Emerging memories as enablers for in-memory layout transformation acceleration and virtualization

Minli Liao
School of Electrical Engineering and Computer Science
The Pennsylvania State University
University Park, Pennsylvania, USA 16802
mjl5868@psu.edu

Jack Sampson
School of Electrical Engineering and Computer Science
The Pennsylvania State University
University Park, Pennsylvania, USA 16802
jms1257@psu.edu

Abstract— Recent works have shown that certain classes of emerging memory technologies lend themselves to organizations that offer equally dense access support for patterns with multiple strides, such as row-column memories. However, with few exceptions, these prior works have only considered such multi-orientation memories (MOMs) and MOM-caching techniques in the context of traditional processor architectures. In this work, we explore the potential for leveraging the capabilities of MOMs to present multiple concurrent views of data organization within the memory hierarchy as a means to offload and overlap inter-kernel marshalling, a range of data layout transformations, and even lazy construction of derivative data structures to work performed by the MOM-capable memories and caches themselves. We demonstrate the potential of MOM-offloading to improve performance and reduce data movement for select computation patterns and describe the application of the approach to broader classes of processing in memory workloads.

I. Introduction

Data structures with logically multidimensional layouts are ubiquitous, from 2D matrices and record tables to N-dimensional abstract graph embeddings. However, for reasons of both history and simplicity, commercial ISAs require the projection of these structures into a single-dimensional address space and the physical structure of traditional memories is heavily optimized to prefer accesses aligned along that same dimension, e.g. with cache line, row buffer, and other locality-optimization effects. However, several new types of emerging memories inherently lend themselves to crosspoint organizations wherein the physical symmetry of the underlying memory array opens up opportunities for equally optimized access in two simultaneous directions. We term these memories *multi-orientation memories* (MOMs), and several recent works have explored both MOMs and how to integrate them with and into caching systems [1], [2], [3] and memories [4], [5].

While prior work on MOMs has exploited scenarios wherein different regions of code expressed different orientation preferences to the same multi-dimensional data structure (e.g. row versus column access across queries to an in-memory database), we propose to do something more radical: We will utilize the separable indexing capabilities of MOMs that allow for different indexing to row and column data within a 2D cache tile to enable the co-location, within a single 2D cache tile, of two distinct software data structures occupying disjoint memory ranges for the special cases where each data structure is a differently realized view of the same data or a subset thereof. While this may seem a narrow restriction, many codes in certain domains do perform explicit transforms. For instance, the reorder function in the Intel DNNL [6] is frequently invoked to transform output orderings between groups of convolutional and other layers in order to allow each collection of kernels to operate in its preferred dimensional ordering. Similar explicit transforms exist in HPC workflows among independently developed workflow portions. Moreover, while software targeting a MOM-enabled platform can simply elide 2D transpose operations and operate in column orientation, higher dimensional transforms will still require some reordering effort, even when mapped to MOMs.

By allowing software to request hardware support for implementing a reorganizing relationship between specific pairs of memory regions, when the system grants such a request (i.e. the specific capacity and organizational restrictions are supported by the cache in question) the reordering kernel can be elided in its entirety and loads and stores to the reordered region can be accessed as soon as all elements of the source data layout are in or have passed through the 2D cache. Our approach will allow, for cache-resident data structures, a potentially zero-copy realization of the reordering kernels that are otherwise needed to perform explicit transforms to realize alternative data views. For non-cache-resident structures, our proposed mechanism will lazily perform the necessary copies on writeback, when a 2D cache tile is evicted, without involving the processor in data reorganization tasks. By restricting the complexity of multi-address indexing to deeper levels of the cache and providing software managed ordering between the accesses to the output data location and its cache-driven transformation, we can keep near-processor memory interfaces simple and only add complexity to the access and fill latencies of less timing-sensitive deeper caches.

To the best of our knowledge, this is the first work to explore utilizing the secondary access dimensions of a MOM to perform address virtualization across producer-consumer pairs where each expects to operate on a logically distinct data structure. While prior works [5] have proposed extensions to main memory for accelerating layout transformations, they have done so by adding new transformation engines to the memory system, rather than by utilizing features naturally supported by MOMs. The contributions of this work include:

- Characterization of opportunities for reordering support via in-memory mechanisms with regard to their suitability for MOM-based acceleration.
- A mixed-orientation multi-address representation for

978-1-7281-4123-7/20/$31.00 © 2020 IEEE

shared producer-consumer buffers that can completely eliminate many intra-cache copy operations.

- An analysis of the bandwidth reduction opportunities possible when replacing explicit software reordering code with MOM-based multi-mapping buffers. We show that our approach can reduce memory traffic by between 20-50%, on average.

II. Background and Related Work

In this section, we provide the background on the abstract MOM model, fundamental hardware considerations for implementing specific classes of MOMs and MOM-caches, and discuss related approaches for providing transform acceleration and fast access to non-contiguous data elements.

