Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads

Asif Ali Khan
Technische Universität Dresden
Germany
asif_ali.khan@tu-dresden.de

Norman A. Rink
Technische Universität Dresden
Germany
norman.rink@tu-dresden.de

Fazal Hameed
Technische Universität Dresden
Germany
fazal.hameed@tu-dresden.de

Jeronimo Castrillon
Technische Universität Dresden
Germany
jeronimo.castrillon@tu-dresden.de

Abstract

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 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). 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. Experimental results demonstrate that the proposed optimizations improve the SPM performance and energy consumption by 24% and 74% respectively compared to an iso-capacity SRAM.

CCS Concepts • **Computer systems organization** → **Embedded systems**; *Tensor contractions*; Energy consumption; • **Compilers** → *Data transformation*; Layout transformation; • **Racetrack memory** → *Shifts minimization.*

Keywords Compiler optimization, data transformation, tensors, tensor contraction, matrix multiplication, racetrack memory, preshifting, prefetching, embedded systems

ACM Reference Format:
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 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3316482.3326351

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 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, 6, 26, 51], also targeting heterogeneous platforms, e.g. [8, 19, 25]. 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 [20, 21].

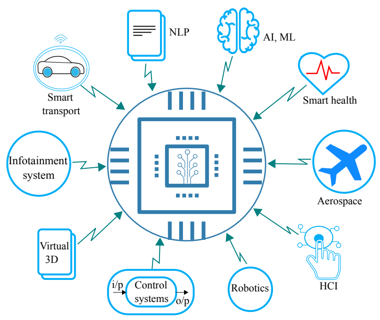

Figure 1. Applications domains for embedded systems in the Internet of Things.

A radically new approach to the design of on-chip memories and the memory hierarchy is offered by *non-volatile memories* (NVM). 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 [43, 44]. Moreover, RTM is very energy-efficient and has ultra-high capacity, which is why it is particularly interesting for deployment in embedded devices that process large tensors.

In this paper we propose and analyze data layouts and architecture support for optimizing the important tensor contraction operation for RTM-based *scratch-pad* memory (SPM). 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. The data layout we propose in this paper asymptotically halves the number of shifts required for tensor contractions. As our analysis shows, this halving of the number of shifts is in fact necessary to give RTM a competitive edge over SRAM-based SPM.

Specifically, this paper 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 brining new tiles into the SPM.
3. Our simulations show that the proposed data layout for tensors in the SPM, paired with suitable architecture support, is required to outperform SRAM in terms of latency. This also reduces the SPM energy consumption by 74%.

We also discuss how languages and compilers can support the generation of efficient code and suitable data layouts for tensor contractions with RTM-based SPM.

The rest of the paper is organised as follows. Section 2 gives a brief overview of the RTM technology, the SPM layout and the tensor contraction operation. Section 3 discusses how various data layouts impact the overall shifting overhead and presents the best data layout for tensor contraction. Section 4 provides a qualitative and quantitative comparison of both the naive and the proposed data layouts with SRAM. Section 5 discusses the state of the art and Section 6 concludes the paper.

2 Background

This section briefly explains the working principle and architecture of racetrack memories. In addition, it provides background on the tensor contraction operation, layout of scratch-pad memories and their placement in embedded systems.

2.1 Racetrack Memory

Racetrack memories have evolved significantly over the last decade. Since their conception in 2008, RTMs have made fundamental breakthroughs in device physics. In RTM version 4.0, several major impediments have been eliminated and improvements in device speed and resilience have been demonstrated [44].

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 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, 52].

