectures for the tensor operation.

8 CONCLUSIONS

In this article, we present techniques to find optimal tensor layouts in RTM-based SPMs for the tensor contraction operation. We propose an efficient memory access schedule and layouts for DRAM to reduce contention in the off-chip DRAM. We show that the proposed SPM layout reduces the number of RTM shifts to the absolute minimum and the proposed contention-aware memory access schedule and memory layout considerably improve DRAM's performance and energy consumption. To enable contractions of large tensors, we divide them into smaller tiles and employ prefetching to hide the tile-switch latency. We put tile switching to good use by alternating the tiles' layout, which further diminishes the number of shifts. Moreover, to improve the access latency of the on-chip SPM, we employ preshifting that suppresses the shift-read serialization. We employ a contention-aware memory access schedule that largely overlaps tensor computation with memory access. Finally, the use of a suitable memory access granularity on top of our DRAM optimizations further reduces the on-chip SPM access latency and off-chip DRAM energy consumption. Our experimental evaluation demonstrates that the proposed SPM layout, paired with suitable architecture support and DRAM optimizations for off-chip memory, improves the RTM-based SPM's performance by 31%, energy consumption by 73% and area by 71% compared to the SRAM-based SPM. Furthermore, the off-chip DRAM energy savings translates to 80%. The demonstrated benefits substantiate that RTM is a promising alternative to SRAM. In addition, DRAM optimizations are important to achieve performance and energy efficiency, particularly in embedded devices that process large tensorial data structures. We believe that the insights provided by this work can be generalized and integrated into larger hardware-software programming stacks for emerging computing systems [7].

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. http://download.tensorflow.org/paper/whitepaper2015.pdf.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2014. *Compilers: Principles, Techniques, and Tools.* Pearson.

[3] Ehsan Atoofian. 2015. Reducing shift penalty in domain wall memory through register locality. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'15).* IEEE Press, Piscataway, N.J., 177–186. http://dl.acm.org/citation.cfm?id=2830689.2830711.

[4] G. Baumgartner, A. A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, Chi-Chung Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. 2005. Synthesis of high-performance parallel programs for a class of *ab initio* quantum chemistry models. *Proc. IEEE* 93 (2005), 276–292.

[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy).*

[6] R. Bläsing, A. A. Khan, P. C. Filippou, C. Garg, F. Hameed, J. Castrillon, and S. S. P. Parkin. 2020. Magnetic racetrack memory: From physics to the cusp of applications within a decade. *Proc. IEEE* 108, 8 (2020), 1303–1321. DOI: 10.1109/JPROC.2020.2975719

[7] Jeronimo Castrillon, Matthias Lieber, Sascha Klüppelholz, Marcus Völp, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andrés Goens, Sebastian Haas, Dirk Habich, Hermann Härtig, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Akash Kumar, Wolfgang Lehner, Linda Leuschner, Siqi Ling, Steffen Märcker, Christian Menard, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt, and Sascha Wunderlich. 2018. A hardware/software stack for heterogeneous systems. *IEEE Transactions on Multi-Scale Computing Systems* 4, 3 (July 2018), 243–259. DOI: https://doi.org/10.1109/TMSCS.2017.2771750

[8] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. [n.d.]. DRAMPower: Open-source DRAM Power and Energy Estimation Tool. http://www.drampower.info.

[9] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. 2002. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems* 13, 11 (Nov 2002), 1105–1123.

[10] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, 269–284. DOI: https://doi.org/10.1145/2541940.2541967

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen.

[12] Xianzhang Chen, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Chun Jason Xue, Weiwen Jiang, and Yuangang Wang. 2016. Efficient data placement for improving data access performance on domain-wall memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 10 (Oct. 2016), 3094–3104. DOI: https://doi.org/10.1109/TVLSI.2016.2537400

[13] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27 (01 2001), 3–35. DOI: https://doi.org/10.1016/S0167-8191(00)00087-9

[14] D. Coppersmith and S. Winograd. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC'87)*. ACM, New York, 1–6. DOI: https://doi.org/10.1145/28395.28396

[15] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model*. Springer US, Boston, MA, 1581–1592. DOI: https://doi.org/10.1007/978-0-387-09766-4_502

[16] Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (Sept. 2018), 27 pages. DOI: https://doi.org/10.1145/3235029

[17] S. Goossens, T. Kouters, B. Akesson, and K. Goossens. 2012. Memory-map selection for firm real-time SDRAM controllers. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 828–831.

[18] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. DOI: https://doi.org/10.1145/1356052.1356053

[19] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. 2001. A family of high-performance matrix multiplication algorithms. In *Proceedings of the International Conference on Computational Sciences - Part I (ICCS'01)*. Springer-Verlag, Berlin, 51–60. http://dl.acm.org/citation.cfm?id=645455.653765.