A. *Multi-Orientation Memory (MOM) Systems*

Fig. 1 depicts an abstract view of a MOM system. A MOM either implements an ISA-exposed logically multi-dimensional address space, or imposes one atop the underlying linear space by treating particular patterns (e.g. specific strides) as if they were densely co-located. The defining feature of MOMs is main memory and/or caches that are capable of containing data in and serving requests for multiple logical (dense) orientations *at the same time*. In the scope of this paper, we will primarily focus on the specific opportunities made available by *physically* 2D memories (e.g. crosspoint memories built with ReRAM [7] or STT [8]) that may not be available with implementations whose multi-dimensionality is only logical in nature. More specifically, in a logically 2D cache, in addition to the challenges of data duplication across orientations, the presence of a line in a one of the two supported dense orientations does not immediately imply that the same data is currently available in the other dense orientation. We discuss the tradeoffs in supporting an exposed versus an implicit multi-dimensional interface in Section IV.

Orientation Fig. 1 depicts the logical multi-orientation overlay for a linear address space. Different colored squares represent different forms of adjacent data along different dimensions. Unlike a one-dimensional address space where only data of contiguous addresses are adjacent (blue squares), adjacent data in a MOM's address space can have non-contiguous addresses (other colored squares, usually some fixed stride apart). In this paper, we use "row" orientation to refer to the special case of contiguous data with unit stride ("O0" in Fig. 1). In some previously proposed MOMs [2], [1], there is a hardware-dependent stride that defines a fixed preferred second dimension, or the unit of transfer directly corresponds to the resources of a physically 2D tile [1], and we refer to data along this hardware-dependent dimension to be "column" oriented (e.g. the orange squares when the row size of this address space is hardware-dependent, "O1" in Fig. 1).

Hardware and Technology Support for MOMs Regardless of how it is achieved, a MOM system must include a memory that appears to provide data in different orientations with roughly similar cost. Directly using gather-scatter operations on top of traditional DRAM would allow for emulating multi-orientation access, but such a system would exhibit highly asymmetric properties among accesses to different orientations. Texture caches [9], [10] exploit tiled locality, which

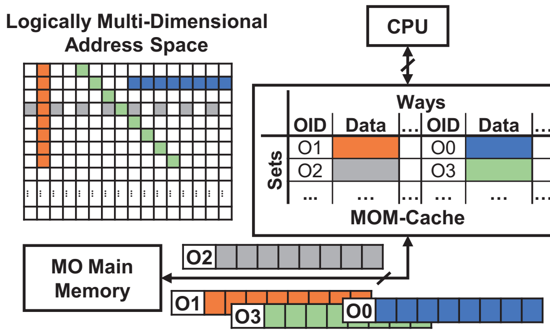

Fig. 1. Abstract view of a MOM system

is multi-dimensional. However, they are generally read-only and have high latencies for all accesses. Recent papers have proposed memories that provide MOM capability directly by data shuffling on DRAMs [3], or leveraging the inherent physical symmetries of some two-terminal emerging memory technologies in crossbar configurations [2], [1] to freely allow reads and writes to data along either row or column orientations. Both approaches have shown substantial performance benefits when applied to MOM-tuned applications.

MOM Caching Recent works [3], [2], [1] that have evaluated architectures with MOMs have also proposed cache hierarchies that can cache oriented data. We refer to all such caches as MOM-caches, regardless of their implementation specifics (e.g. whether in-cache data is only logically or physically contiguous in multiple orientations). While regular SRAM caches using orientation metadata to virtualize multi-orientation support have been more broadly proposed [3], [2], [1], this work relies on physically multi-dimensional caches [1] in order to realize transformations between software-defined source-destination pairs of orientation preference. Compared with prior physically-2D caching, we relax the constraint that the vertical ("column") stride be fixed as a memory configuration parameter and thereby enable dynamic formation of tiles wherein row-locality in the source dimension is maintained while simultaneously constructing column locality in the desired output ordering.

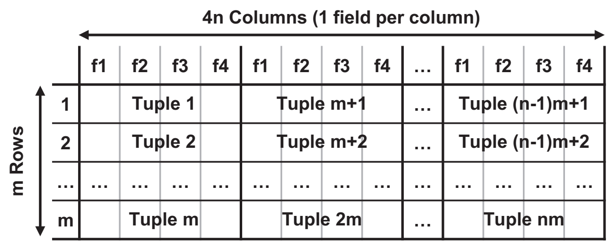
4n Columns (1 field per column)
m Rows
f1 f2 f3 f4 f1 f2 f3 f4 ... f1 f2 f3 f4
1 Tuple 1 Tuple m+1 ... Tuple (n-1)m+1
2 Tuple 2 Tuple m+2 ... Tuple (n-1)m+2
... ...
m Tuple m Tuple 2m ... Tuple nm

Fig. 2. MOM data layout of an IMDB table

B. *Applications with Strong Affinities for MOMs*

MOMs are particularly appealing for applications with highly structured multi-dimensional data that does not have a single dominant access orientation. For example, in-memory database (IMDB) applications have access patterns that vary by query, the same matrix may be accessed as the left matrix in one computation and the right matrix in the next, etc. Fig. 2 shows an example of a MOM-aligned data layout proposed in Wang, et al. [2], where tuples for a mixed OLTP/OLAP in-memory database are stored consecutively in row orientation, and fields are stored consecutively in column orientation. MOMs directly address the needs of such IMDBs by enabling both row-dominant (OLTP) and column-dominant (OLAP) accesses to the same table, as well as allowing column-oriented reads to specific fields in comparison-based filtering queries to exist alongside row-oriented readouts of matching data from the same tables. Our proposed approach goes further in also accelerating some view realization operations on such tables to produce derivative table organizations.