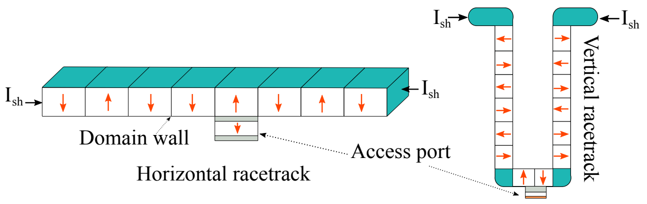

Figure 2. RTM horizontal and vertical placement

The access latency of RTMs depends on how quickly DWs inside a wire can be moved when a shift current is applied. In the RTM 1.0, the maximum DW velocity reported was 100 m s^{-1} [43]. With the development of new structures where a magnetic film is grown on top of a heavy metal, the velocity of DW increased to up to 300 m s^{-1} [36]. However, a major drawback of these designs is that the magnetic film is very sensitive to external magnetic fields. They also exhibit fringing fields, restricting closer packing of DWs in the nano-wire. RTM 4.0 eliminates these impediments by adding an extra magnetic layer on top, which fully compensates the magnetic moment of the bottom layer. Consequently, the magnetic layer does not exhibit fringing fields and is insensitive to external magnetic fields. Moreover, due to the exchange coupling of the two magnetic layers, the DWs velocity can reach up to $1000\,\mathrm{m\,s^{-1}}$ [44, 62].

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 [7].

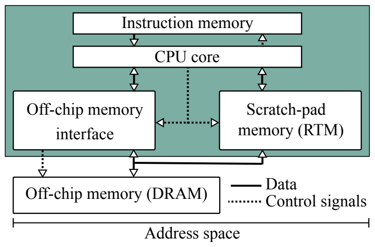

Figure 3. System architecture

Figure 4 shows the detailed SPM architecture. Since the typical SRAM-based SPMs have small capacity [7], 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 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.

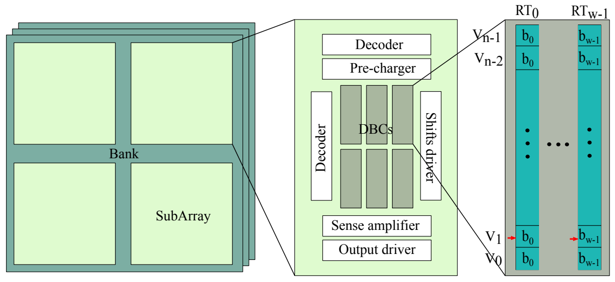

Figure 4. Architecture of the proposed RTM-based SPM

2.3 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 emphasise 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,

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

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

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

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. [49]. 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 Data 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):

dimensions of $\tilde{A}$:	$N_1,\ N_2$
dimensions of $\tilde{B}$:	$N_2,\ N_3$
dimensions of C :	$N_1,\ N_3$

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

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

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 Memory 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

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

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

computed and stored row-wise. Figure 5 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 $\tilde{B}$).

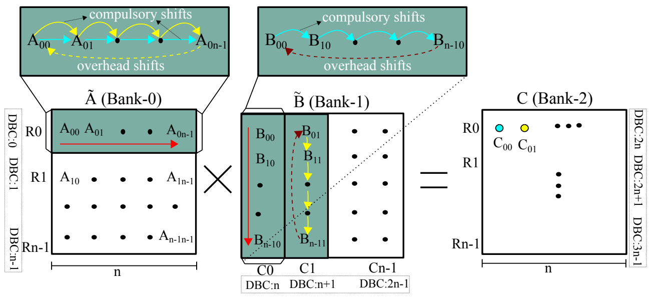

Figure 5. Tensor contraction with a naive memory layout

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 $\tilde{B}_{10}$ 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 5 demonstrate this process that results in the entry C_{00} of the tensor C, which is marked by a blue dot. At this point in time, 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 $\tilde{A}$, $\tilde{B}$,

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

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 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 produce the correct result, the column $C1$ of $\tilde{B}$ must be stored in reverse order in DBC-(n+1), as depicted in Figure 6. 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

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

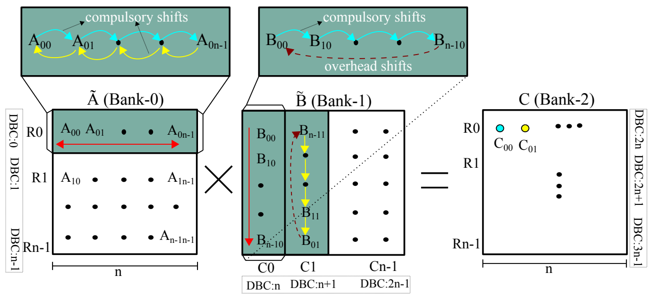

Figure 6. Tensor contraction with partially optimized memory layout (note the layout of $C1$ in $\tilde{B}$ and the access order of $R0$ in $\tilde{A}$)

