Hardware Abstractions for targeting EDDO Architectures with the Polyhedral Model

Angshuman Parashar
NVIDIA
aparashar@nvidia.com

Prasanth Chatarasi*
IBM Research
prasanth@ibm.com

Po-An Tsai
NVIDIA
poant@nvidia.com

Abstract

Unlike cache-based load-store architectures, Explicit Decoupled Data Orchestration (EDDO) architectures are programmed using decoupled but synchronized programs running at various units on the hardware, moving data between storage units and/or performing computations. As such, they present a unique programming challenge.

In this paper, we propose a set of hardware abstractions to represent EDDO architectures, enabling them to be targeted with the polyhedral model for statically analyzable workloads. The abstractions are rich enough to support EDDO architectures with arbitrarily deep storage hierarchies, hierarchical parallelism and support for temporal and spatial reuse exploitation via multicast, peer-to-peer communications, and spatial reduction. We also frame the abstractions within the context of an in-progress project called PolyEDDO, which is a mapping analysis and code-generation framework for EDDO architectures.

1 Introduction

The energy and performance costs of moving data across and within chips are becoming increasingly problematic [24]. Explicit Decoupled Data Orchestration (EDDO) architectures alleviate some of these costs by provisioning a chip with a distributed set of efficient state machines that stage and move data between memory and compute units across the chip [5, 6, 9, 10, 13, 14, 18, 20, 25]. EDDO architectures can achieve superior efficiency to traditional cache-based (or Implicit-Coupled) architectures on a range of valuable algorithm domains [23]. Efficiency is achieved due to several factors:

- Dedicated (and often statically programmed) state machines for memory management, address generation, network transfers and computation are more efficient at their respective tasks than more general homogeneous engines.
- Pipelined, decoupled address generation leads to decreased load-to-use latencies, reducing buffer landing zone requirements, contracting register file/buffer sizes, and thereby reducing their energy profile.
- Stylized storage idioms such as buffets [23] are more efficient than either caches or double-buffered scratchpad memories.
- The architectures are built with data reuse in mind, with deep memory hierarchies and widgets to exploit various forms of spatial reuse such as peer forwarding, multicast, and spatial reduction [17].

Many of these features draw inspiration from desirable attributes of fixed-function ASICs, which are distilled and generalized into more flexible forms in EDDO architectures to service the needs of a larger domain of applications. However, these capabilities come with a price—the added complexity of programming an EDDO architecture, both for human programmers as well as for automated mapping tools. The challenge arises due to the following reasons:

- Unlike traditional architectures, there is no single program, not even a homogeneous set of programs. Instead, a set of distributed, heterogeneous programs must be written to perform computations and to copy, distribute and collect/merge data between multiple sources and destinations. These programs must synchronize and work in concert to execute the algorithm and its mapping. This complexity is exacerbated by the fact that EDDO architectures have deep, interconnected storage hierarchies with varying degrees of parallelism.
- Reuse analysis (accounting for local temporal reuse, peer forwarding, parent multicast, spatial reductions) is a critical component of compilation flow.
- Estimating execution time and energy efficiency is critical for cost modeling. Simplistic cost models such as the number of DRAM accesses do not work [8, 22]; on-chip buffer access and network transfer costs must be accounted for as well.

An even broader challenge is that there is no single EDDO architecture. There are various architectures with different hardware topologies, varied composition of hardware elements, and varying degrees of programmability. Furthermore, they are rapidly evolving, making it unappealing to build a single toolchain targeted at a specific architecture.

To address these challenges, we present a flexible hardware abstraction for EDDO architectures. This abstraction, which we call *Hardware Space-Time* (or *HST*), can describe the topologies of a variety of EDDO architectures with user-specifiable, arbitrarily deep memory hierarchies and nested physical parallelism, along with non-hierarchical units if they exist, using the power of the *polyhedral model*. Given an

*This work was conducted while this author was a graduate student at Georgia Tech.

algorithm mapping, this abstraction symbolically captures the runtime behavior of hardware units in the architecture across space and time.