In similar fashion, MOMs can enable more direct implementations of linear algebra operations to achieve the efficiencies of more complex code transformations by allowing equally dense column and row oriented accesses to MOM-aligned data. As has been previously noted [3], a broad range of applications, including key-value stores, graph processing, and graphics, can potentially derive benefits from MOMs and MOM-caching by exploiting simultaneously dense accesses to both multiple fields within a memory object and the same field across many memory objects beyond the obvious case of IMDBs. However, no widely used compiler infrastructures or codebases currently target MOM models or provide memory allocators designed to align with the logical-physical mapping requirements of proposed rigid row-column MOMs or otherwise pack data in multiple simultaneous access localities.

C. Related Work

Symmetric and Transpose Memories In addition to two-terminal technologies such as STTRAM [11], [12], PCM [13], [14] and ReRAM [7] that, in crosspoint topologies, can perform read and write operations via either of the vertical or horizontal crosspoint wire sets [8], the possibility of symmetric access has also been explored in other technologies and array structures. For example, FeFET [15] memories have been proposed with symmetric access capabilities [16], as have multi-layer SRAMs built with monolithic 3D integration [17], and symmetric access DRAMs [18] and SRAMs [19].

Access Pattern Locality The most direct benefit of MOMs is that they can provide spatial locality in multiple simultaneous access stride orientations. Many previous works aim to improve performance by increasing data locality through other means. Various compiler optimizations, such as layout optimizations [20], [21], [22], [23], [24], loop reordering and tiling [25], [26], [27], [28], etc. improve data locality by co-aligning data layout and access patterns in a cache-favorable fashion. Recent works have provided hardware support for more diverse data locality by allowing caching of non-unit-stride data within a single cache line [3], [2], [1], and have demonstrated considerable performance benefits when combined with proper compiler optimization. However, no previous work has attempted to use MOM-caching to map multiple distinct address ranges into the same physical cache tile.

III. Motivation and Characterization

A. Typical transformation in applications

Many applications with multi-dimensional data structures have dense access patterns corresponding to particular data layouts and the cost of performing a data layout transformation can be preferable to performing sparse accesses. Typical transformations include matrix transpose, dimension reordering on high dimensional matrices, joining/splitting data structures, etc. For example, in scientific applications with multiple computation kernels, each computation kernel has a preferred data layout to get maximum data locality or reuse and accelerate the computation, and input data to these computation kernels must be transformed to the preferred layout if it is not already ordered in alignment with that kernel. Database applications also exhibit data transformation, where different views are realized across content from one or more tables.

In this work, we focus on Intel's DNNL ***reorder*** primitive as a case study. The reorder primitive transforms the order of dimensions of a multi-dimensional matrix from a producer order to a consumer order. In DNNL, computation kernels have been optimized with respect to specific source dimension orders, and require input data to these kernels to be layed out as such. For a network with multiple layers where a previous layer's output is fed to the next layer (e.g. as in deep neural networks), if the produced data from the previous layer is in a different layout than the next layer's requirement, a reorder is invoked. We choose to focus on the DNNL reorder primitive because the explicit reordering can be easily isolated, and the kernels represent various forms of reordering and tiling in high dimensional data that provide broad coverage of real-world transformation examples.

B. Overhead of transformation

While performing transformations in general improves the performance by accelerating the computation that consumes the data being transformed, it is not without its costs. For instance, matrix transpose prior to performing matrix multiplication is generally seen as cost-effective, as the cost of the transpose is amortized beneath the more asymptotically expensive matrix multiplication. However, the same argument would not be present for matrix addition, as the costs of transposition and addition would be similar.

To characterize the performance overhead of layout transformations, we profile the performance of 11 examples from DNNL. Fig. 3 shows the fraction of time spent in reorder kernels as opposed to computation kernels for these examples. On average (geomean), 10% of execution time is spent as reordering overheads. This shows that there is an opportunity for performance improvement with acceleration to reorder kernels. Since layout transformation only moves data around without modification or performing calculation, we can naturally relieve the processor of such a task by offloading it to the memory system and, further, perform the transform with memory-level, rather than processor-level parallelism. More specifically, in this work, we propose performing the layout transformation with physically 2D last-level caches, where, for two of the three offload models considered, the transformation overhead can be elided.

C. Hardware overhead to support reordering in a 2D cache

There are several challenges to perform reordering in the memory system. First, the memory system has to calculate the address mapping of pre-reorder (source) addresses to post-reorder (destination) addresses. Second, since the memory