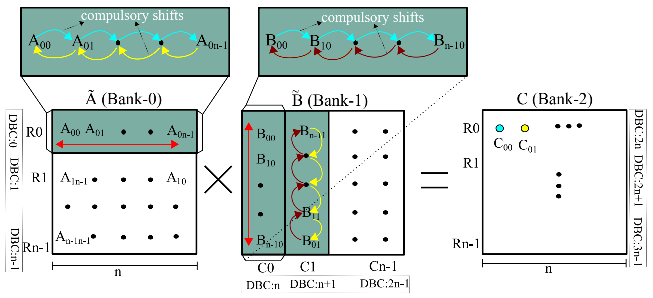

Figure 7. Tensor contraction with the optimized memory layout (note the layout of $R1$ in $\tilde{A}$ and the access order of columns in $\tilde{B}$)

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

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

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 7, 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.

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

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 $\tilde{A}$ 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 8, where T_1, T_2, T_3 are the respective numbers of tiles in each dimension, as in Section 3.1.

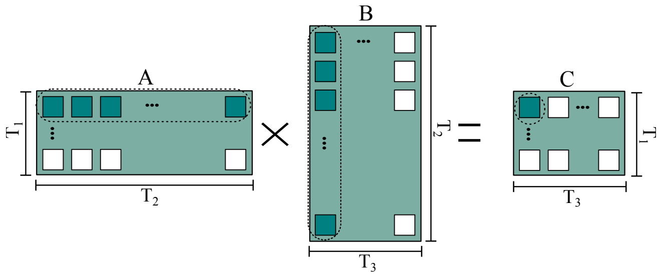

Figure 8. Tile-wise tensor contractions (tile-size: $n \times n$)

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

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

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_1T_3 tiles, adding all of these shifts to Equation (11) yields

$$\text{Total shifts}_{\text{tiled}} = \begin{array}{l} \left.\begin{array}{l} T_1T_2T_3 \cdot (2n^3 - n^2 - n) \\ +T_1T_2T_3 \cdot 2n(n-1) \\ +T_1T_3 \cdot 2n(n-1) \end{array}\right\} \text{compulsory shifts} \\ \left.\begin{array}{l} +T_1T_2T_3 \cdot (n^2 - n) \\ +T_1T_2T_3 \cdot 2n(n-1) \\ +T_1T_3 \cdot 2n(n-1) \end{array}\right\} \text{overhead shifts} \end{array}$$

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 (cf. Tables 1, 2), the tile-switch latency contributes significantly to the total latency and can thus pose a serious performance problem.

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 completely 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 (accessed in burst-mode).

When the computation of the last row of the resultant tensor C starts, the first $n-2$ rows in the next tile of $\tilde{A}$ have already been loaded into the SPM. The compiler can then start prefetching the $(n-1)$-th row 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 $\tilde{B}$, without incurring any latency. At this point, the compiler requests prefetching the last row of $\tilde{A}$ and last column of $\tilde{B}$ for the new tiles. This way, the significant tile-switch latency is fully hidden by overlapping it with computations. Note that the amount of off-chip accesses remains unchanged.

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 (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

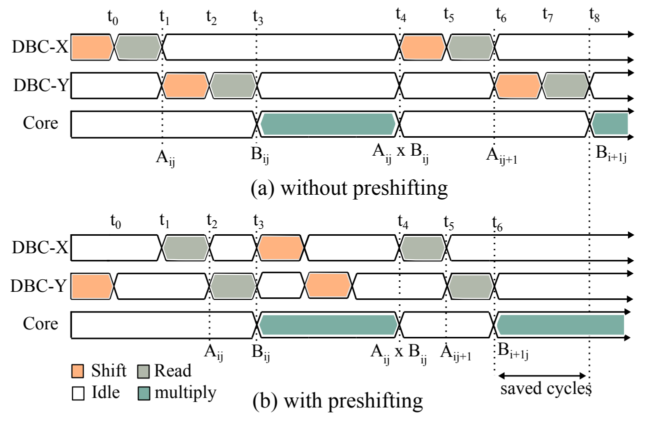

Figure 9. Overlapping DBCs shift latency with computation (DBC X and Y store the elements of $\tilde{A}$ and $\tilde{B}$ respectively)

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 9. 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 [35]. A suggestion for detecting tensor contractions in general-purpose languages has been made in [14], relying on polyhedral methods for the analysis of loop nests [12]. To the best of our knowledge, no assessment exists of how effective 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, 6, 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 [35] is also applicable to MATLAB programs. New DSLs for signal processing [46, 48] have recently been developed, in particular also for embedded applications [30].

