

Programmable Data Layouts via On-The-Fly Data Transformation

Cole Strickler
University of Kansas
Lawrence, Kansas, USA

Manos Athanassoulis
Boston University
Boston, Massachusetts, USA

Renato Mancuso
Boston University
Boston, Massachusetts, USA

Heechul Yun
University of Kansas
Lawrence, Kansas, USA

Abstract

The performance of modern computing systems is heavily bottlenecked by the widening gap between processing throughput and main memory latency, a problem exacerbated when an application’s logical data access patterns misalign with the physical layout of data in memory. To address this, we propose a hardware-software co-design framework comprising the Data Transformation Language (DTL) and the Data Transformation Unit (DTU). DTL allows programmers to concisely express complex data layout reorganizations, which the compiler translates into efficient hardware routing configurations. The DTU, a hardware engine logically situated between the Last-Level Cache (LLC) and main memory, transparently evaluates these configurations on-the-fly to gather scattered data elements into dense, logically contiguous cache lines. By converting explicit layout transformations into virtual views, this approach eliminates the need for explicit data materialization in software. We evaluate the DTU within a quad-core RISC-V SoC using FireSim, demonstrating substantial improvements in execution speed and memory efficiency.

Keywords: data transformation, data layout, cache locality, memory systems, hardware-software co-design

ACM Reference Format:

Cole Strickler, Renato Mancuso, Manos Athanassoulis, and Heechul Yun. 2027. Programmable Data Layouts via On-The-Fly Data Transformation. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS ’27)*, April 11–15, 2027, Heraklion, Greece. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3845814.3850077>



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS ’27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04

<https://doi.org/10.1145/3845814.3850077>

1 Introduction

Data movement between processing elements (CPUs, GPUs, and other accelerators) and main memory is a major bottleneck for both performance and energy efficiency in modern systems. This challenge has become more acute with the proliferation of data-intensive applications across machine learning, databases, and high-performance computing. Compounding the problem, memory technology scaling has lagged behind the throughput growth of contemporary processing units [20, 33, 56], creating a pronounced memory performance gap. To mitigate long-latency accesses, modern systems rely on multi-level caches, complex prefetching strategies [4, 6, 18, 25], and increased memory-level parallelism (MLP). However, the effectiveness of these mechanisms highly depends on how data is laid out in memory.

In many applications, the logical organization of data does not align with the layout required for efficient utilization of the memory hierarchy. As a result, software often performs explicit data layout transformations, materializing multiple versions of the same data object optimized for different access patterns [7]. These transformations incur substantial overheads in data movement, memory footprint, and execution time. Furthermore, they entangle algorithmic logic with layout and hardware-specific optimizations, complicating programmability and maintainability.

Existing approaches attempt to address this challenge through both software and hardware techniques. Compiler-driven frameworks and tensor systems [15, 39, 43] typically generate a single materialized layout optimized for a target computation. Supporting multiple layouts requires duplicating data, performing additional transformations, or introducing complex index mappings [21], all of which incur additional memory and runtime overhead. The Processing-Near-Memory (PNM) and Processing-In-Memory (PIM) approaches [34, 49] reduce data movement by bringing computation closer to data. Unfortunately, these approaches often restrict the class of supported computations and require significant architectural and programming model modifications.

To address these challenges, recent works have tried to build *on-the-fly* transformation engines, which focus on moving only useful data into the memory hierarchy and providing software with ideal locality. These methods have targeted

application specific access patterns including databases [36, 41, 52] and tensor operations [22]. Our design builds upon these approaches, but looks to overcome domain specificity to make cache line contents generally *programmable*, enabling a unified abstraction applicable across diverse workloads.

We propose a hardware–software co-design framework comprising the Data Transformation Language (DTL) and the Data Transformation Unit (DTU). DTL provides a concise, C-like specification that allows programmers to express how logical data coordinates map to physical memory addresses. The DTU, a hardware engine written in Chisel HDL [8] and situated between the LLC and main memory, evaluates these compiler-generated mappings on-the-fly. By dynamically gathering scattered positional data into dense, logically contiguous cache lines, the DTU achieves ideal cache locality. Crucially, this framework converts explicit layout transformations into virtual views, eliminating materialization overheads, reducing memory footprint, and operating entirely without modifications to existing computation kernels.

In summary, we make the following contributions:

- We introduce a formal abstraction that decouples logical data access from physical memory layout, enabling layout transformations to be expressed as programmable address mappings.
- We design and implement the *Data Transformation Unit* (DTU), a hardware engine that constructs programmable cache lines on-the-fly.
- We develop the *Data Transformation Language* (DTL) and compiler infrastructure to express and realize transformation mappings.
- We provide a full-system implementation—including synthesizable hardware, compiler toolchain, OS driver and runtime library—and evaluation on a RISC-V SoC in FireSim, demonstrating that programmable data transformation is both practical and effective across a range of workloads.

2 Background

In this section, we provide background on the importance of data layouts and on recent work on domain-specific on-the-fly data transformation engines.

2.1 The Data Layout Challenge

Modern computing systems rely heavily on multi-level cache hierarchies and hardware prefetchers to bridge the widening performance gap between the CPU and main memory [56]. Caches exploit spatial and temporal locality by fetching data from main memory in fixed-size contiguous blocks, known as cache lines. Hardware prefetchers complement this by monitoring memory access streams and proactively fetching adjacent or predictably strided cache lines before the CPU explicitly requests them [27].

However, these mechanisms are highly optimized for dense, predictable memory access patterns. While data-intensive applications have been heavily optimized to generate such access patterns (e.g., column-stores for data analytics [2, 10]), providing the optimal layout for ad hoc queries remains an open question [7, 41], because the access patterns remain unpredictable. In a similar vein, data-intensive applications involving high-dimensional tensor manipulations often exhibit data traversals that are misaligned with the physical memory layout [51]. This misalignment destroys spatial locality. The CPU is forced to fetch full cache lines only to consume a small fraction of the loaded elements, leading to poor cache line utilization and inflated working set sizes. Furthermore, the resulting irregular, large-stride access patterns easily defeat traditional hardware prefetchers, leading to frequent cache misses, saturated memory bandwidth, and stalled compute pipelines.

The need for well-aligned, densely packed data is even more pronounced when using Single Instruction Multiple Data (SIMD) instructions and other hardware accelerators that exploit data parallelism [17]. These vector execution units require contiguous memory operands to achieve peak computational throughput. The performance degradation caused by irregular data is so severe that highly optimized software libraries frequently resort to explicitly duplicating data into a reorganized, contiguous layout prior to execution [2, 14]. Despite the execution time required for this data movement and the memory footprint overhead incurred by materializing intermediate views (e.g., *im2col* for convolutions), this software-driven data duplication remains a necessary trade-off to unlock optimal compute performance.

2.2 Domain-Specific Data Transformation Engines

To address the severe performance penalties associated with data layout mismatches, recent research has explored hardware-accelerated, on-the-fly data reorganization. These approaches introduce specialized transformation engines situated between the CPU and main memory. By intercepting memory requests, these engines dynamically gather non-contiguous data elements from the backing store and repack them into dense, optimal cache lines before delivering them to the processor’s cache hierarchy.

Notable examples of this paradigm include Relational Memory (RME) [41] and the Tensor Memory Engine (TME) [22]. RME is specifically tailored for relational database systems; it dynamically extracts and packs target columns from a base row-store layout, presenting contiguous columns to the CPU [41, 52]. Similarly, TME is designed to accelerate tensor operations by executing on-the-fly multi-dimensional slicing, transposing, and folding, feeding dense data streams to the processing pipelines.

While both RME and TME successfully demonstrate the benefits of decoupling access pattern from physical layout,

they are fundamentally constrained by their domain-specific designs. RME relies on hardware logic tailored specifically to database geometry (e.g., row sizes and column offsets), whereas TME is tightly coupled to fixed tensor stride manipulations. This lack of generality means that neither engine can be easily adapted to accelerate arbitrary layout transformations across broader application domains. Consequently, there remains a critical need for a fully programmable, generalized data transformation unit capable of supporting arbitrary, on-the-fly memory re-mappings.

3 Motivating Example

To motivate on-the-fly data transformation, consider a naive matrix multiplication $C = AB$, where A and B are 16×16 matrices stored in row-major order. Assume each matrix element occupies 4 bytes and the cache-line size is 64 bytes. A conventional matrix-multiplication kernel computes each element of C by taking the dot product of a row of A and a column of B .

The accesses to A have good spatial locality because consecutive elements of a row are contiguous in memory. The accesses to B , however, are strided. Since each row of B contains 16 elements, it occupies exactly one 64-byte cache line. Consequently, two consecutive elements of the same column of B are separated by 64 bytes and reside in different cache lines. A traversal of one column therefore touches 16 different cache lines while consuming only one 4-byte element from each line. This mismatch between the stored layout of B and the access pattern of the computation results in poor spatial locality and can generate substantially more cache and memory traffic than the amount of useful data consumed.

A common optimization is therefore to transpose B before performing the multiplication. After transposition, the elements that formed a column of B form a contiguous row of B^T . The multiplication can then traverse both A and B^T sequentially, making effective use of each fetched cache line. In this example, the 16 elements required from B for a dot product occupy 16 separate cache lines in the original layout, but exactly one 64-byte cache line in the transposed layout.

However, this software-based transposition requires additional computation to rearrange the matrix, additional data movement to read and rewrite its elements, and additional memory capacity to store the transformed intermediate representation—all of which are undesirable overheads.