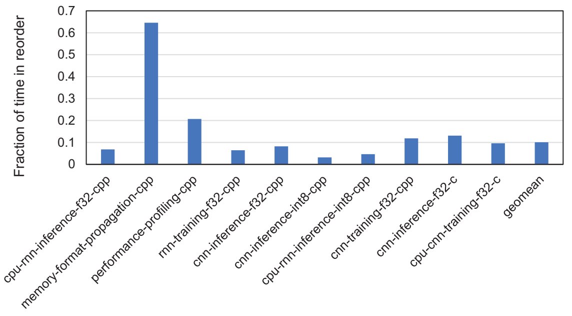

Fig. 3. Fraction of time spent in reorder among kernel sequences using DNNL

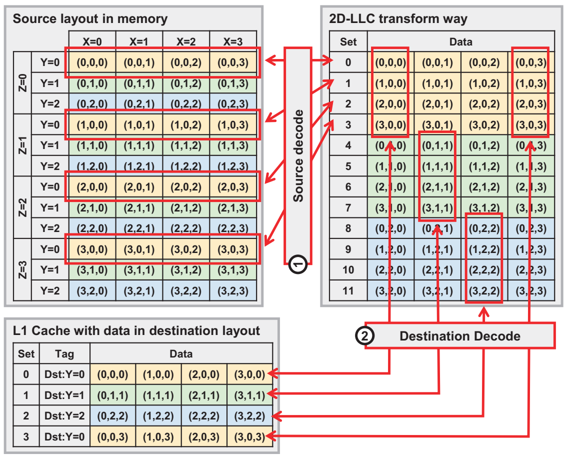

Fig. 4. Illustration of transform with 2D LLC

system operates on data of cache line granularity, it requires support for an efficient gather operation if data in destination cache lines comes from multiple source cache lines. To address the first challenge, some address calculation unit will be required to perform a physical to physical translation. However, 2D memories already inherently support multiple indexing functions. To address the second challenge, we propose using physically 2D caches. We leverage this multi-orientation property so that the data is written to the substrate in one orientation with cache lines of data in the pre-reorder layout, and data is read out from the substrate in the other orientation with cache lines of data in the post-reorder layout.

Fig. 4 illustrates an example of the proposed data layout transform with physically 2D caches. In this example, a 3D matrix with dimensions X, Y and Z, is being reordered from ZYX (X is the dense dimension) to YXZ (Z is the dense dimension), and the cacheline size is 4 elements of this matrix. The elements are always denoted as (Z,Y,X) regardless of ordering. As illustrated, the source layout in the memory has dense data on the X dimension, and each cacheline transferred from the memory to LLC will have 4 consecutive elements in the X dimension. Under a normal decode for LLC, the cacheline with elements (0,0,*) maps to the set 0, (0,1,*) to set 1, ..., (3,2,*) to set 12. To gather the Z dimension, we first use the modified address calculation unit to change the cache decode of source address (①). In this example, instead of using the ZY as index, we use YZ as index, which puts (1,0,*) in set 1 instead of set 4, and (0,1,*) in set 4 instead of set 1. With the column read capability of a 2D substrate, we can now read a column cacheline out of the cache, and the previous source decode has prepared the data so that the column cachelines has consecutive elements in the Z dimension. To provide data in YXZ order, we read the data out of the 2D cache in column, and assign the corresponding YXZ destination address (②). The destination address is generalized as a starting address for the destination (Dst) plus YXZ, denoted as Dst:YXZ. The upper level caches of the system is agnostic of the 2D cache transform, and keep simple address decoding. The Dst:YXZ will be decoded as a normal cacheline with Z as the block offset, X as the set index, and Dst:Y for tag.

Based on the parameters of reordering, different hardware support is required. For example, if the address calculation uses only numbers that are powers of 2, it can be achieved with simple bit mapping, while non-power of 2 numbers will require adder and/or multiplier operations. We classify the transform cases into 5 categories based on the hardware overhead: 1) power of 2 cacheline stride, 2) power of 2 gather stride, 3) non-power of 2 cacheline stride, 4) non-power of 2 gather stride, 5) others.

For the first category, consecutive data in a cacheline stays the same during reorder, hence no gather is required, and only hardware to implement the virtualized address calculation is required. On top of that, the stride of source address between consecutive cachelines in the destination layout is power of 2 multiple of cacheline size. For example, the dimensions of the source is 4x2 cachelines, and the destination is 2x4 cachelines, and we denote the source cacheline addresses as 0, 1, ..., 7, then the correponding source address in the destination layout order is 0, 2, 4, 6, 1, 3, 5, 7, which can be calculated by simply swapping the second and the third bit of the cacheline address. For the second category, while the stride between source addresses in destination layout is still a power of 2 multiple of cacheline size, the consecutive data in a cacheline in destination layout comes from data in multiple cachelines in the source layout, and a gather is required. In this work, the gather is done naturally with the careful data placement in the 2D memory substrate, hence the additional hardware overhead beyond that needed for a physically 2D cache is small. For the third category, the reorder granularity is larger than or equal to the cacheline size as in the first category, except that the stride between source addresses is not a power of 2 number. This category requires an additional multiplier and adder to calculate the destination address. For the fourth category, the reorder granularity is smaller than cacheline size, and the stride between source addresses is a multiple of cacheline size but not a power of 2 number. This category can still benefit from the physically 2D cache gather from category 2 since elements from different source cachelines are still aligned, but requires the multiplier and adder from category 3 to calculate the addresses.

The fifth category includes other cases, where the stride on source address in destination layout is not a multiple of cacheline size. This means that even with the careful placement of source cachelines in 2D cache, the consecutive elements of the destination cacheline cannot naturally fall in the same column cacheline, and data from multiple row or column cachelines are required piece up an entire cachline with consecutive data in destination layout. Additional hardware

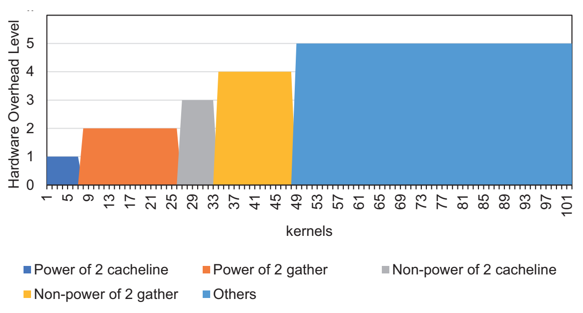

Fig. 5. Distribution of hardware overhead across kernels

such as buffers, shifters, and even scratch pads are required to perform reordering, and we consider such approaches too expensive to consider for the scope of this paper.