In the area of scientific computing, DSLs for tensor operations have been in use for some time, e.g. [5]. 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 [26, 47].

4 Evaluation

This section describes our experimental setup. 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.

4.1 Experimental Setup

The architectural simulations are carried out in the racetrack memory simulator RTSim [24]. The configuration details for SRAM- and RTM-based SPM are listed in Table 1. 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.

Table 1. Configuration details for SRAM and RTM

Technology	32 nm
SPM size	48 KiB
Number of banks	3
Word/bus size	32 bits (4 B)
Transfer inititation cost (β)	30 ns
Off-chip latency	60 ns
Off-chip bus latency	2 ns
Number of RTM ports per track	1
Number of tracks per DBC in RTM	32
Number of domains per track in RTM	64

The latency, energy and area numbers for iso-capacity SRAM and RTM are extracted from Destiny [38] and are provided in Table 2. These values include the latency incurred and the energy consumed by the row/column decoders, sense amplifiers, multiplexers, write drivers, shift drivers (only for RTM).

For 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 on top of all configurations to hide the latency of off-chip accesses as explained in Section 3.5.

Table 2. 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

4.2 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 1.57×, 79% and 24% compared to RTM-naive, RTM-opt and SRAM respectively. Likewise, the energy improvement translates to 23%, 8.2% and 74% respectively.

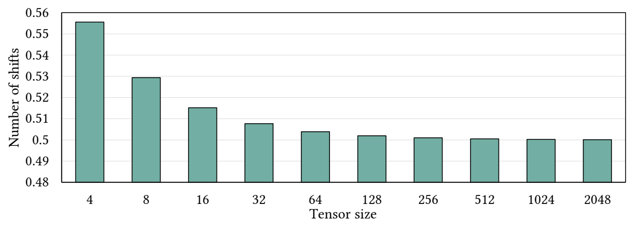

Figure 10. Number of shifts in the optimized layout for different tensor sizes (normalized against naive)

4.2.1 Comparing RTM-naive and RTM-opt

Figure 10 compares the number of shifts incurred by the naive and the optimized layouts. As highlighted, the optimized layout (Section 3.3) approximately cuts the number of shifts in half. Although for smaller tensors, the reduction in shifts is less than 50% and the impact of overhead shifts incurred by tensor C is more evident (cf. Equation (10)); however, this becomes insignificant as the tensors' size increases beyond 128.

As a result, the optimized layout reduces the average runtime by 77% and the overall energy consumption by 15% 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 (cf. Figure 13) comes from reducing the number of shifts while the reduction in leakage energy is due to shorter runtime.

4.2.2 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 9(a). To completely eliminate this serialization latency, the preshift optimization (Section 3.6) entirely overlaps the shift and the read latency (cf. Figure 9b). This improves the average runtime and energy consumption by 79.8% and 8.2% 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.

4.2.3 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 1.33× runtime degradation compared to SRAM. This runtime degradation is caused by the increased shift cost (cf. Figure 10) and the shift-read serialization latency (cf. Figure 9a). Although RTM-opt reduces the shift cost, its average runtime is still 56% worse compared to SRAM. Our combined optimizations (i.e. RTM-opt-preshift), employing the optimized RTM layout and preshifting, reduce the average runtime by 24% compared to SRAM.

Figure 11 shows that the runtime advantage of our combined optimizations is more pronounced in larger tensors. For smaller tensors, the initial tile load latency almost completely offsets the runtime improvement in SPM accesses. In contrast, the impact of initial tile load latency is imperceptible in larger tensors where the average runtime is dominanted by the SPM accesses.

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. 79%) 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. Another interesting observation is that the contribution of the dynamic energy in smaller tensors is not very prominent. Since smaller tensors produce fewer SPM accesses and the relative runtime for smaller tensors is larg, the contribution of dynamic energy to the total energy consumption is small.