Our approach avoids these costs by performing the transformation in hardware on the fly, without materializing the transposed matrix in physical memory. We return to this example throughout the paper to illustrate the design and operation of the system. The remainder of the paper develops the abstraction, programming interface, and hardware mechanisms that enable this capability.

4 Data Transformation

This section introduces *data transformation* and its underlying theoretical framework. The fundamental principle is the decoupling of the application’s *logical view* (how memory objects are addressed) from its *positional view* (how objects are serialized in memory), defined formally as follows:

Definition 4.1 (Positional Address Space). A *positional address space* is an address space in which data addresses directly encode the position in memory where the data item to be accessed is stored. Once an object is materialized in memory, it has a unique positional address space. Data addresses in this space are henceforth called *positional data addresses* or simply *positional addresses*.

Definition 4.2 (Logical Address Space). A *logical address space* is an address space where the layout progression of addresses reflects the order of data items as required by the semantics of the algorithm or application. Data addresses in this space are henceforth called *logical data addresses* or simply *logical addresses*.

For example, considering the matrix multiplication example in Section 3, the original matrix B resides in a positional address space following the way it is stored in memory, whereas the proposed approach enables the definition of a new logical address space to directly access B^T .

Note that the concepts of logical and positional addresses are orthogonal to the concepts of *virtual* and *physical* addresses. Indeed, from our perspective, a virtual-to-physical translation only establishes a 1:1 mapping of positional data addresses. As such, both positional and logical address spaces can be defined in virtual or physical address space.

In traditional computing systems using positional addresses, the burden of computing the correct position of each data item falls on the application logic. Because the architectural design of memory hierarchy components (e.g., caches, interconnects, and memory controllers) favors cases where the access pattern matches the positional view, a mismatch often results in a significant performance impact. Addressing the inefficiencies arising from the mismatch has been the focus of extensive literature, as discussed in Section 2.

In this paper, we propose a programmable, **hardware-assisted data transformation** approach to address mismatches between an application’s data access pattern and the positional layout of its data objects.

Definition 4.3 (Data Transformation). *Data transformation* is the on-the-fly translation of logical data addresses accessed by an application into their corresponding positional data addresses. This allows data items that are positionally scattered to appear as logically contiguous to the application, without requiring explicit data re-layout in positional space.

Leveraging data transformation enables the proactive assembly of multiple useful data items from non-contiguous

locations into a single, contiguous l -byte cache line. Given an item size of d bytes, constructing one such cache line requires gathering l/d items.

4.1 Logical to Positional Mapping

Consider a n -dimensional data object (tensor) $\chi^{D_n \times D_{n-1} \times \dots \times D_1}$ following the layout needed by the application semantics, where $\{D_n, D_{n-1}, \dots, D_1\} \in \mathbb{Z}^+$. Any logical address for a data item in χ has a unique set of *logical coordinates*. Abusing a C-like notation, it is easy to see that the logical address corresponding to the element $\chi[x_n][x_{n-1}] \dots [x_1]$ has logical coordinates $(x_n, x_{n-1}, \dots, x_1)$. Furthermore, we note that the mapping between logical coordinates and logical addresses is always reversible.

First, the *index function* L is used to compute the logical address corresponding to logical coordinates $(x_n, x_{n-1}, \dots, x_1)$:

$$L(x_n, x_{n-1}, \dots, x_1) = \sum_{i=1}^n x_i S_{D_i}, \quad (1)$$

where S_{D_i} are dimension strides defined as follows:

$$S_{D_1} = 1, \quad S_{D_i} = D_{i-1} S_{D_{i-1}} \quad \text{for } i > 1$$

Next, given a logical address, the *inverse index function* L^{-1} recovers the full set of logical coordinates from a logical linear address:

$$L^{-1}(\text{addr}) = \begin{cases} x_n = \text{addr} // S_{D_n} \\ x_{n-1} = (\text{addr} \bmod S_{D_n}) // S_{D_{n-1}} \\ x_{n-2} = ((\text{addr} \bmod S_{D_n}) \bmod S_{D_{n-1}}) // S_{D_{n-2}} \\ \vdots \\ x_1 = (\text{addr} \bmod S_{D_n}) \dots \bmod S_{D_2} // S_{D_1} \end{cases} \quad (2)$$

Once the logical coordinates have been recovered, the corresponding positional address of the data item can be computed using a function $F(x_n, x_{n-1}, \dots, x_1)$ with the following interface:

$$F : (\mathbb{Z}^+)^n \rightarrow \mathbb{Z}^+. \quad (3)$$

Figure 1 illustrates this. In theory, F can be any function of the form given in Eq. (3). In practice, however, the definition of F is constrained by hardware considerations such as available memory, pipeline latency, functional unit composition, and available pipeline depth/width. These constraints determine the class of functions that can be efficiently realized in hardware. The pipeline structure can be instantiated and parameterized at design time to include desired function subclasses. F is not required to preserve contiguity, stride regularity, dimension ordering, or any other property of the original layout. This decoupling of logical coordinates from positional placement is what enables complex re-mappings and, in particular, the construction of dense, high-utility cache lines.

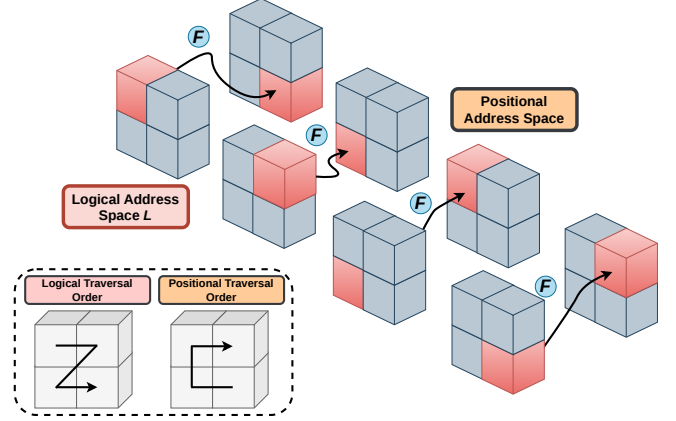


Figure 1. F maps a logical coordinate to a positional address.

Example. Consider the matrix B introduced in Section 3, which is allocated with dimensions 16×16 and stored positionally in row-major order. Suppose we request a logical view of B^T over this backing store. When accessing the 55th item of B^T in logical space, its coordinates (x_2, x_1) are recovered by applying $L^{-1}(55) = (55 // 16, 55 \bmod 16) = (3, 7)$, corresponding to element $B^T[3][7]$. The mapping function $F(x_2, x_1) = 16x_1 + x_2 = 115$ then computes the corresponding positional address of $B[7][3]$ in the original matrix B .

4.2 Efficient Cache Line Packing

Recall that data transformation allows filling each cache line of size l with l/d items that are sequential in logical address space. Sequential traversal in logical space can be performed efficiently without repeatedly applying L . Indeed, once the set of logical coordinates $(x_n, x_{n-1}, \dots, x_1)$ is recovered, the coordinates of the subsequent data item can be computed using the *coordinate successor function* defined in Alg. 1.

Algorithm 1 Coordinate Successor Function N

```

1: Input:  $\vec{C}_j = \{x_n, x_{n-1}, \dots, x_1\}$  with  $0 \leq x_i < D_i$ 
2: Output:  $\vec{C}_{j+1} = \{x_n, x_{n-1}, \dots, x_1\}$ 
3: for  $i = 1$  to  $n$  do
4:    $x_i \leftarrow x_i + 1$ 
5:   if  $x_i < D_i$  then
6:     break
7:   else
8:      $x_i \leftarrow 0$ 
9:   end if
10: end for
11: return  $(x_n, x_{n-1}, \dots, x_1)$ 

```

Using L^{-1} , F , and N , we can construct a packed cache line starting from any logical linear address using Alg. 2. The transformation is semantically powerful because F is not bound to a fixed structural template and, in principle, may be chosen from a broad class of functions, subject to hardware realizability.

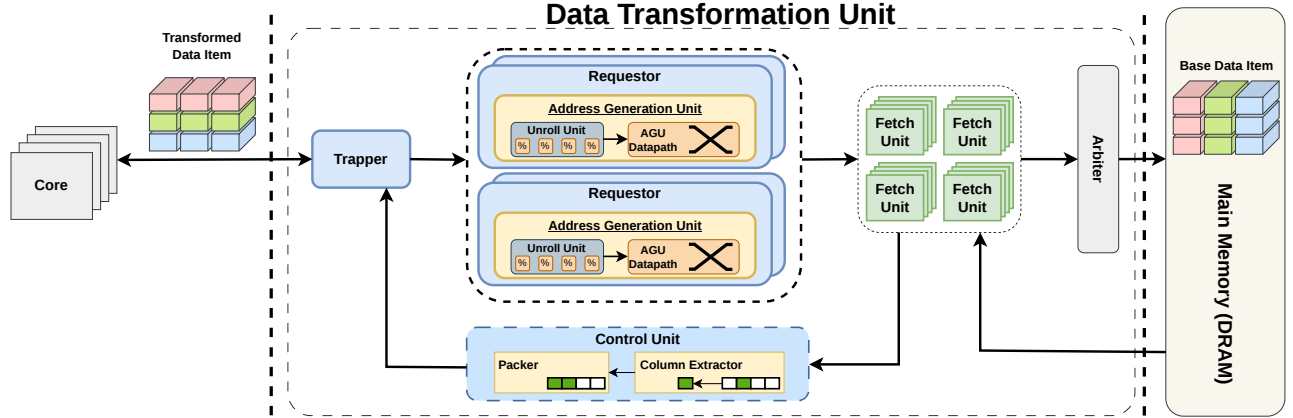


Figure 2. Bird’s eye view of the DTU design. The DTU is instantiated between the processor-side LLC and main memory. It traps cache line misses for a set of registered transformed data objects. The DTU dynamically constructs the content of cache lines to be returned. This is done by applying the corresponding DTL kernel to retrieve contiguous logical data items that are scattered in positional space.