Although there exist prior efforts that use the polyhedral model [2, 12, 27] to infer the runtime behavior of memory units from program execution order/schedules or compute units from space-time mapping of iteration points [7], HST *explicitly* models the runtime behavior of all memory and compute units, allowing for more straightforward analysis and code generation for EDDO architectures. Our approach is complementary to existing abstractions for capturing program execution orders such as multi-dimensional timestamps [11, 15] and schedule trees [28] because HST requires a mapping (program execution order) as an input to model the hardware runtime behavior. To the best of our knowledge, none of the prior works explicitly model and analyze the behavior of all hardware units across space and time using the polyhedral model in a way that HST does.

The HST abstraction enables a range of EDDO architectures to be targeted with a universal analysis/code-generation framework. We have been working on a framework called *PolyEDDO*, which leverages the polyhedral model's power for precise reasoning at compile-time. Given an HST representation of an EDDO architecture, a workload specification, and a mapping of the workload onto the hardware, PolyEDDO analyzes the behavior of the mapping over the *space* of hardware units across the architecture and the *time* of execution of the algorithm. This analysis is used to (a) generate activity counts that can be fed into a performance/energy cost model and (b) generate decoupled, distributed codeblocks representing data transfer and computation activities that cooperatively execute the logic of the workload. Our longer-term objective is to integrate PolyEDDO with an optimizer (or *mapper*) for EDDO architectures. The mapper would take a user-provided *unmapped* algorithm specification and a description of the architecture and generate optimized *mapped* program binaries for that architecture.

The primary focus of this paper is the HST abstraction itself. We complement this with a high-level overview of our work-in-progress PolyEDDO framework to concretize the value of the abstraction.

2 Background

This section gives a brief overview of the background material on EDDO architectures and workload domains that suit them. We expect readers to have a fair background on the polyhedral model, especially affine sets and maps.

2.1 EDDO architectures

Explicit Decoupled Data Orchestration (or EDDO) architectures represent a class of domain-specific architectures. Many recent domain-specific accelerators, such as Eyeriss [5, 6], NVDLA [20], Rapid AI [9], SIMBA [25], Morph [13],

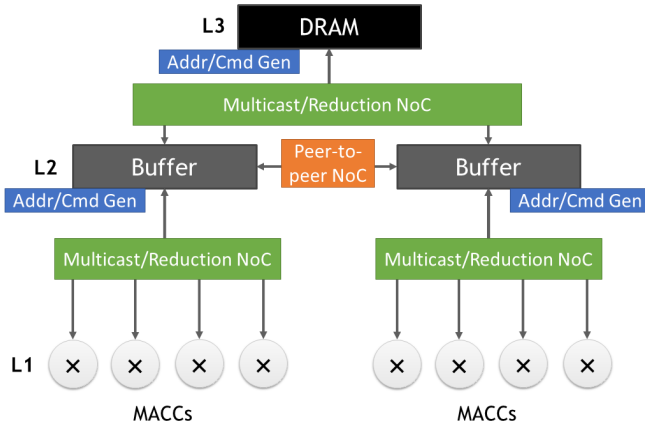


Figure 1. Example EDDO architecture. Programmable units include MACC units, address and command generators at each memory level, multicast and reduction NoC connecting hardware units in different levels, and peer-to-peer NoC connecting hardware units in the same level.

MAERI [18], and Extensor [14], are examples of EDDO architectures. Early instances started out being fairly fixed-function, but the general trend has been an evolution towards increasing programmability while retaining some of the goodness of fixed-function ASICs. Unlike traditional general-purpose architectures (CPUs and GPUs), EDDO architectures specialize in a particular domain and use explicit, decoupled data orchestration to optimize the data movement and synchronization between parallel and spatially-distributed hardware units. Figure 1 shows an example 3-level EDDO architecture for dense tensor algebra. In this example, L1 denotes compute units (multiply-and-accumulate units, MACCs), while L2 and L3 are buffers and main memory (DRAM). There are also dedicated and statically-programmed state machines for storage/memory management (command generation), address generation, and network transfers (NoCs).

Explicit, decoupled data movement. Programmers *explicitly* and precisely control when and where data is used via programming distributed state machines. Data is explicitly moved between source and destination locations, which include main memory, on-chip buffers and compute elements. When data reuse patterns are statically known, explicit data movement is always more efficient as it avoids hardware overheads such as cache lookups in implicit data orchestration, as demonstrated by many current DNN accelerators.