[20] F. Hameed, A. A. Khan, and J. Castrillon. 2018. Performance and energy-efficient design of STT-RAM last-level cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 6 (June 2018), 1059–1072. DOI: https://doi.org/10.1109/TVLSI.2018.2804938

[21] J. Hu, C. J. Xue, Q. Zhuge, W. Tseng, and E. H. Sha. 2013. Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 6 (June 2013), 1094–1102. DOI: https://doi.org/10.1109/TVLSI.2012.2202700

[22] M. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. 2004. Banked scratch-pad memory management for reducing leakage energy consumption. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'04)*. IEEE Computer Society, Washington, DC, 120–124. DOI: https://doi.org/10.1109/ICCAD.2004.1382555

[23] M. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. 2005. Compiler-guided leakage optimization for banked scratch-pad memories. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13, 10 (Oct, 2005), 1136–1146. DOI: https://doi.org/10.1109/TVLSI.2005.859478

[24] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*. ACM, New York, 690–695. DOI: https://doi.org/10.1145/378239.379049

[25] Asif Ali Khan, Andrés Goens, Fazal Hameed, and Jeronimo Castrillon. 2020. Generalized data placement strategies for racetrack memories. In *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE) (DATE'20)*. EDA Consortium, 1502–1507.

[26] A. A. Khan, F. Hameed, R. Bläsing, S. Parkin, and J. Castrillon. 2019. RTSim: A cycle-accurate simulator for racetrack memories. *IEEE Computer Architecture Letters* 18, 1 (Jan 2019), 43–46. DOI: https://doi.org/10.1109/LCA.2019.2899306

[27] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart S. P. Parkin, and Jeronimo Castrillon. 2019. Shiftsreduce: Minimizing shifts in racetrack memory 4.0. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–23.

[28] Asif Ali Khan, Norman A. Rink, Fazal Hameed, and Jeronimo Castrillon. 2019. Optimizing tensor contractions for embedded devices with racetrack memory scratch-pads. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*. Association for Computing Machinery, New York, 5–18. DOI: https://doi.org/10.1145/3316482.3326351

[29] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L. Pouchet, A. Rountev, and P. Sadayappan. 2019. A code generator for high-performance tensor contractions on GPUs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, N. J., 85–95.

[30] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. DOI: https://doi.org/10.1145/3133901

[31] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra. 2009. Optimizing matrix multiplication for a short-vector SIMD architecture—CELL processor. *Parallel Comput.* 35 (03 2009), 138–150. DOI: https://doi.org/10.1016/j.parco.2008.12.010

[32] Nikolaos Kyrtatas, Daniele G. Spampinato, and Markus Püschel. 2015. A basic linear algebra compiler for embedded processors. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE'15)*. EDA Consortium, San Jose, CA, 1054–1059. http://dl.acm.org/citation.cfm?id=2757012.2757058.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.* 5, 3 (Sept. 1979), 308–323. DOI: https://doi.org/10.1145/355841.355847

[34] Mengjie Mao, Wujie Wen, Yaojun Zhang, Yiran Chen, and Hai Li. 2017. An energy-efficient GPGPU register file architecture using racetrack memory. *IEEE Trans. Comput.* 66, 9 (2017), 1478–1490.

[35] Devin Matthews. 2016. High-performance tensor contraction without BLAS. *CoRR* abs/1607.00291 (2016). arxiv:1607.00291 http://arxiv.org/abs/1607.00291.

[36] Vijay Menon and Keshav Pingali. 1999. High-level semantic optimization of numerical codes. In *Proceedings of the 13th International Conference on Supercomputing (ICS'99)*. ACM, New York, 434–443. DOI: https://doi.org/10.1145/305138.305230

[37] S. Mittal, J. S. Vetter, and D. Li. 2015. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (June 2015), 1524–1537.

[38] S. Mittal, R. Wang, and J. Vetter. 2017. DESTINY: A comprehensive tool with 3D and multi-level cell memory modeling capability. *Journal of Low Power Electronics and Applications* 7, 3 (2017).

[39] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[40] Joonas Multanen, Asif Ali Khan, Pekka Jääskeläinen, Fazal Hameed, and Jeronimo Castrillon. 2019. SHRIMP: Efficient instruction delivery with domain wall memory. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'19)*. ACM, New York, 1. DOI: https://doi.org/10.1109/ISLPED.2019.8824954

[41] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba. 2006. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *Proceedings of the Conference on High Performance Computing for Computational Science - VECPAR 2006*. 305–318.