Fig. 5 shows the classification of reorder kernels from DNNL based on the hardware overhead level. It is shown that about 7% of the kernels falls under the 1st category, 19% under the 2nd, 7% under the third, 15% under the fourth, and 52% under the fifth. In this work, we only focus on the reorder kernels from the first 4 categories, which covers about half of the reorder kernels.

IV. Transform-capable Cache Architectures

We consider three distinct ways in which a physically-2D MOM cache can be used to perform orientation transformations. In all cases, we consider MOM support only in the last level cache, but discuss different degrees of accessibility, ordering, and expression of the transform to be applied between software and hardware structures.

Sequentialized access with inaccessible intermediate 2D tiles as transform engines (Simple-Explicit) In the most restricted model, software would specify two memory regions and the transformation between them, reserving a fixed quantity of 2D cache tiles to use as transformation engines for that particular pair. In this model, the transformation occurs explicitly, via software invocation, and cannot begin until the entire source region has been generated. Each 2D cache tile would be extended with an address generation unit and then 1) load in the relevant rows from the traditional portions of the cache memory system (row-wise gather), 2) when all rows within a given tile have been populated, write the data from column-aligned cache blocks into row-oriented cache blocks in their destination region via single-cache-block transfers into the traditional portions of the last-level cache. In this model, there is no potential for zero copy, and thus the only direct benefits of the 2D cache memory are 1) cheaper implementation of the gather operation at the 2D tile level than a traditional gather or scratchpad solution, and the associated non-involvement of the processor in the data reorganization, similar to previously proposed transform accelerators [4], [5], 2) because no fine-grained software access ever occurs directly to elements of the 2D tiles, there are limited concerns about latency or hardware cost in implementing more complex address calculations (i.e. generating addresses over a known range of multidimensional offsets, even for non-power-of-2 strides, is substantially simpler than performing the reverse operation) and 3) the 2D tiles can also be allocated as a set of non-transforming 2D cache blocks for more traditional caching without substantial hardware overhead. While the tile-level transform operations can be parallelized with respect to each other, accesses to the destination region cannot safely proceed until all transforms have occurred.

The other two models we consider trade off either hardware or interface complexity to enable *implicit* rather than explicit transformations. Both models have a different software interface requirement than the aforementioned explicit transform model, in that, rather than having to wait for the transformation to complete before accessing the destination memory region, the only requirement is that all cache lines in the source region have been flushed to the last-level cache before accesses to the new region begin. Both approaches rely on augmenting 2D tiles with dual addressing scheme support such that reads from the destination memory region, filtered by range at the cache controller, will naturally access columns of data written in row-orientation into to the source region, likewise selected for non-standard inter-row stride by range comparison at the cache controller.

Derived multi-dimensional addresses (Complex-Implicit) Physically 2D memories can provide their most unique benefits, i.e. zero copy capabilities, when they allow co-locating two reorderings of the same data structure that logically exist in different memory ranges within the same physical cache block. For power-of-2 inter-cacheline strides, supporting such simultaneous access capabilities is straightforward and represents minimal costs over previously proposed row-column memory arrays in terms of configurable bit selection within addresses. However, supporting non-power-of-two inter-cacheline strides (See 3rd and 4th bars in Figure 5) could require prohibitively expensive address calculation mechanisms (e.g. integer division/modulo operators) that would be impractical for many implementation scenarios. That said, with some degree of HW-SW co-design, it may not be necessary to support arbitrary non-power-of-2 strides in order to achieve reasonable coverage of inter-kernel transformations. For instance, if the non-power-of-2 behavior is related to the use of small (e.g. 3x3) kernels, then the supporting a limited set of specific, small, non-power-of-2 indexing components may be viable for some platforms.

Exposed multi-dimensional address interface (Simple-Implicit) If we can change the way that addresses are expressed between the processor and caches and among cache layers, for addresses within the specified transformational regions, then dual addressing for arbitrary non-power-of-2 strides can be performed with substantially reduced hardware costs, relative to the previous approach. Rather than having to derive the dimensional addresses from a traditional memory address, if the data request is transmitted as a base and a vector of indices, generating the relevant cache fields requires only multiplication and addition, for non-power-of-2 strides, and only configurable bit selection for power-of-2 strides. While explicit acknowledgement of the multidimensional nature of data layout is, in some sense, the most natural way of expressing an access to a MOM, it is also the least backwards compatible. This approach would require ISA extensions and alterations of the inter-cache fill request representation in order to be able to support both traditional (linear) memory space requests and MOM-specific base+index-vector memory requests. In practice, we would expect that ISA support would only extend to a finite number of explicitly specified memory dimensions (to limit the size of a given request to something comparable to existing linear addresses), and that any additional logical dimensions would have to be consolidated with respect to the generated accesses (e.g. A[I][J][K][L] represented as A[IJ][K][L] in an ISA supporting a maximum of 3 explicit dimensions).