Algorithm 2 Data Transformation Algorithm

```

1: Input:  $logical\_addr$  = logical linear address
2: Input:  $l$  = cache line size
3: Input:  $d$  = data size
4: Output:  $\vec{P}$  = packed cache line
5:  $\vec{P} = \langle \rangle$ 
6:  $\vec{coords} = L^{-1}(logical\_addr)$ 
7:  $needed\_items = l/d$ 
8:  $i = 0$ 
9: while  $i < needed\_items$  do
10:    $positional\_address = F(\vec{coords})$ 
11:    $data = read\_memory(positional\_address, d)$ 
12:    $\vec{P}.push\_back(data)$ 
13:    $\vec{coords} = N(\vec{coords})$ 
14:    $i = i + 1$ 
15: end while
16: return  $\vec{P}$ 

```

5 System Overview

We now introduce the architectural support and programming infrastructure for hardware-assisted data transformation. The proposed system consists of two main elements, namely the Data Transformation Unit (DTU) and the associated Data Transformation Language (DTL). Together, they enable programmable on-the-fly data transformation, avoiding the need to materialize transformed layouts in memory. The DTU is logically instantiated between the LLC and main memory. It is programmed using layout transformation kernels written using the provided DTL and compiled using the associated DTL compiler.

5.1 Data Transformation Unit (DTU)

The DTU is a hardware module that exposes a programmable interface for providing transformed views of data objects.

Figure 2 provides an overview of the DTU. The DTU effectively converts a single cache line request into (potentially) multiple main memory accesses for individual data items. It then reassembles the result into a cache line returned to the processor according to a programmable layout transformation kernel. This operation corresponds to the retrieval of a set of data items that are sequential in logical space but scattered in positional space.

As mentioned in Section 4, data transformation can be performed entirely in virtual or physical address space. For simplicity of discussion and implementation, we consider a DTU design operating with physical addresses. As such, virtual-to-physical address translations are performed before requests reach the DTU, and both logical and positional addresses are always physical addresses. The proposed DTU logically sits between the LLC and main memory and traps any transaction that falls within a configurable range of physical addresses, called the *shadow range*. The shadow range is chosen to be non-overlapping with the range of physical addresses assigned to main memory or any other memory-mapped device and is exposed via a TileLink [48] Manager node [5]. Normal memory accesses do not fall within the shadow range, are not routed through the DTU, and incur no extra access latency.

Consequently, for requests targeting memory within the shadow range, the DTU sits on the critical path to main memory. This placement allows the DTU to intercept memory requests, issue independent requests to main memory, and reorganize the returned data before it is delivered back to the cache hierarchy. Due to this placement, any additional delay introduced by the DTU directly increases the reorganized data access latency. As a result, the DTU microarchitecture must be designed to offer a compromise between supporting

rich and programmable transformations while minimizing reorganization latency.

As shown in Figure 2, the DTU is composed of several pipeline stages that perform specialized functions. A single DTU supports the simultaneous configuration of multiple data transformation kernels. The **Trapper** module is responsible for receiving LLC misses handled by the DTU and generating compliant responses. The Trapper also matches incoming requests against the set of active configurations to determine how the request should be processed.

Each **Requestor** module instance handles a single kernel and transforms an incoming request into a sequence of outbound memory accesses. The Requestor effectively recovers logical coordinates from incoming logical addresses and issues the necessary sequence of positional data accesses to construct cache lines of logically contiguous data items. The internal machinery responsible for this process is the **Address Generation Unit (AGU)**, which contains two components: the **Unroll Unit** and the **AGU Datapath**. The Requestor provides the AGU with incoming requests and associated metadata, while the AGU generates the memory addresses used for outbound memory accesses.

Upon receiving a request, the AGU first passes the address to the Unroll Unit, which evaluates L^{-1} (Eq. (2)) to recover the logical coordinate vector. These logical coordinates are then written into registers within the AGU Datapath. The coordinates are subsequently dispatched for computation, after which the coordinate successor algorithm is applied (Algorithms 1 and 2). Specifically, the AGU Datapath evaluates the compiler-generated function F (Eq. (3)) for each coordinate vector until enough addresses have been generated to construct a full cache line.

The Requestor receives each generated address, converts it into a bus-aligned memory request, attaches metadata (e.g., cache line placement and valid-data mask), and forwards the request to the **Fetch Unit**. The Fetch Unit forwards requests to main memory and returns responses to the **Control Unit**. Within the Control Unit, the **Column Extractor** extracts the relevant data from each response and forwards it to the **Packer**. The Packer uses the metadata generated by the Requestor to place each extracted element in its correct position within the cache line being constructed. Once a full cache line has been assembled, the Control Unit forwards it to the Trapper, which returns the response upstream to the cache hierarchy.

Nearly every component of the DTU can be parameterized, allowing the design to be instantiated with different performance characteristics. The parameters include the supported configurations, pipeline width, memory-level parallelism, and AGU characteristics (i.e., the set of supported F and L).

5.2 Data Transformation Language (DTL)

The DTL is a small domain specific language (DSL) for concisely defining layout transformation kernels. Due to space

constraints, we provide a full formal DTL specification in the appendix in Listing 3 and use this section to describe its key features. Our DTL allows the programmer to provide a concrete instantiation of L , L^{-1} , N , and F —see Section 4. These functions are implicitly specified by the structure of a DTL kernel, which closely resembles C code to remain friendly to end-users.

Consider once again the simple example of a matrix **B** positionally stored in row-major order and logically accessed as its transpose B^T . Listing 1 shows an example of a DTL kernel that exports the logical view of B^T , assuming that **B** is a 16×16 matrix.

Listing 1. Example DTL kernel for implicitly transposing the 16×16 matrix **B** from Section 3.

```

1 int ROWS = 16;
2 int COLS = 16;
3
4 for (int col = 0; col < COLS; col++) {
5   for (int row = 0; row < ROWS; row++) {
6     out = row*COLS+col; } }
```

From the above example and with reference to the full specification, we note a few distinctive structural features of a DTL kernel. First, a set of integer constants can be declared (Line 2). Second, the main body of the kernel must follow a set of nested **for** loops (Lines 4 and 5). Each **for** loop must declare and initialize a fresh loop variable, provide a loop condition using the less-than "<" operator against an explicit or previously-defined constant, and include a plus-one increment rule for the current loop variable. The innermost loop must contain one or more **out** statements (Line 6). The right-hand side of the **out** statement calculates the positional address at the current iteration. Furthermore, the innermost loop body contains an optional conditional (one of **>**, **≥**, **<**, **≤**, **isEven?**, **switch**, **pad?**, or **isEdge?**) followed by one or more **out** statements. Parameters to conditional statements can only be loop variable identifiers. Each conditional branch must contain the same number of out statements such that L and L^{-1} are efficiently computable. Additionally, a limited number of fixed-size data arrays can be indexed within an out-statement, and multiple out-statements in the innermost loop can be used, effectively defining an additional dimension whose bounds equal the number of out-statements and, therefore, extend the logical coordinate space accordingly. These capabilities further enhance the generality of DTL, enabling a single kernel to represent a much broader class of transformations.

Despite its resemblance to imperative C-like syntax, a DTL kernel does not execute iteratively in the conventional sense. Rather, it defines a mapping F from the logical coordinate space (the Cartesian product of the loop index domains) to offsets within a data object in positional space. The logical coordinate space defines a multi-dimensional data object, where each loop corresponds to a dimension, the bounds define that dimension's extent, and the order of loop nesting determines the linearization order. From this dimensional

definition, an access stride structure and thus a concrete instantiation of L and L^{-1} are derived, while the traversal order over the logical coordinate space defines the coordinate successor function N .

The **out** statement in the innermost loop is the concrete realization of F : it computes an offset address as a function of constants and loop indices, identifying the data element to be accessed in the positional layout. In our current implementation and targeted benchmarks, we support the following set of operators within F : $\{+, -, *, \text{nop}\}$. These elements are then exposed in the traversal order of the logical coordinate space, thereby defining the linearized layout of the transformed data object. Intuitively, the DTL kernel exposes a transformed logical view of the base data object, as if C code of similar structure was actually executed to materialize the transformation.

Listing 2 showcases a more complex DTL kernel that takes slices along the depth of a 3D tensor and alternately rotates each 2D slice left or right. DTL supports a restricted class of conditionals that allow the mapping function to vary as a function of the coordinate vector. To do so, the kernel uses the conditional operator `isEven?` which allows construction of piecewise mappings over the logical coordinate space. These predicates are evaluated using bounded combinational logic and add no additional pipeline latency.

Listing 2. Example DTL kernel applying alternate left-/right rotations to each slice of a 3D tensor.

```

1 int stride_d = 400*400;
2 int stride_h = 400;
3 int MAX_DEPTH = 100;
4 int MAX_HEIGHT = 400;
5 int MAX_WIDTH = 400;
6
7 for (int d = 0; d < MAX_DEPTH; d++) {
8     for (int w = 0; w < MAX_WIDTH; w++) {
9         for (int h = 0; h < MAX_HEIGHT; h++) {
10             if (isEven? d) // Rotate Right
11                 out = d*stride_d+(MAX_HEIGHT-1-h)*stride_h+w;
12             else // Rotate Left
13                 out = d*stride_d+h*stride_h+(MAX_WIDTH-1-w); } } }

```

5.3 DTU Runtime and API

In addition to the DTU hardware and the DTL compiler, we also provide the necessary runtime drivers and programming APIs to leverage DTU functionalities in user applications. These components include (1) DTU kernel compilation and registration, (2) memory management and access methods, and (3) synchronization primitives. The runtime is implemented in part as an **OS DTU driver** and in part as a **DTU user-space application library**.