To underscore the importance of the dynamic energy consumption, we separate it from the leakage energy in Figure 13. As can be observed, the total dynamic energy of RTM (naive) can get worse compared to SRAM if the shifting overhead

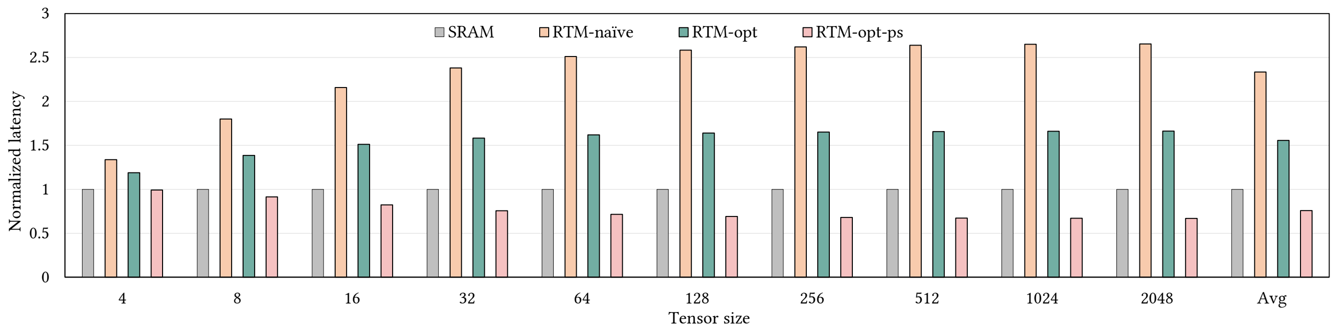

Figure 11. Latency comparison

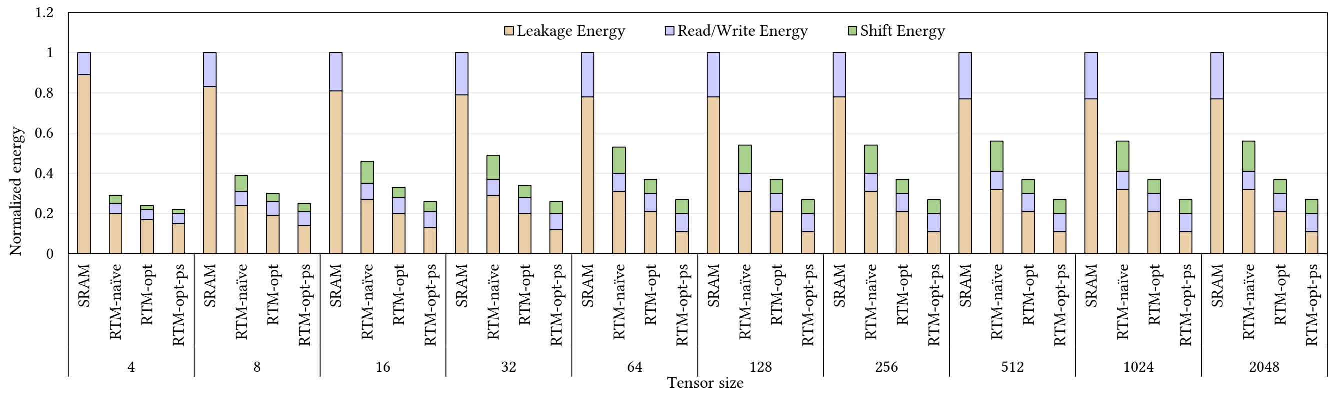

Figure 12. Overall energy breakdown

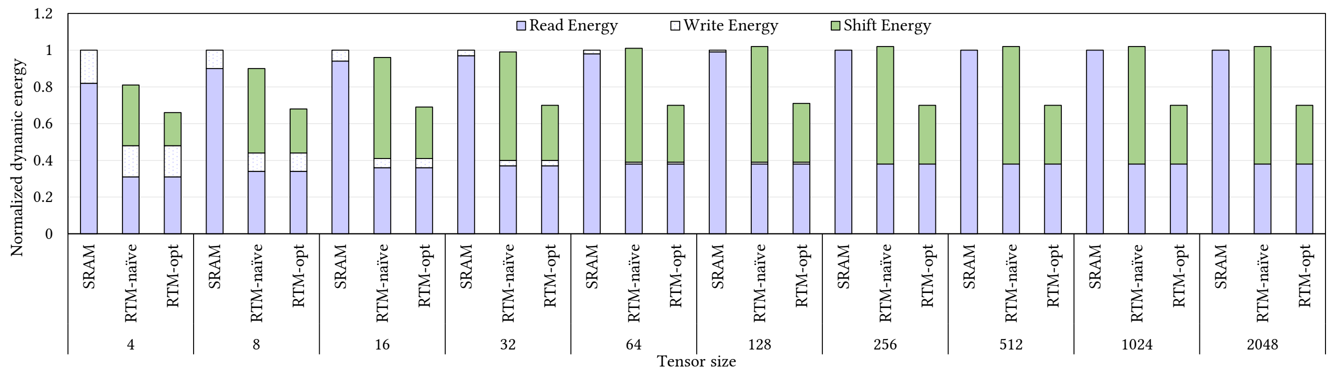

Figure 13. Dynamic energy breakdown

is not handled properly. However, with the combined optimizations in place where each SPM access requires at most one shift, the dynamic energy consumption of RTM reduces by 30.6% compared to SRAM.

The dynamic read energy of SRAM (58.7 pJ) is higher than the combined read plus single shift energy required in RTM (22.5 + 18.9 = 41.4 pJ) for the optimized layout (cf. Table 2). Although the combined write plus single shift energy in RTM (35.4 + 18.9 = 54.3 pJ) is higher compared to SRAM (38.6 pJ) dynamic write energy. However, the RTM write energy does not have a significant impact on the dynamic energy consumption because the tensors contractions are dominated by reads. The number of reads in tensors contractions is approximately $2n$ times higher than the number of writes. As a result, the contribution of the write energy becomes less prominent when the tensor size gets larger, as can be seen in Figure 13.

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 2.

5 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.

5.1 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 $O(n^{2.376})$ [11, 55]. MM has been implemented on almost all novel and parallel compute platforms [15, 29, 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 [31]. The *automatically tuned linear algebra software* (ATLAS) library auto-detects the underlying architecture and automatically optimize algorithms for it [10, 60]. Other work [15, 16] focuses on the partitioning of matrices that best suits the memory hierarchy. For embedded platforms, efficient implementations of MM have been presented on ARMv7 [13], DSP [40] and FPGA [28]. 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 [25, 49]. Alternative approaches that avoid transpositions [34] or are based on polyhedral compilation methods [14] have also been explored. It has also recently been demonstrated that, instead of relying on polyhedral methods for the analysis and transformation of loops, metaprogramming techniques can be used at least as effectively in optimizing tensor kernels [51], 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 [8, 54].