TABLE I
EASE OF HARDWARE SUPPORT FOR DIFFERENT EXECUTION MODEL / REORGANIZATION COMPLEXITY PAIRINGS

	Simple-Explicit	Complex-Implicit	Simple-Implicit
Po2 cache-line	Bit remapping	Bit remapping	Bit remapping
Po2 gather	Address gen. MAC	Bit remapping	Bit remapping
Non-Po2 cache-line	Address gen. MAC	Integer Div.	Address gen. MAC
Non-Po2 gather	Address gen. MAC	Integer Div.	Address gen. MAC

Table I provides an overview of the tradeoffs among each of the three in-cache transformation models. Depending on the relative importance of workload coverage, address generation costs, backwards compatibility, and data movement reduction, each of the three models is a plausible alternative for specific, restricted domains.

A. Software interface

Whether using an implicit or explict transformation, there are several key software requirements to employ a transform-capable cache. Since the transforms are applied to physical addresses, both the source and destination region must be allocated within the same large-page or segment-based allocation. Software must register these regions in advance via a system-level request interface, similar to those proposed for other region-based semantic filtering approaches [29]. Such requests may be rejected dynamically based upon competing requests for resources from other programs or threads, lack of support within the particular cache for the selected strides or data sizes, or other constraint violations. It is assumed that, when a registration request is rejected, a software fallback will be used instead. However, more complex schemes exposing the particular reasons for rejecting a registration of a source-destination-transform tuple could be employed that allow for adaptive selection of subregions to be transformed in order to match the capacity or occupancy challenges of a particular system at runtime.

For the Simple-Explicit execution model, transform-barrier APIs must be used to separate the writing of the source format, the transformation, and operations on the destination format. The flushing of any source region lines from the L1 and L2 caches will be performed implicitly when the transformation is explicitly invoked. For both implicit models, writing of the source region need only be separated from operations on the destination via a flush-source API call. In all three models, an alternative to flushing is to disable caching of lines in the source region except at the last level cache. Note that lines in the destination region could remain cacehable in both cases. We assume that both APIs will be simultaneously supported, but that only one would be in use at a time for a given source-destination pair, depending on the read-write or write-only nature of the generation of data within the source region.

TABLE II
SIMULATED CACHE CONFIGURATION

Block Size	64B
L1	32KB, 4-way associative, traditional 1D SRAM
L2	256KB, 8-way associative, traditional 1D SRAM
L3	1MB/core, 8-way associative, 2D 128KB per transform way

While supporting 2D transform memory directly at the L1 would remove the need for any cache flushing concerns, the technologies that currently offer physically 2D cache blocks are poor matches for both the latency and endurance demands of an L1 cache, even before the additional complexity of multi-indexing is considered.

V. METHODOLOGY

A. Benchmarks

We use the Intel DNNL [6] reorder primitive kernels to evaluate the effectiveness of 2D-LLC reorder. These reorder kernels perform dimension reordering of 4D and 5D data (including batch and channel dimensions), including transformation into blocked format (similar to tiling). Each invocation of reorder consists of a series of smaller reordering kernel applications, wherein each of these smaller kernels operates on data sizes less than 128KB, which can be mapped to a single way in the 2D-LLC during reorder transform. The evaluated JIT-generated reorder kernels are numbered with ordering from low to high hardware cost as described in section III.

B. Cache Simulation

We use an in-house, trace-driven cache simulator for modeling the memory transfers performed in a 3-level cache system with a traditional 1D SRAM private L1 and L2 (agnostic of 2D properties) and a physically 2D L3 (as the reorder transformation engine) to evaluate the effectiveness in transfer reduction of the proposed scheme. Traces extracted from the execution of DNNL reorder operations are used to drive the cache simulation. We extend an initial model of column access based on the 2D physical cache tiles in George, et al. [1] with additional multi-indexing and transform support mechanisms. Each way supports only 1 column gather granularity, and only 1 way of the 2D L3 is used for reorder for any given kernel. For a 64 byte cacheline, based on element sizes, there are 5 possible granularities: 64x1 byte, 32x2 bytes, 16x4 bytes, 8x8 bytes, 4x16 bytes, 2x32 bytes. In this work we use an 8-way 2D L3 that can support all of them. We simulate address decode capability for non-power-of-2 strides for both source and destination regions, which is cost-practical under the first and third execution models and covers roughly half of the kernels examined (category 1 through 4). Table II shows the parameters of the simulated cache hierarchy.

VI. EVALUATION

We evaluate the memory transfer reduction of using 2D-LLC as a reorder engine against using a CPU as the reorder engine. The source data is assumed to reside in the main memory, hence when CPU is used as the reorder engine, data