At system initialization, the OS driver allocates a contiguous range of physical addresses backed by main memory that will be used to store base objects. It also allocates the shadow range from where physical addresses for non-materialized transformed objects are allocated. The driver also instantiates a memory manager that will handle memory allocations for both base objects and transformed logical objects. Finally,

the driver instantiates low-level bidirectional communication interfaces with user-space that are used by the DTU library. The primary abstraction exported to user applications by the DTU library is the *transformation handle*. The transformation handle is an object that encapsulates all the methods and resources required to register and use a DTU kernel, and to interact with logical objects. A high-level view of this interface is shown in Figure 3.

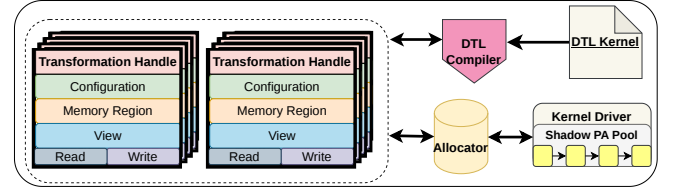


Figure 3. DTU runtime overview: transformation handles encapsulate the resources necessary for DTL kernel deployment.

The first step for an application is to allocate enough memory, through the library-exported API, to store the base object in positional space. A successful first step returns a valid transformation handle to the application. Upon success, the following two resources are set up: (1) *Memory Region*: a continuous memory-backed range of physical addresses for the base object; (2) *Write Port*: a virtual-to-physical mapping to this range is established. The application can use this write pointer to initialize/update the base object. Writes to the base object use non-temporal (NT) stores targeting the DRAM-backed region, ensuring that updates bypass the cache hierarchy and are immediately visible to the DTU.

Second, the application registers a compiled DTU kernel using the previously obtained handle. Upon successful registration, the size of the logical object is computed and the following is set up: (1) *Configuration*: a Requestor module inside the DTU is configured with the provided kernel and the resulting module configuration is reflected in the handle; (2) *View*: a continuous range of shadow physical addresses for the transformed object; (3) *Read Port*: a virtual-to-physical mapping such that a read pointer can be used by the application to access the transformed logical data object. The read pointer can then be used to access the logical view of the object, and can be passed in to unmodified compute kernels expecting the logical layout.

Listing 3 shows an example matrix-multiplication workflow using the DTU to perform an on-the-fly matrix transpose. First, the application allocates positional memory space for matrix **B** (Line 3) and initializes the matrix (Line 8). Next, the transpose DTL kernel (Listing 1) is compiled (Line 11) and programmed into the DTU (Line 15). Finally, the application obtains a pointer to the transformed B^T in logical address space (Line 18), which is passed directly to the matrix-multiplication kernel (Line 21).

```

1 // allocate enough size to store base object,
2 // reserve hardware resources
3 DTL::Handle handle =
4     dtlLib::AllocDTUHandle(16*16*sizeof(T));
5
6 // initialize matrix B
7 T* B = handle->GetWriteRegion();
8 init_matrix(B, rows=16, cols=16);
9
10 // compile std::string dtl_kernel from Listing 1
11 DTL::Kernel kernel =
12     DTL::Compile(dtl_kernel);
13
14 // program the compiled kernel to the DTU
15 DTL::ProgramHardware(handle, kernel);
16
17 // shadow region read access
18 T* B_T_view = handle->GetReadRegion();
19
20 // compute over the logical view using an unmodified method
21 matmul_pretransposed(A, B_T_view, C, M=16, N=16, K=16);

```

Listing 3. DTU runtime and API workflow for the matrix-multiplication example introduced in Section 3. The code initializes the positional representation of B and accesses the resulting B^T logical view using an otherwise unmodified matrix-multiplication kernel.

5.4 Coherence and Synchronization

Note that all read accesses to the transformed object are internally issued through shadow physical address (SPA) regions, which lie outside the physical address range of main memory and provide a level of indirection over the underlying in-memory object that enables address translation by the DTU. However, this indirection alone does not guarantee data freshness. For example, a transformed read may bring a cache line into the cache hierarchy, after which the base object in main memory is updated, leaving the cached line stale. To ensure that subsequent reads observe updated data, we employ *SPA remapping*. Specifically, when visibility of updates is required, the runtime allocates a new SPA region and remaps the virtual addresses to this new region, forcing future accesses to hit a different physical address. Because LLCs are physically tagged, cache lines associated with the previous mapping do not alias with the new one, preventing stale data from being reused without requiring explicit cache invalidation. This mechanism enables multiple simultaneous views of the same data object and provides software control over when updates become visible. Consequently, multiple cores can safely hold different transformed views of the same base object concurrently.

SPA remapping requires updating page-table entries, invalidating corresponding TLB entries, refetching data from DRAM, and executing a few MMIO writes to the DTU’s configuration registers. Although these costs are non-trivial, remapping is seldom required given the discrete read/write phase-based access patterns of our target workloads, allowing these overheads to be well amortized. Furthermore, using hugepages greatly reduces the TLB invalidation overhead.

Lastly, because the shadow PA space is finite, stale cache lines could potentially be encountered if address range reuse occurs due to wraparound. Although extremely unlikely in practice (for instance, in a RISC-V Sv48-based [55] virtual memory system, this range spans from the maximum memory-backed address up to $2^{47} - 1$), correctness is guaranteed by issuing a cache flush upon wraparound—a cost that is well amortized due to its rare occurrence.

6 Implementation

This section describes the implementation of the DTU’s Address Generation Unit (AGU) and the DTL compiler.

6.1 Address Generation Unit (AGU)

Address generation is fully decoupled from the rest of the system. A well-defined interface between the Requestor and the AGU enables a producer–consumer interaction model. The Requestor activates the AGU by passing in a new request and in turn the AGU will output a sufficient number of addresses to construct a cache line. The AGU has two key components: the **Unroll Unit** which is the physical realization of L^{-1} (Eq. (2)), and the **AGU Datapath** which performs the computation defined by F (Eq. (3)). Putting this together, the Unroll Unit recovers (via L^{-1}) the coordinate vector from an input linear logical address (L), which is used to initialize the AGU Datapath; the datapath then iteratively applies F and the coordinate successor function N to generate the sequence of addresses required to construct a cache line.

Unroll Unit. The Unroll Unit transforms a logical linear address into its corresponding coordinate vector, providing a programmable realization of Eq. (2). Computation of L^{-1} requires a series of integer divisions, which have notoriously long multi-cycle latency in their hardware implementation [37]. Because these divisions must be performed sequentially on the critical path to main memory, these multi-cycle divisions could easily dominate any performance gains made by improved cache locality. To combat this, we use the insight that every divisor is known at compile time to perform each operation using the *magic division algorithm* [54]. The magic division algorithm allows an integer division operation to be performed using a precomputed set of “magic” numbers and multiplication, addition, and bit shift operations instead of costly hardware divisions. This algorithm allows each division to be performed in two deterministic cycles, avoiding a complex long or variable latency implementation while maintaining a high operating frequency.

Given a DTL kernel, the compiler (Section 5.2) extracts stride information from the loop nest that defines the linearization order and uses it to compute the corresponding magic constants. These constants parameterize the divisions in Eq. (2) and are written to configuration registers within the Unroll Unit.

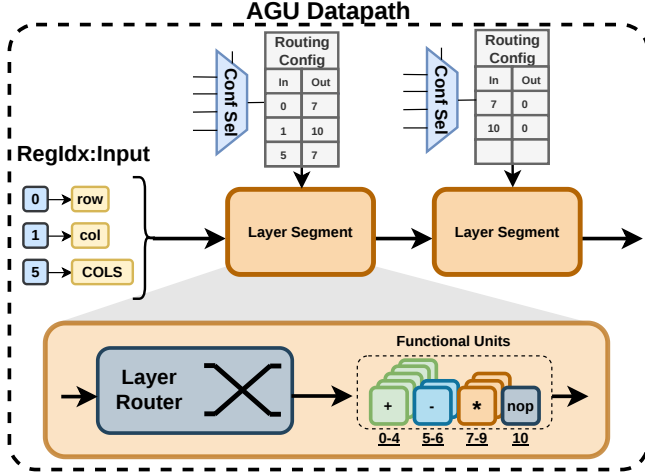


Figure 4. AGU datapath configured to compute the $F(row, col) = row * COLS + col$ mapping given by the matrix transpose DTL kernel of Listing 1.

The Unroll Unit can be flexibly instantiated to support any number of maximum dimensions or offset widths. Additionally, the DTU allows for runtime configuration with different coordinate dimensions. Accordingly, the Unroll Unit intelligently utilizes dimensional knowledge of the data object to only perform the needed number of divisions - avoiding incurring unnecessary latency. Once coordinates are recovered, they are passed to the next stage of the pipeline: the AGU Datapath.

AGU Datapath. The AGU Datapath takes a logical coordinate vector as input and produces the sequence of addresses corresponding to the data items that compose a cache line. It provides a programmable realization of the functions F and N . Function F maps each coordinate vector to its corresponding positional address, while N defines the progression to the next coordinate. The ordering induced by N matches the iteration order of the nested loop structure in a DTL kernel.

Operationally, each coordinate vector is fed into a pipelined computation that evaluates F to produce an address. The coordinate is then updated according to N , and this process repeats until a sufficient number of addresses have been generated to construct a cache line.