5.2 Racetrack Memory

RTMs, being a promising alternative to existing conventional and non-conventional memory technologies, have been explored all across the memory hierarchy with different optimization objectives. For instance, the RTM-based GPU register file has been reported to be both energy as well as area efficient compared to the traditional SRAM-based register file [33, 58]. On lower cache levels, RTM reduced the energy consumption by 69% compared to an iso-capacity SRAM. When evaluated at last level in the cache hierarchy, RTM reportedly outperformed SRAM and STT-RAM by improving the area, energy and performance efficiency by 6.4x, 1.4x and 25% respectively [50, 56].

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 [56]. 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, 33, 50, 56, 57]. 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 [9, 23, 32] 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.

5.3 Scratch-Pad Memory

On-chip SPMs have long been used in embedded systems [4, 20]. Due to their excellent storage structure, they have also been employed in the accelerators designed for convolutional and deep neural networks [7]. Compared to caches, SPMs are faster, consume less power and are under the full control of the programmer/compiler [22]. 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 [17, 27] 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 issues. To combine the latency benefit of SRAM with the energy benefit of NVMs, NVM-SRAM hybrid SPMs have also been proposed [18].

To make effective utilization of the SPMs and improve their performance, various techniques have been proposed. For instance, the data allocation algorithms presented in [4, 42] judiciously partition the program variables into the on-chip SPM and the off-chip DRAM at compile-time. However, the data allocation is static, i.e., does not change during program execution. The algorithms presented in [53] make dynamic allocation of both stack and global data in the SPM. While all these data allocation techniques were aimed at improving data locality, none of them consider energy and I/O overhead.