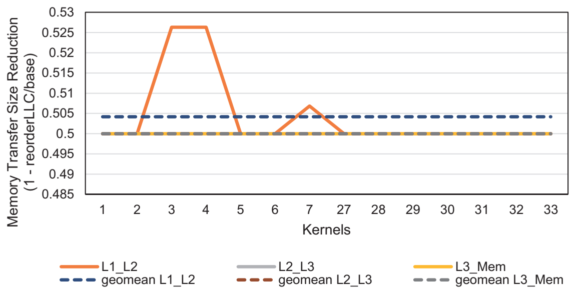

Fig. 6. Transfer size reduction with 2D LLC as reorder engine

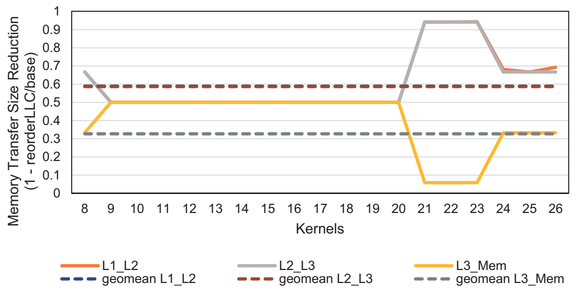

Fig. 7. Transfer size reduction with 2D LLC as reorder engine

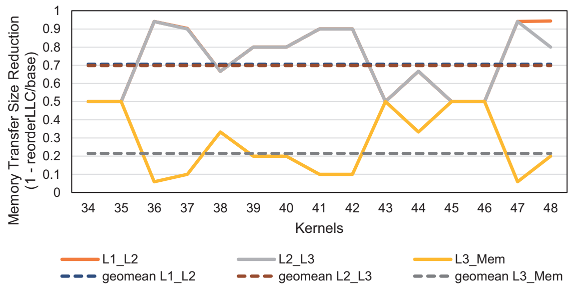

Fig. 8. Transfer size reduction with 2D LLC as reorder engine

will have to be loaded from the memory to the CPU, gathered in CPU (via vector register operations), and then the CPU would write the destination data to L1. For fair comparison, the transfer size for 2D-LLC as a reorder engine also includes the traffic of gathered destination data from 2D-LLC to L1.

For each of the kernels evaluated, we show the transfer size reduction between L3 and main memory, L2 and L3, and L1 and L2. Since 2D-LLC as a transform engine does not incur data traffic between L1 and CPU, the transfer reduction on this link is always 100%, and we do not show this data in the result graphs since it is always 1.

Fig. 6 shows the transfer size reduction on kernels with hardware overhead from category 1 and 3. While different in hardware cost, these kernels have the same behavior in the sense that they all read cachelines from a source region and write cachelines with the same data to a different position in the destination region. Since all loaded source data has been written to the destination region, the total size of transfer for source and destination regions is the same.

We can see that for these kernels, the transfer size reduction is always near 0.5. With CPU reordering, the partial writes from registers to L1 require the destination cachelines to be loaded to L1 first, and this requires both source and destination data to be read from memory to L1. For data transfer between L1-L2, and L2-L3, the 2D-LLC transform does not require source data to be transferred, and this reduces the transfer size by the source data size. For transfer between L3-memory, the 0.5 reduction comes from 2D-LLC providing entire cachelines of data at once, which removes the requirement of loading an explicit copy of the destination region from memory.

For kernels number 3, 4 and 7, the transfer reduction between L1 and L2 increases slightly over 0.5 due to L1 writebacks when performing CPU transforms. Without source region data in the L1, 2D-LLC is free from L1 caching problems as long as the destination region is cache resident.

Fig. 7 shows the transfer reduction result of kernels with hardware overhead from category 2. It can be observed that there is more variation in the transfer size reduction compared to the previous case. This is because data loaded from the source region are gathered to form cachelines in the destination region, and the evaluated kernels do not always perform the reorder for the entire destination (split destination region for parallelization). Hence the size of the source data may be larger then that of the destination data, and for kernels 8, 24, 25 and 26, each cacheline in the destination requires data from 2 cachelines in the source (32B from each), which results in 1/3 traffic reduction in the L3-memory transfer, and 2/3 traffic reduction in the L1-L2 and L2-L3 traffic. Kernels 21, 22, and 23 gather data from 16 cachelines to destination, hence the traffic reduction is 1/17 and 16/17 for L3-memory and L1-L2, L2-L3 respectively. Note that, compared to multiple CPU cores running such kernels for parallelism, the 2D-LLC would reduce memory transfer to each of the cores at the same time. On average, the L1-L2 traffic and L2-L3 traffic are reduced by about 0.59, and the L3-memory traffic is reduced by about 0.33.

Fig. 8 shows the transfer size reduction for kernels with hardware overhead from category 4. These kernels shows more varying situations, with different source region size used to gather into destination region cachelines. In general, when the use of source region data is sparse, the L1-L2 and L2-L3 traffic reduction is high, and the L3-memory reduction is low. On average, the L1-L2 and L2-L3 transfer size is reduced by about 0.7, and the L3-memory transfer size is reduced by about 0.2.

VII. Conclusion

In this paper, we have described approaches for innately supporting transform operations through modest extensions of previously proposed 2D cache tile address generation. Our proposed changes do not require redesign of the fundamental 2D cache tiles themselves, allowing their continued use as MOM caches when not performing dimensional reordering, view realization, or other data layout transformation tasks. We characterize the coverage across observed classes of transform that will be possible with different levels of hardware investment in address calculation and different associated possibilities for performing zero-copy transforms by virtue of simultaneously addressing two distinct physical memory ranges within a single 2D cache tile. Using these zero-copy capabilities, we demonstrate that up to 50% of memory transfer operations can be elided by moving to an in-cache 2D transform approach for a range of reordering operations performed in the Intel DNNL [6] library that have compatible capacity and stride constraints.

References

[1] S. George, M. J. Liao, H. Jiang, J. Kotra, M. Kandemir, J. Sampson, and V. Narayanan, "Mdacache:caching for multi-dimensional-access memories," in *51st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 842–855, Oct 2018.