The AGU Datapath is organized as a sequence of pipeline stages composed of simple functional units. In our implementation, these units are $\{+, -, *, \text{nop}\}$, matching the operator set currently supported by DTL. Each stage can be instantiated with a configurable composition of functional units, while programmable routing modules between stages map inputs to the appropriate units to realize F . A high level view of the AGU Datapath is shown in Figure 4.

This layered organization enables tight pipelining across successive coordinate vectors, allowing each computation to overlap with its predecessor and amortize pipeline latency.

The routing configuration is a simple input-to-output mapping, and serves as the compiler target for a DTL kernel.

6.2 DTL Compiler

Each DTL kernel is compiled into a layout transform configuration, which includes the parameterization of the Unroll Unit, the coordinate successor function N , and the routing configuration for the AGU Datapath. Configurations can be compiled online or offline and can be dynamically attached, removed, or saved at runtime. The extraction of the Unroll Unit configuration and coordinate successor function from a kernel was described in the previous section; the routing configuration, which realizes F , is derived from the kernel’s **out**-statements. Each **out**-statement specifies a sequence of operators and operands, with operands mapped to hardware locations and operators mapped to functional units in the AGU Datapath. The dependency structure of these expressions forms an abstract syntax tree, where leaves correspond to inputs and higher nodes depend on the results of lower nodes. Operations at the same tree level can be scheduled onto the same layer of the AGU Datapath, allowing the compiler to greedily construct a VLIW-like schedule to maximize per-cycle throughput and minimize pipeline depth. Each functional unit has a layer-specific index, and the compiler realizes a schedule by defining an input index to output index mapping for each router module between layers. The compiler is aware of the current DTU parameterization, and attempts to realize a schedule that fits the current parameterization of functional units and layer count by moving operations across layers when legal. Our compiler implementation also supports a number of standard optimization passes (i.e., constant propagation, constant folding, dead code elimination, value numbering, copy propagation) for reducing resource count and increasing schedule quality. An example schedule for the running matrix transpose example is shown in Figure 4.

The primary architectural constraint is that the DTU lies on the critical path to memory; consequently, any latency introduced within the DTU directly increases memory access latency. Minimizing this internal latency therefore drives both the hardware and compiler design. To address this, the system enables flexible computation schedules that execute multiple operations per cycle, while tightly pipelining successive computations of F . This approach significantly expands the space of profitably realizable layout transformations, providing a highly general and efficient framework for mapping complex transformations onto the hardware.

6.3 DTU Access Latency

Normal memory accesses bypass the DTU logic entirely. Therefore, normal memory accesses experience no additional latency. The shadow range is exposed via a TileLink Manager node, and only memory accesses targeting this range will be routed to the DTU.

Accesses to DTU-transformed objects are routed through the DTU and therefore incur additional latency relative to ordinary memory accesses. This latency is not constant: producing a single logical cache line may require fetching data from multiple, potentially disparate, positional locations in DRAM. The total latency thus depends on both the specific transformation being performed and the number and location of the underlying memory accesses required to construct the response.

The structural latency of the DTU pipeline itself—including address generation, routing, and index-calculation logic—ranges from 13 to 27 cycles depending on the active DTL kernel. This variation stems from the variable number of integer divisions required to compute L^{-1} in the Unroll Unit. Even at the high end of this range, the structural latency overhead is small relative to a conventional DRAM access, which typically requires hundreds of cycles. Moreover, prior work has shown that prefetching can effectively hide much of the additional latency introduced by on-the-fly transformation [52].

7 Evaluation

In this section, we evaluate the DTU across a range of workloads and configurations to quantify its impact on execution time, memory system behavior, and hardware cost.

7.1 System Setup

We use FireSim [28] as our simulation environment. FireSim is an FPGA-accelerated, cycle-exact, full-system simulator. It enables simulation of a complete SoC, with the FPGA operating at 100 MHz while the simulated target runs at a configurable frequency. Simulation results are not affected by the FPGA speed limitation, as a specialized compiler infrastructure decouples the target design from the FPGA host [32], enabling realistic performance evaluation. On FireSim, we instantiate a quad-core Berkeley-Out-Of-Order Machine (BOOM) [13] RISC-V SoC using Chipyard [5] and implement the proposed DTU hardware block, written in Chisel HDL [8], within the SoC, operating in the memory bus clock domain at 1GHz. We use Linux kernel version 6.2 and a Buildroot-based root filesystem [11] as the runtime environment. The details of our platform are shown in Table 1.

Table 1. Evaluation platform specifications.

Component	Specifications
CPU	4x BOOM@3.2GHz, Out-Of-Order, 3-wide, ROB: 96, LSQ: 24/24 L1: 32K(I)/32K(D) (8-Way), 6 MSHRs
Prefetcher	L1 4-Ahead Multi Next Line [9]
Shared L2 Cache	1MB (16-way), 1 Bank [1]
DTU	16 Fetch Units, 8 Requestors, 1GHz
Main Memory	4GB DDR3 1066 MHz, 1 Rank, 8 Banks
Operating System	Buildroot Linux v6.2.0

7.2 Benchmarks

For benchmarks, we use the following five representative workloads:

Database Column Projection refers to a database query operation of the form `SELECT C1, C2, . . . CN FROM TABLE`, which is commonly used in analytical processing. The performance of this query is highly sensitive to whether the database uses column- or row-wise storage and the number of projected columns. In this benchmark, we run this query on an in-memory database using column-wise, row-wise storage, or DTU over a row-wise storage backend. The goal is to show that starting with a row-store backend, DTU allows us to always unlock the benefits of a column-store [3]. Note that the DTU can enable the development of a hardware-assisted Hybrid Transactional/Analytical Processing [41, 50, 58] engine that exhibits the benefits of a row-store for transactions and a column-store for analytical queries. The benchmark we use here is adopted from the RME framework [41].

Im2Col is an optimization technique used in convolutional neural networks to speed up convolution operations by converting them into standard matrix multiplications. Instead of sliding a filter across an input image using slow, nested loops, Im2Col extracts every overlapping image patch the filter will process, flattens them into individual columns and assembles them into a single large 2D matrix. The convolutional filters are similarly flattened into another matrix. By restructuring the data in this manner, the entire convolution step can be computed simultaneously as a single, optimized General Matrix Multiplication (GEMM). For this benchmark, we apply Im2Col over a 720×1080 RGB image, followed by a GEMM, to execute a convolution operation with one filter. For the CPU baseline, we compare against the Darknet [26] Im2Col implementation.

Image Augmentation is a preprocessing technique used in training vision networks meant to improve generalization and prevent overfitting. For this benchmark, we perform a fused NHWC to NCHW permutation and augmentation over a batch of images [31]. We perform augmentation from the set {RotateLeft, RotateRight, ReflectX, ReflectY, RotateLeft+ReflectX, RotateRight+ReflectX, ReflectX+ReflectY} producing seven new unique views of each image in the batch. We follow the augmentation with an intensity scaling operation that further doubles the set of augmented images.

Tensor Unfolding is an operation that converts a higher-order tensor into a matrix (matricization) such that one axis becomes a row while the rest become columns. Formally, for an n -order tensor $X \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_n}$, $I_j \in \mathbb{Z}^+$, a mode- k unfolding yields a matrix $X_{(k)} \in \mathbb{R}^{(I_k \times \prod_{j \neq k} I_j)}$. Unfolding across each mode yields different memory access characteristics. For this benchmark, we apply an unfolding operation and then perform a Hadamard product using the resulting matrix. We utilize a $32 \times 32 \times 32 \times 32$ tensor.

Vol2Col is a generalization of **Im2Col** to four dimensions, where a 4D tensor is lowered into a matrix suitable for applying the 3D convolution kernel via matrix multiplication. In this case, the output matrix scales in size cubically with the kernel size. For this benchmark, we apply **Vol2Col** to an input tensor and then perform the GEMM convolution. We fix the volume size to $3 \times 16 \times 128 \times 128$ and the number of output channels to one. We utilize the PyTorch **Vol2Col** implementation [38].

7.3 Effect on Execution Time and WSS

In this subsection, we evaluate DTU’s effects on performance and memory consumption.

Figure 5a shows the normalized execution time results for the *Database Column Projection* benchmark. In a row-wise layout (Row), only a fraction of each fetched cache line contains useful data, as unwanted attributes are interleaved with the target fields, resulting in suboptimal performance. In contrast, a columnar layout (Col) stores each attribute contiguously, allowing accesses to retrieve only the relevant data and thereby improving spatial locality and reducing unnecessary memory traffic, resulting in measurably better performance.

The DTU, on the other hand, provides a contiguous view of the selected attributes even when the underlying table is stored in a row-wise layout, outperforming both Row and Col. While both the DTU and a columnar layout achieve ideal spatial locality for individual columns, the DTU outperforms the columnar approach when projecting many attributes. This advantage arises because the DTU interleaves the selected columns into a single contiguous stream, enabling even simple prefetchers to be effective [52]. In contrast, the columnar layout generates one memory stream per projected column; as the number of projected columns increases, this can exceed the stream-tracking capacity of the hardware prefetcher, reducing its effectiveness.

Figures 5b through 5e show the impact on execution time and working-set size (WSS) of employing our DTU in the four data layout transformation benchmarks. Here, we break down the execution time between the data layout transformation (*Transform*) and subsequent computation (*Compute*). However, in the DTU experiments, the reported execution time only reflects pure computation because the transformation is performed by the DTU hardware on the fly. Comparing the *Compute* portions therefore isolates any runtime overhead incurred when consuming a DTU-provided layout from the performance gains obtained by eliminating explicit transformation.