To minimize the data transfer between the off-chip DRAM and the on-chip SPM, Kandemir et al. [22] 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. [18] proposed 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. 5.2). The proposed layout effectively diminishes the amount and impact of RTM shifts in tensor contractions.

6 Conclusions

In this paper, we present techniques to find optimal tensor layouts in RTM-based SPMs for the tensor contraction operation. We explain the rationale that led to the derivation of the optimized layout. We show that the proposed layout reduces the number of RTM shifts to the absolute minimum. To enable contractions of large tensors, we divide them into smaller tiles and employ prefetching to hide the tile-switch latency. Moreover, we put tile switching to good use by alternating the tiles' layout, which further diminishes the number of shifts. Finally, to improve the access latency of the on-chip SPM, we employ preshifting that suppresses the shift-read serialization and enables single-cycle SPM access. Our experimental evaluation demonstrates that the proposed layout, paired with suitable architecture support, improves the RTM-based SPM's performance by 24%, energy consumption by 74% and area by 71% compared to the SRAM-based SPM. The demonstrated benefits substantiate that RTM is a promising alternative to SRAM, particularly in embedded devices that process large tensorial data structures and thus enable inference and similar applications.

Acknowledgments

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).

References

[1] Martín Abadi and Ashish Agarwal et al. 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, NJ, USA, 177–186. http://dl.acm.org/citation.cfm?id=2830689.2830711

[4] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627).* 73–78. https://doi.org/10.1145/774789.774805

[5] Gerald Baumgartner, Alexander A. Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. 2005. Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models. *Proc. IEEE* 93 (2005), 276–292.

[6] 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).*

[7] 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, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

[8] 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 *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

[9] 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. https://doi.org/10.1109/TVLSI.2016.2537400

[10] 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. https://doi.org/10.1016/S0167-8191(00)00087-9

[11] D. Coppersmith and S. Winograd. 1987. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87).* ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/28395.28396

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

[13] Kun Feng, Cheng Xu, Wei Wang, ZhiBang Yang, and Zheng Tian. 2012. An Optimized Matrix Multiplication on ARMv 7 Architecture.

[14] 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. https://doi.org/10.1145/3235029

[15] 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. https://doi.org/10.1145/1356052.1356053

[16] 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, Heidelberg, 51–60. http://dl.acm.org/citation.cfm?id=645455.653765

[17] 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. https://doi.org/10.1109/TVLSI.2018.2804938

[18] 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. https://doi.org/10.1109/TVLSI.2012.2202700

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

[20] 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, USA, 120–124. https://doi.org/10.1109/ICCAD.2004.1382555

[21] 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. https://doi.org/10.1109/TVLSI.2005.859478

[22] 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, NY, USA, 690–695. https://doi.org/10.1145/378239.379049

[23] Asif Ali Khan, Fazal Hameed, Robin Blaesing, Stuart Parkin, and Jeronimo Castrillon. 2019. ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0. *arXiv e-prints*, Article arXiv:1903.03597 (Mar 2019). arXiv:cs.ET/1903.03597

[24] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart Parkin, and Jeronimo Castrillon. 2019. RTSim: A Cycle-Accurate Simulator for Racetrack Memories. *IEEE Computer Architecture Letters* 18, 1 (Jan 2019), 43–46. https://doi.org/10.1109/LCA.2019.2899306

[25] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas 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, NJ, USA, 85–95. http://dl.acm.org/citation.cfm?id=3314872.3314885

[26] 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. https://doi.org/10.1145/3133901

[27] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 256–267.

[28] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan. 2009. FPGA Based High Performance Double-Precision Matrix Multiplication. In *2009 22nd International Conference on VLSI Design*. 341–346. https://doi.org/10.1109/VLSI.Design.2009.13

[29] 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. https://doi.org/10.1016/j.parco.2008.12.010

[30] 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, USA, 1054–1059. http://dl.acm.org/citation.cfm?id=2757012.2757058

[31] 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. https://doi.org/10.1145/355841.355847

[32] H. Mao, C. Zhang, G. Sun, and J. Shu. 2015. Exploring data placement in racetrack memory based scratchpad memory. In *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*. 1–5. https://doi.org/10.1109/NVMSA.2015.7304358

[33] 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.