[2] P. Wang, S. Li, G. Sun, X. Wang, Y. Chen, H. Li, J. Cong, N. Xiao, and T. Zhang, "Rc-nvm: Enabling symmetric row and column memory accesses for in-memory databases," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 518–530, Feb 2018.

[3] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 267–280, ACM, 2015.

[4] B. Akin, J. C. Hoe, and F. Franchetti, "Hamlet: Hardware accelerated memory layout transform within 3d-stacked dram," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, Sep. 2014.

[5] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked dram," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 131–143, ACM, 2015.

[6] Intel, "Deep neural network library (dnnl)." https://intel.github.io/mkl-dnn/index.html. Accessed: 11/1/2019.

[7] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. Shimakawa, T. Takagi, T. Mikawa, and K. Aono, "An 8 mb multi-layered cross-point reram macro with 443 mb/s write throughput," *IEEE Journal of Solid-State Circuits*, vol. 48, pp. 178–185, Jan 2013.

[8] A. Aziz, N. Jao, S. Datta, and S. K. Gupta, "Analysis of functional oxide based selectors for cross-point memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 12, pp. 2222–2235, 2016.

[9] Z. S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (New York, NY, USA), pp. 108–120, ACM, 1997.

[10] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pp. 235–246, March 2010.

[11] C. J. Lin, S. H. Kang, Y. J. Wang, K. Lee, X. Zhu, W. C. Chen, X. Li, W. N. Hsu, Y. C. Kao, M. T. Liu, W. C. Chen, Y. Lin, M. Nowak, N. Yu, and L. Tran, "45nm low power cmos logic compatible embedded stt mram utilizing a reverse-connection 1t/1mtj cell," in *Electron Devices Meeting (IEDM), 2009 IEEE International*, pp. 1–4, IEEE, 2009.

[12] W. Zhao, S. Chaudhuri, C. Accoto, J. Klein, C. Chappert, and P. Mazoyer, "Cross-point architecture for spin-transfer torque magnetic random access memory," *IEEE Transactions on Nanotechnology*, vol. 11, pp. 907–917, Sep. 2012.

[13] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

[14] D. Kau, S. Tang, I. Karpov, R. Dodge, B. Klehn, J. Kalb, J. Strand, A. Diaz, N. Leung, J. Wu, S. Lee, T. Langtry, K. wei Chang, C. Papagianni, J. Lee, J. Hirst, S. Erra, E. Flores, N. Righos, H. Castro, and G. Spadini, "A stackable cross point phase change memory," in *2009 IEEE International Electron Devices Meeting (IEDM)*, pp. 1–4, Dec 2009.

[15] S. Salahuddin and S. Datta, "Use of negative capacitance to provide voltage amplification for low power nanoscale devices," *Nano Letters*, vol. 8, no. 2, pp. 405–410, 2008. PMID: 18052402.

[16] S. George, K. Ma, A. Aziz, X. Li, A. Khan, S. Salahuddin, M.-F. Chang, S. Datta, J. Sampson, S. Gupta, and V. Narayanan, "Nonvolatile memory design based on ferroelectric fets," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 118, ACM, 2016.

[17] S. R. Srinivasa, X. Li, M. Chang, J. Sampson, S. K. Gupta, and V. Narayanan, "Compact 3-d-sram memory with concurrent row and column data access capability using sequential monolithic 3-d integration," *IEEE Trans. VLSI Syst.*, vol. 26, no. 4, pp. 671–683, 2018.

[18] Y. Chen and Y. Liu, "Dual-addressing memory architecture for two-dimensional memory access patterns," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 71–76, March 2013.

[19] K. Bong, S. Choi, C. Kim, S. Kang, Y. Kim, and H. Yoo, "14.6 a 0.62mw ultra-low-power convolutional-neural-network face-recognition processor and a cis integrated with always-on haar-like face detector," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 248–249, Feb 2017.

[20] S. Rubin, R. Bodík, and T. Chilimbi, "An efficient profile-analysis framework for data-layout optimizations," *SIGPLAN Not.*, vol. 37, pp. 140–153, Jan. 2002.

[21] E. Z. Zhang, H. Li, and X. Shen, "A study towards optimal data layout for gpu computing," in *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, (New York, NY, USA), pp. 72–73, ACM, 2012.

[22] M. T. Kandemir, A. N. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee, "Enhancing spatial locality via data layout optimizations," in *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, (London, UK, UK), pp. 422–434, Springer-Verlag, 1998.

[23] D. Cho, S. Pasricha, I. Issenin, N. Dutt, Y. Paek, and S. Ko, "Compiler driven data layout optimization for regular/irregular array access patterns," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '08, (New York, NY, USA), pp. 41–50, ACM, 2008.

[24] Y. Zhang, W. Ding, M. Kandemir, J. Liu, and O. Jang, "A data layout optimization framework for nuca-based multicores," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 489–500, ACM, 2011.

[25] M. Wolfe, "More iteration space tiling," in *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pp. 655–664, Nov 1989.

[26] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 101–113, ACM, 2008.

[27] B. Bao and C. Ding, "Defensive loop tiling for shared cache," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, Feb 2013.

[28] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, (New York, NY, USA), pp. 233–246, ACM, 1984.

[29] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 235–246, IEEE Computer Society, 2006.