Note that *Im2Col*, *Image Augmentation*, *Tensor Unfolding*, and *Vol2Col* are all operations that require materialization into a new view, which in the CPU baseline results in substantial data movement and inflates the WSS. The DTU is able to completely eliminate the materialization operation, removing the need for intermediate allocations, drastically

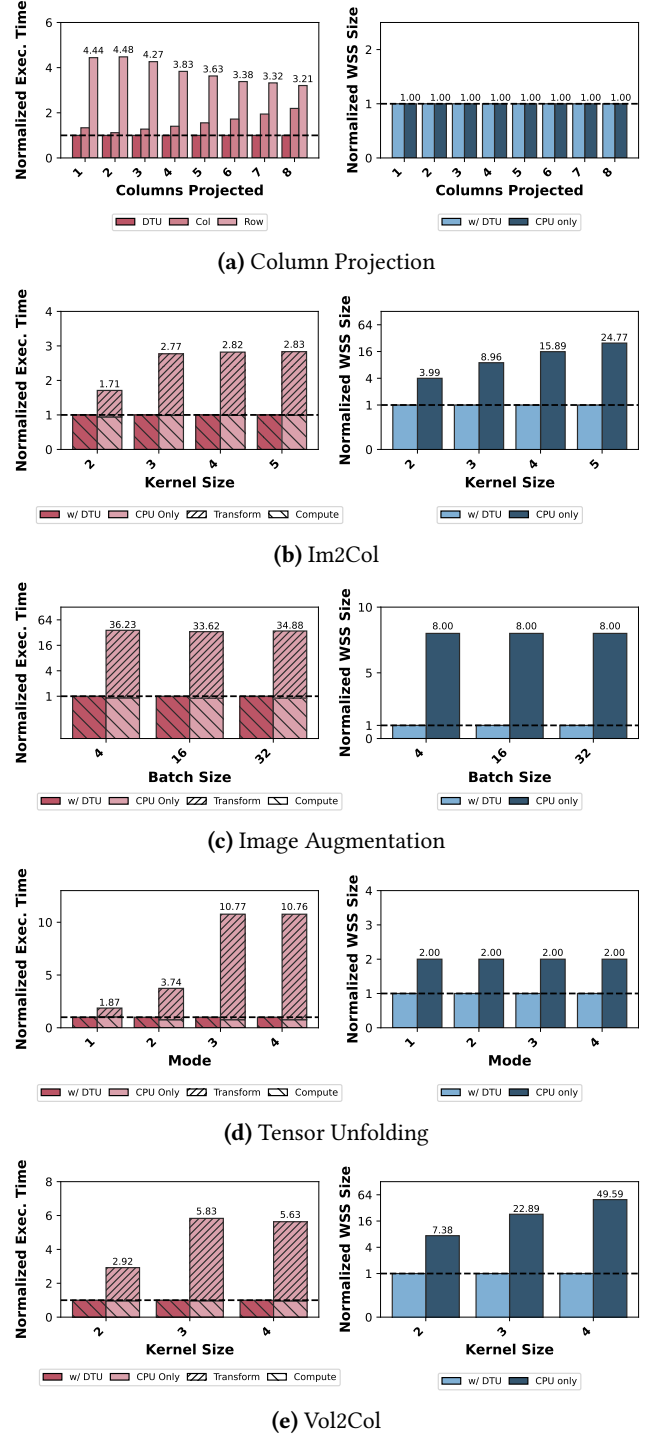


Figure 5. Effects on execution time and WSS.

reducing WSS. In *Im2Col* and *Vol2Col* in particular, the DTU effectively provides a transformed view that is larger than the base data object—up to $\sim 50\times$ in *Vol2Col* with kernel size four—without requiring explicit materialization in memory.

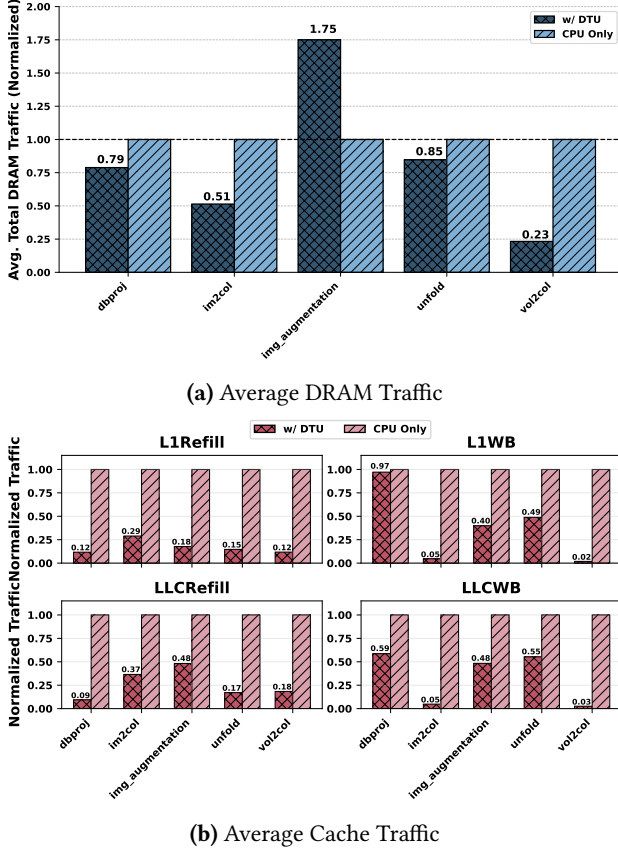


Figure 6. Effects on the memory hierarchy.

In these benchmarks, it is important to note that the execution time reduction achieved by the DTU primarily comes from the elimination of the explicit data layout transformation cost, while the actual computation cost remains largely unchanged (as low as 1.5% overhead for Im2Col).

7.4 Effect on Memory Hierarchy

We evaluate the impact that the DTU has on the memory hierarchy in terms of cache refills, write-backs, and main memory bandwidth usage.

Figure 6 shows (a) the average usage of main memory (DRAM) bandwidth and (b) the cache statistics for L1 and LLC, with and without using DTU. Here, the results of each benchmark are averaged across all evaluated parameter configurations (referring to the x -axis in Figure 5) to display a single data point for each benchmark. For *Database Column Projection (dbproj)*, we use the *Row* configuration in Figure 5a as the “CPU Only” baseline.

The DTU’s impact on DRAM traffic varies by benchmark; while the *img_aug* benchmark exhibits increased traffic, all of the others show a significant reduction. Increased DRAM traffic occurs in the *img_aug* case because the DTU may generate multiple DRAM requests for elements that already lie within a single cache-line sized transfer unit. This effect is

not fundamental, and could be mitigated with more aggressive request coalescing between DTU-generated addresses, or via reuse of fetched data across related accesses.

Importantly, the DTU consistently reduces both cache misses and write-back traffic (at L1 and LLC) across all benchmarks. This improvement stems from presenting the application logic with densely packed cache lines that exhibit ideal data locality and eliminate intermediate materialization, thus increasing cache efficiency and overall performance.

7.5 Implementation Overhead

In this subsection, we report the hardware implementation overhead of the DTU. To quantify the implementation cost of our design, we synthesize a quad-core BOOM SoC using Cadence Genus with Hammer automation [30], targeting the ASAP 7nm [16] technology node. We measure area and timing overheads for a variable number of parallel DTU configurations (Requestors).

Table 2. Area overhead of various DTU configuration counts relative to the quad-core SoC.

Configurations	1	2	4	8
Area (% of SoC)	0.45%	0.85%	1.47%	2.79%

Table 2 shows the area overhead for four DTU variants. As the configuration count increases, the area overhead ranges from 0.45% for a single context to 2.79% for a DTU provisioned with 8 parallel configurations. The 8-configuration variant was used for the results presented in our evaluation. We include smaller configuration counts to show that the DTU can readily be parameterized for different deployment targets and workload needs. All but one of our evaluated workloads utilize only one configuration. The exception is image augmentation which uses seven configurations to export seven different transformed views simultaneously. The configuration count does not affect performance, but restricts the number of transformation views that can be simultaneously exported. Post-synthesis timing analysis reveals that the DTU meets timing at frequencies up to 1.11 GHz for all configuration counts.

8 Related Work

Several others have explored dynamic data reorganization within the memory hierarchy to better match application access patterns. Early efforts such as GS-DRAM [47] and Impulse [12] focused on supporting gather/scatter-style accesses at the memory controller level. The RME [36, 41, 52] applied this idea to database systems, while the TME [22] focused on dense line packing for tensor objects. PLIM [42] and CAESAR [40] investigate systems level integration of these systems into SoCs that include a tightly coupled FPGA. These prior methods were structurally restrictive in the range of

expressible access patterns and constrained to a particular application class. In contrast, our approach puts forward a framework for general cache line construction not tied to a specific domain.

Systems such as *tākō* [46] and *Leviathan* [45] extend the cache hierarchy with programmable, near-data execution mechanisms that can react to memory events. In contrast, our approach exposes locality and flexible data views through compiler-directed mappings, rather than relying on programmable execution within the memory hierarchy.

LEGO [53], Linear Layouts [57], and Axe [23] introduce abstractions for representing tensor layouts that enable compilers to optimize data movement and mapping onto GPU hardware. NVIDIA's CUTLASS [35] provides a library of highly optimized GPU kernels that embody such layout and scheduling strategies. In contrast, our approach is processing-element agnostic, operates at memory-block granularity, and does not require active computation on the processing element to realize data transformations.