[42] N. Park, W. Liu, V. K. Prasanna, and C. S. Raghavendra. 2000. Efficient matrix multiplication using cache conscious data layouts. In *Proceedings of HPCMO User Group Conference*.

[43] S. Parkin, M. Hayashi, and L. Thomas. 2008. Magnetic domain-wall racetrack memory. *Science* 320, 5873 (2008), 190–194.

[44] Stuart Parkin and See-Hun Yang. 2015. Memory on the racetrack. *Nature Nanotechnology* 10, 3 (2015), 195–198.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[46] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (Feb 2005), 232–275. DOI: https://doi.org/10.1109/JPROC.2004.840306

[47] N. A. Rink, I. Huismann, A. Susungi, J. Castrillon, J. Stiller, J. Fröhlich, and C. Tadonki. 2018. CFDlang: High-level code generation for high-order methods in fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. ACM, New York, Article 5, 10 pages. DOI: https://doi.org/10.1145/3183895.3183900

[48] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. 2000. Memory access scheduling. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*. 128–138.

[49] Daniele G. Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO'16)*. ACM, New York, 117–127. DOI: https://doi.org/10.1145/2854038.2854060

[50] Paul Springer and Paolo Bientinesi. 2018. Design of a high-performance GEMM-like tensor-tensor multiplication. *ACM Trans. Math. Softw.* 44, 3, Article 28 (Jan. 2018), 29 pages. DOI: https://doi.org/10.1145/3157733

[51] Z. Sun, Wenqing Wu, and Hai Li. 2013. Cross-layer racetrack memory design for ultra high density and low power consumption. In *Proceedings of the 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[52] Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2018. Meta-programming for cross-domain tensor optimizations. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2018)*. ACM, New York, 79–92. DOI: https://doi.org/10.1145/3278122.3278131

[53] L. Thomas, See-Hun Yang, Kwang-Su Ryu, B. Hughes, C. Rettner, Ding-Shuo Wang, Ching-Hsiang Tsai, Kuei-Hung Shen, and S. S. P. Parkin. 2011. Racetrack memory: A high-performance, low-cost, non-volatile memory based on magnetic domain walls. In *Proceedings of the 2011 International Electron Devices Meeting*. 24.2.1–24.2.4. DOI: https://doi.org/10.1109/IEDM.2011.6131603

[54] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR* abs/1802.04730 (2018). arxiv:1802.04730

[55] Virginia Vassilevska Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, 887–898. DOI: https://doi.org/10.1145/2213977.2214056

[56] Rangharajan Venkatesan, Vivek Kozhikkottu, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. 2012. TapeCache: A high density, energy efficient cache based on domain wall memory. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'12)*. ACM, New York, NY, USA, 185–190. DOI: https://doi.org/10.1145/2333660.2333707

[57] Rangharajan Venkatesan, Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2014. STAG: Spintronic-tape architecture for GPGPU cache hierarchies. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, 253–264.

[58] D. Wang, L. Ma, M. Zhang, J. An, H. Li, and Y. Chen. 2017. Shift-optimized energy-efficient racetrack-based main memory. *Journal of Circuits, Systems and Computers* 27 (09 2017), 1–16. DOI: https://doi.org/10.1142/S0218126618500810

[59] Z. Wang, Z. Gu, M. Yao, and Z. Shao. 2015. Endurance-aware allocation of data variables on NVM-based scratch-pad memory in real-time embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (Oct 2015), 1600–1612. DOI: https://doi.org/10.1109/TCAD.2015.2422846

[60] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC'98)*. IEEE Computer Society, Washington, DC, USA, 1–27.

[61] H.-S. Philip Wong, Simone Raoux, Sangbum Kim, Jiale Liang, John Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth Goodson. 2010. Phase change memory. 98 (12 2010).

[62] H. Xu, Y. Alkabani, R. Melhem, and A. K. Jones. 2016. FusedCache: A naturally inclusive, racetrack memory, dual-level private cache. *IEEE Transactions on Multi-Scale Computing Systems* 2, 2 (April 2016), 69–82. DOI: https://doi.org/10.1109/TMSCS.2016.2536020

[63] Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and W. Zhao. 2015. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *Proceedings of the 20th Asia and South Pacific Design Automation Conference*. 100–105. DOI: https://doi.org/10.1109/ASPDAC.2015.7058988

[64] P. Zhang and Y. Gao. 2015. Matrix multiplication on high-density multi-GPU architectures: Theoretical and experimental investigations. Lecture Notes in Computer Science, vol. 9137, 17–30. DOI: https://doi.org/10.1007/978-3-319-20119-1_2

Received November 2019; revised April 2020; accepted April 2020