[34] Devin Matthews. 2016. High-Performance Tensor Contraction without BLAS. *CoRR* abs/1607.00291 (2016). arXiv:1607.00291 http://arxiv.org/abs/1607.00291

[35] 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, NY, USA, 434–443. https://doi.org/10.1145/305138.305230

[36] I. Mihai Miron, T. Moore, H. Szambolics, L. Buda-Prejbeanu, S. Auffret, B. Rodmacq, S. Pizzini, J. Vogel, M. Bonfim, A. Schuhl, and G. Gaudin. 2011. Fast Current-induced Domain-wall Motion Controlled by the Rashba Effect. 10 (06 2011), 419–23.

[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] S. Narayanamoorthy, H. A. Moghaddam, Z. Liu, T. Park, and N. S. Kim. 2015. Energy-Efficient Approximate Multiplication for Digital Signal Processing and Classification Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 6 (June 2015), 1180–1184. https://doi.org/10.1109/TVLSI.2014.2333366

[41] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. 2007. Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment. In *High Performance Computing for Computational Science - VECPAR 2006*. 305–318.

[42] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. 1997. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of the 1997 European Conference on Design and Test (EDTC '97)*. IEEE Computer Society, Washington, DC, USA, 7–11. http://dl.acm.org/citation.cfm?id=787260.787762

[43] Stuart Parkin, Masamitsu Hayashi, and Luc Thomas. 2008. Magnetic Domain-Wall Racetrack Memory. 320 (05 2008), 190–194.

[44] Stuart Parkin and See-Hun Yang. 2015. Memory on the Racetrack. 10 (03 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. https://doi.org/10.1109/JPROC.2004.840306

[47] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude 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, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3183895.3183900

[48] 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, NY, USA, 117–127. https://doi.org/10.1145/2854038.2854060

[49] 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. https://doi.org/10.1145/3157733

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

[51] 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, NY, USA, 79–92. https://doi.org/10.1145/3278122.3278131

[52] 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 *2011 International Electron Devices Meeting*. 24.2.1–24.2.4. https://doi.org/10.1109/IEDM.2011.6131603

[53] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. *ACM Trans. Embed. Comput. Syst.* 5, 2 (May 2006), 472–511. https://doi.org/10.1145/1151074.1151085

[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 http://arxiv.org/abs/1802.04730

[55] Virginia Vassilevska Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 887–898. 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. https://doi.org/10.1145/2333660.2333707

[57] Danghui Wang, Lang Ma, Meng Zhang, Jianfeng An, Hai Li, and Yiran Chen. 2017. Shift-Optimized Energy-Efficient Racetrack-Based Main Memory. *Journal of Circuits, Systems and Computers* 27 (09 2017), 1–16. https://doi.org/10.1142/S0218126618500810

[58] Shuo Wang, Yun Liang, Chao Zhang, Xiaolong Xie, Guangyu Sun, Yongpan Liu, Yu Wang, and Xiuhong Li. 2016. Performance-centric register file design for GPUs using racetrack memory. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 25–30. https://doi.org/10.1109/ASPDAC.2016.7427984

[59] Z. Wang, Z. Gu, M. Yao, and Z. Shao. 2015. Endurance-Aware Allocation of Data Variables on NVM-Based Scratchpad Memory in Real-Time Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (Oct 2015), 1600–1612. 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. http://dl.acm.org/citation.cfm?id=509058.509096

[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] See-Hun Yang, Kwang-Su Ryu, and Stuart Parkin. 2015. Domain-wall Velocities of up to 750 m/s Driven by Exchange-coupling Torque in Synthetic Antiferromagnets. 10 (02 2015).

[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 *The 20th Asia and South Pacific Design Automation Conference*. 100–105. https://doi.org/10.1109/ASPDAC.2015.7058988

[64] Peng Zhang and Yuxiang Gao. 2015. Matrix Multiplication on High-Density Multi-GPU Architectures: Theoretical and Experimental Investigations. *Lecture Notes in Computer Science* 9137, 17–30. https://doi.org/10.1007/978-3-319-20119-1_2