9 Future Work

There are many optimizations that deserve to be explored both at the hardware and compiler level. Including even more aggressive compiler optimizations could further decrease needed hardware resources, total cycle latency, and area overhead. The loop based structure of DTL makes affine kernels conducive to analysis with the polyhedral model [44]. Building on this structure, a compiler could identify access patterns expressible as DTL transformations, synthesize corresponding DTL programs, and use a cost model to determine when offloading the transformation is beneficial [24].

There are also multiple avenues to integrate our framework into existing technologies such as MLIR [29], where we envision DTU being used to provide materialization-free transformations into existing tensor framework cost models. Another is deeper integration into RISC-V, particularly targeting memory management methods to provide more flexibility to DTU users. We also believe that the flexibility of the DTU interface makes it a strong candidate for tight integration into a database query optimizer, where it could enable more effective optimization decisions.

Finally, while this work focuses on memory-centric data transformation, integrating DTU with compute-centric accelerators such as Gemmini [19] is an interesting direction for future work. We believe DTU can complement such architectures by reducing data movement overheads and enabling more efficient data layouts prior to computation.

10 Conclusion

This paper introduced DTL and DTU: a hardware–software co-design framework for generalized, on-the-fly data transformation. By enabling programmers to describe complex layout reorganizations in DTL, the DTU transparently bridges

the gap between logical data structures and physical memory layouts. Our end-to-end evaluation on a RISC-V SoC shows that the DTU achieves substantial gains in execution speed and memory efficiency without requiring extensive modifications to computation kernels. Ultimately, this approach effectively decouples data layout from computation, enabling optimal access patterns while preserving software simplicity.

Acknowledgments

This research is supported in part by NSF CCF-2403013 and NSF CCF-2403012.

References

- [1] 2024. SiFive Inclusive Cache. <https://github.com/chipsalliance/rocket-chip-inclusive-cache/tree/main>. Accessed: 2024-05-09.
- [2] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280.
- [3] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 967–980.
- [4] Sam Ainsworth and Timothy M. Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads (ASPLOS '18). Association for Computing Machinery, New York, NY, USA, 578–592. doi:10.1145/3173162.3173189
- [5] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* (July 2020), 10–21.
- [6] M. Annavaram, J.M. Patel, and E.S. Davidson. 2001. Data prefetching by dependence graph precomputation. In *Proceedings 28th Annual International Symposium on Computer Architecture*. 52–61. doi:10.1109/ISCA.2001.937432
- [7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 583–598. <https://www.cs.cmu.edu/~jarulraj/papers/2016.tile.sigmod.pdf>
- [8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language (DAC '12). 1216–1225.
- [9] Berkeley Architecture Research preFetchers [n. d.]. Berkeley Architecture Research preFetchers. <https://github.com/ucb-bar/bar-fetchers>.
- [10] Peter Boncz. 2002. *Monet: A next-Generation DBMS Kernel For Query-Intensive Application*. Ph.D. Dissertation. <http://dare.uva.nl/document/64315>
- [11] Buildroot Team. 2026. Buildroot - Making Embedded Linux Easy. <https://buildroot.org/>. Accessed: 2026-04-11.
- [12] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. 1999. Impulse: building a smarter memory controller. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. 70–79. doi:10.1109/HPCA.1999.744334
- [13] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive,*

- Synthesizable, Parameterized RISC-V Processor*. Technical Report UCB/EECS-2015-167. University of California-Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECSz2015-167.html>
- [14] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft.
 - [15] 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>
 - [16] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandrasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016). doi:10.1016/j.mejo.2016.04.006
 - [17] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 82–93. doi:10.1145/996841.996853
 - [18] Babak Falsafi and Thomas F. Wenisch. 2014. A Primer on Hardware Prefetching. In *A Primer on Hardware Prefetching*. <https://api.semanticscholar.org/CorpusID:28158752>
 - [19] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2022. *Gemini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration*. IEEE Press, 769–774.
 - [20] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (2024), 33–39. doi:10.1109/MM.2024.3373763
 - [21] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (2023). <https://api.semanticscholar.org/CorpusID:257633012>
 - [22] Denis Hoornaert, Cole Strickler, Manos Athanassoulis, Marco Caccamo, Heechul Yun, and Renato Mancuso. 2026. Tensor Memory Engine: On-the-fly Data Transformation for High-Performance Computing. arXiv:2604.13319 <https://arxiv.org/abs/2604.13319>
 - [23] Bohan Hou, Hongyi Jin, Guanjie Wang, Jinqi Chen, Yaxing Cai, Lijie Yang, Zihao Ye, Yaoyao Ding, Ruihang Lai, and Tianqi Chen. 2026. Axe: A Simple Unified Layout Abstraction for Machine Learning Compilers. arXiv:2601.19092 [cs.DC] <https://arxiv.org/abs/2601.19092>
 - [24] Xianglong Huang, Zhenlin Wang, and K.S. McKinley. 2001. Compiling for the Impulse memory controller. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. 141–150. doi:10.1109/PACT.2001.953295
 - [25] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. 2004. Effective stream-based and execution-based data prefetching (ICS '04). Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/1006209.1006211
 - [26] Redmon Joseph and D Santosh. 2016. YOLO: real-time object detection. Available: <https://pjreddie.com/darknet/yolov1/>, [Accessed: 5 January 2018] (2016).
 - [27] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 364–373.
 - [28] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 29–42. <https://doi.org/10.1109/ISCA.2018.00014>
 - [29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308
 - [30] Harrison Liew, Daniel Grubb, John Wright, Colin Schmidt, Nayiri Krzysztofowicz, Adam Izraelevitz, Edward Wang, Krste Asanović, Jonathan Bachrach, and Borivoje Nikolić. 2022. Hammer: a modular and reusable physical design flow tool: invited. In *DAC (San Francisco, California)*. doi:10.1145/3489517.3530672
 - [31] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1025–1040. <https://www.usenix.org/conference/atc19/presentation/liu-yizhi>
 - [32] Albert Magyar, David Biancolin, John Koenig, Sanjit A. Seshia, Jonathan Bachrach, and Krste Asanovic. 2019. Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 1–8.
 - [33] Sally A. McKee. 2004. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers (Ischia, Italy) (CF '04)*. Association for Computing Machinery, New York, NY, USA, 162. doi:10.1145/977091.977115
 - [34] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, Rachata Ausavarungnirun, Mohammad Sadrosadati, and Geraldo F. Oliveira. 2025. A Modern Primer on Processing in Memory. arXiv:2012.03112 [cs.AR] <https://arxiv.org/abs/2012.03112>
 - [35] NVIDIA. 2025. CUTLASS: Principled Abstractions for Handling Multidimensional Data Through Tensors and Spatial Microkernels. <https://developer.nvidia.com/blog/cutlass-principled-abstractions-for-handling-multidimensional-data-through-tensors-and-spatial-microkernels/>. Accessed: 2026-04-11.
 - [36] Tarikul Islam Papon, Ju Hyoun Mun, Konstantinos Karatsenidis, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2024. Effortless Locality on Data Systems Using Relational Fabric. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (2024), 7410–7422. doi:10.1109/TKDE.2024.3386827
 - [37] Udayan S. Patankar and Ants Koel. 2021. Review of Basic Classes of Dividers Based on Division Algorithm. *IEEE Access* 9 (2021), 23035–23069. doi:10.1109/ACCESS.2021.3055735
 - [38] PyTorch Team. 2026. PyTorch vol2col implementation (source code). <https://github.com/pytorch/pytorch/blob/main/aten/src/Aten/native/vol2col.h>. Accessed: 2026-04-11.
 - [39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines (PLDI '13). Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176
 - [40] Shahin Roozkhosh, Denis Hoornaert, and Renato Mancuso. 2022. CAESAR: Coherence-Aided Elective and Seamless Alternative Routing via on-chip FPGA. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*. 356–369. <https://doi.org/10.1109/RTSS55097.2022.00038>
 - [41] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoun Mun, Tarikul Islam Papon, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Memory: Native In-Memory Accesses

- on Rows and Columns. In *26th International Conference on Extending Database Technology (EDBT)*.
- [42] Shahin Roozkhosh and Renato Mancuso. 2020. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 296–309. doi:10.1109/RTAS48715.2020.00006
- [43] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.
- [44] P. Sadayappan and Uday Bondhugula. 2008. Effective automatic parallelization and locality optimization using the polyhedral model. <https://api.semanticscholar.org/CorpusID:9516753>
- [45] Brian C. Schwedock and Nathan Beckmann. 2024. Leviathan: A Unified System for General-Purpose Near-Data Computing. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1278–1294. doi:10.1109/MICRO61859.2024.00095
- [46] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. 2022. *tākō*: a polymorphic cache hierarchy for general-purpose optimization of data movement. *Proceedings of the 49th Annual International Symposium on Computer Architecture (2022)*. <https://api.semanticscholar.org/CorpusID:249205275>
- [47] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A. Kozuch, and Todd C Mowry. 2015. Gather-Scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 267–280. doi:10.1145/2830772.2830820
- [48] SiFive. 2017. SiFive TileLink Specification. Accessed: 2024-05-10.
- [49] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2019. Near-memory computing: Past, present, and future. *Microprocess. Microsyst.* 71, C (Nov. 2019), 15 pages. doi:10.1016/j.micpro.2019.102868
- [50] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. *The VLDB Journal* 33, 5 (2024), 1485–1515. <https://doi.org/10.1007/s00778-024-00858-9>
- [51] Paul Springer, Tong Su, and Paolo Bientinesi. 2017. HPTT: a high-performance tensor transposition C++ library. In *Proceedings of the ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY@PLDI)*. 56–62. doi:10.1145/3091966.3091968
- [52] Cole Strickler, Ju Hyoung Mun, Connor Sullivan, Denis Hoornaert, Renato Mancuso, Manos Athanassoulis, and Heechul Yun. 2025. Micro-architectural Exploration of the Relational Memory Engine (RME) on RISC-V SoC using FireSim. <https://www.vldb.org/2025/Workshops/VLDB-Workshops-2025/ADMS/ADMS25-08.pdf>. *Proceedings of the VLDB Endowment*. ISSN 2150 (2025).
- [53] Amir Mohammad Tavakkoli, Cosmin E. Oancea, and Mary Hall. 2026. LEGO: A Layout Expression Language for Code Generation of Hierarchical Mapping. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 228–241. doi:10.1109/CGO68049.2026.11394846
- [54] Henry S. Warren. 2012. *Hacker’s Delight* (2nd ed.). Addison-Wesley Professional.
- [55] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2016. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*. Technical Report UCB/EECS-2016-118. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html>
- [56] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.
- [57] Keren Zhou, Mario Lezcano-Casado, Adam P. Goucher, Akhmed Rakhmati, Jeff Niu, Justin Lebar, Pawel Szczerbuk, Peter Bell, Phil Tillet, Thomas Raoux, and Zahi Moudallal. 2025. Linear Layouts: Robust Code Generation of Efficient Tensor Computation Using $\mathbb{F}_2$. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (Pittsburgh, PA, USA) (ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 132–146. doi:10.1145/3760250.3762221
- [58] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1771–1775. <http://doi.acm.org/10.1145/3035918.3054784>

A DTL Grammar Specification

This section presents the grammar specification of the Data Transformation Language (DTL), defining the syntax of valid kernels.

Listing 3. DTL Grammar Specification

```

1 <program> ::= <constdecls> <forstatement>
2
3 <constdecls> ::= <constdecls> <constdecl> | epsilon
4
5 <constdecl> ::= <opt> <type> ID ASSIGN INTLITERAL SEMICOL
6 | <opt> <type> ID ASSIGN LCURLY <intlist> RCURLY
   SEMICOL
7
8 <opt> ::= NOPT | epsilon
9
10 <intlist> ::= INTLITERAL COMMA <intlist>
11 | INTLITERAL
12 | epsilon
13
14 <forstatement> ::= FOR LPAREN <constdecl> <id> LT <expr> SEMICOL
15 | <unarystmt> RPAREN LCURLY <forstatement>
16 | RCURLY
17 | FOR LPAREN <constdecl> <id> LT <expr>
18 SEMICOL <unarystmt> RPAREN LCURLY <innermost>
19 RCURLY
20
21 <innermost> ::= <outstatements>
22 | <ifstatement>
23 | <switchstatement>
24
25 <outstatements> ::= <outstatements> <outstatement>
26 | <outstatement>
27
28 <ifstatement> ::= IF LPAREN <ifpredicate> RPAREN LCURLY
29 <outstatements> RCURLY ELSE LCURLY
30 <outstatements> RCURLY
31
32 <ifpredicate> ::= <id> <compareop> <id>
33 | ISEVEN id
34 | PAD <id> <id> <id> <id>
35 | ISEGE <id>
36 | ISEGE <id> OR <id>
37 | ISEGE <id> AND <id>
38
39 <compareop> ::= LT | LTE | GT | GTE
40 <boolop> ::= AND | OR
41
42 <switchstatement> ::= SWITCH LPAREN <id> RPAREN LCURLY <
   casestatemnts> RCURLY
43
44 <casestatemnts> ::= <casestatement> <casestatemnts>
45 | <casestatement>
46
47 <casestatement> ::= CASE COLON <outstatements>
48
49 <outstatement> ::= OUT ASSIGN <expr> SEMICOL
50
51 <type> ::= INT
52
53 <expr> ::= <expr> <exprop> <expr>
54 | <term>
55
56 <exprop> ::= CROSS | STAR | MINUS | LT
57
58 <term> ::= <factor>
59 | LPAREN <expr> RPAREN
60
61 <factor> ::= INTLITERAL
62 | <loc>
63
64 <loc> ::= <id>
65 | <id> LBRACKET <id> RBRACKET
66
67 <id> ::= ID

```

Listing 4. Example DTL Kernel Performing Tensor Slicing over the inner two dimensions

```

1 int D4 = 64;
2 int D3 = 64;
3 int D2 = 64;
4 int D1 = 64;
5 int stride_d2 = D1;
6 int stride_d3 = D2*stride_d2;
7 int stride_d4 = D3*stride_d3;
8 int sel_factor_d1 = 8;
9 int sel_factor_d2 = 2;
10 int sel_factor_d3 = 1;
11 int sel_factor_d4 = 1;
12
13 // bounds equal to D4/sel_factor_d4
14 for (int i = 0; i < D4; i++)
15 {
16     // bounds equal to D3/sel_factor_d3
17     for (int j = 0; j < D3; j++)
18     {
19         // bounds equal to D2/sel_factor_d2
20         for (int k = 0; k < D2; k++)
21         {
22             // bounds equal to D1/sel_factor_d1
23             for (int l = 0; l < D1; l++)
24             {
25                 out = l*(sel_factor_d1) + j*(stride_d2*
26                     sel_factor_d2) + k*(stride_d3*
27                     sel_factor_d3) + i*(stride_d4*
28                     sel_factor_d4);
29             }
30         }
31     }
32 }

```

Listing 5. Example DTL Kernel Performing a 3 Column Projection

```

1 int col_offsets = {4, 32, 48};
2 int row_size = 64;
3 int ROW_COUNT = 500000;
4
5 for (int i = 0; i < ROW_COUNT; i++)
6 {
7     for (int j = 0; j < 3; j++)
8     {
9         out = i*row_size + col_offsets[j];
10     }
11 }

```

Listing 6. Example DTL Kernel Performing Vol2Col

```

1 int depth = 16;
2 int height = 128;
3 int width = 128;
4 int channels = 3;
5 int KSIZE = 3;
6
7 for (int c = 0; c < channels; c++) {
8     for (int kt = 0; kt < KSIZE; kt++) {
9         for (int kh = 0; kh < KSIZE; kh++) {
10             for (int kw = 0; kw < KSIZE; kw++) {
11                 for (int t = 0; t < depth-KSIZE+1; t++) {
12                     for (int h = 0; h < height-KSIZE+1; h++) {
13                         for (int w = 0; w < width-KSIZE+1; w++) {
14                             {
15                                 out = (c*(depth*height*width) + w +
16                                     kw)
17                                     + ((t + kt)*(height*width)
18                                     + (h + kh)*width);
19                             }
20                         }
21                     }
22                 }
23             }
24         }
25     }
26 }

```

B More DTL Kernel Examples

In this section, we present further examples of DTL kernels to complement the earlier entries.

C Artifact Appendix

C.1 Abstract

This artifact provides the source code associated with the DTU system presented in this paper. It includes the modified FireSim/Chippyard system integration, DTU and AGU RTL implementations, the Data Transformation Language (DTL) implementation, and the benchmark suite used in the paper.

- **Program:** DTL benchmark suite and FireSim-based evaluation environment.
- **Compilation:** RISC-V software toolchain, FireSim/Chippyard hardware build flow.
- **Run-time environment:** FireSim/Chippyard-based RISC-V SoC simulation environment.
- **Software:** Ubuntu 20.04-based host system, Xilinx Vivado
- **Hardware:** Xilinx VCU118 FPGA
- **Experiments:** Set up the Firesim environment, build the bitstream with the DTU-enabled config, compile and set up the software benchmark suite, run the benchmark suite in Firesim on the VCU118, and collect output logs in CSV format over the Firesim serial port.
- **How much disk space required (approximately)?:** At least 300 GB
- **How much time is needed to prepare workflow (approximately)?:** 4-9 hours total. Firesim setup (2-3 hours). Bitstream compilation (2-6 hours).
- **How much time is needed to complete experiments (approximately)?:** 6 hours
- **Publicly Available?:** Yes.

C.2 Description

C.2.1 How to access. All source code associated with the artifact is publicly available on GitHub. The artifact consists of a modified Firesim repository containing the system-level integration of the DTU, separate repositories containing the DTU and AGU RTL implementations, the DTL language implementation, and the DTL benchmark suite. Repository links are provided at:

- FireSim Fork
- Data Transformation Unit
- Data Transformation Language
- Benchmark Suite

For archival access and reproducibility, we also provide the following Zenodo DOIs:

- DTL Language/Compiler
- Data Transformation Unit
- Address Generation Unit
- Benchmark Suite

C.2.2 Hardware dependencies. Full-system evaluation requires a Xilinx VCU118 FPGA and a Linux-based host system. At least 300 GB of available disk space is recommended for the Firesim/Chippyard environment, FPGA build products, and benchmark artifacts.

C.2.3 Software dependencies. Firesim v1.18.0 is officially supported on Ubuntu 20.04. The evaluation environment additionally requires the Firesim/Chippyard software stack, a RISC-V cross-compilation toolchain, and Xilinx Vivado for FPGA bitstream generation.

C.3 Installation

The artifact repositories contain installation and setup instructions for each component. Full-system evaluation requires installation of Firesim v1.18.0 and Chippyard on an Ubuntu 20.04 host, installation of the DTL toolchain and benchmark suite, and configuration of Xilinx Vivado for FPGA bitstream generation. Detailed installation commands, required software versions, and repository revisions are provided in the corresponding artifact repositories. A complete Firesim environment setup typically requires approximately 2–3 hours, excluding FPGA bitstream generation.