EDDO architectures also perform *decoupled* data movement: at each hardware level, the pipelined, decoupled address and command generators independently push data to other levels. This decoupled but synchronized data movement leads to decreased load-to-use latencies, reducing buffer landing zone requirements, contracting register file/buffer sizes, and thereby reducing their energy profile.

Programming (configuring) these distributed state machines is the *code generation* for EDDO hardware.

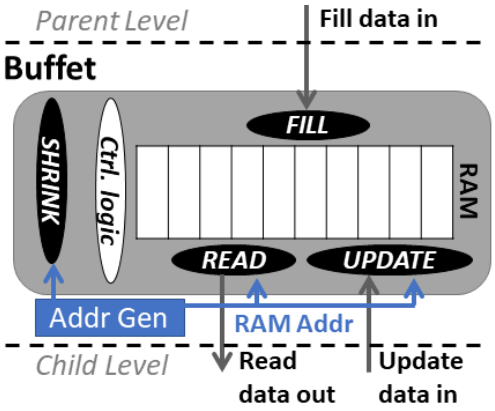


Figure 2. Buffet storage idiom (adopted from [23]).

Synchronization. Timing and synchronization of data transfers is critical for functional correctness, as an early transfer could overwrite live data, and a late transfer results in efficiency loss. Moreover, delivering data in incorrect order or skipping can potentially lead to a deadlock between distributed units.

To address these issues, our target EDDO architectures leverage *buffets [23]*, a recently proposed storage idiom, to provide the data-staging and synchronization primitives. Buffets allow efficient and decoupled data `fills`, `reads`, and `updates` with fine-grained synchronization. Buffets are more efficient than either caches or double-buffered scratchpads [23], and with random-access/update support, they are also more flexible than simple hardware FIFOs.

Figure 2 shows the operations supported by buffets. Hardware units at a parent level (e.g., main memory) can `fill` fresh data into a buffet. Hardware units at a child level (e.g., a datapath or an inner-level buffet) can `read` or `update` any *live* data elements in the buffet. The buffet is addressed relative to the base of the live window. [1] When data is determined dead (not useful anymore), a client can invoke a `shrink` operation to advance the live window (and also clear space for future `fills`). The control logic inside the buffet manages translations from the user-visible address space to the local RAM addresses, and also performs synchronization between the various operations. For example, it stalls a `read` operation if the data is not yet `filled`.

Data reuse. EDDO architectures include deep memory hierarchies and various network structures such as multi-cast, broadcast, and forwarding to exploit various reuse forms to optimize data movement. In particular, there are three major forms of data reuse: 1) Temporal reuse: reusing data present in a hardware unit by the same consumer multiple times, 2) Spatial reuse: reusing data present in a hardware unit by multiple consumers at the same time using multi-cast/broadcast, 3) Spatial-temporal reuse: reusing data present in a hardware unit by a peer using forwarding links.

[1] EDDO architectures often use decoupled address-generator state machines to generate these addresses.

2.2 Workload domain

Although we expect our contributions presented in this paper to support a broader class of algorithm domains, our present PolyEDDO implementation focuses on EDDO architectures that are designed and specialized for the perfectly nested affine loops in *dense tensor algebra* space, such as GEMM, convolutions, and tensor contractions. These perfectly nested affine loops further lend our framework to using a polyhedral model to analyze, optimize, and generate code for the EDDO architecture.

In future we expect our framework to support imperfectly nested loops and a sequence of affine loop nests as input workloads. Furthermore, while we expect PolyEDDO can also adopt recent work [1, 26] on using the polyhedral model for sparse matrices, we leave such analysis and code generation for EDDO architectures for future work.

2.3 Polyhedral model

The polyhedral model is a powerful algebraic framework that has enabled significant advances to the analysis and transformation of affine programs by precisely capturing the program's dynamic execution order at compile-time. This program execution order can be used to reason about certain aspects of the dynamic behavior of a hardware's compute and memory units. However, conventional architectures such as CPUs, GPUs involve memory units that are non-deterministic concerning loads and stores, for, e.g., cache-based memory systems. Although there are a few prior efforts [2, 12, 27] to reason about their behavior with assumptions, the non-deterministic nature makes it challenging to precisely reason about the dynamic behavior, and this challenge gets exacerbated with multiple memory units and multiple levels. However, all the hardware units in the EDDO architectures are (mostly) statically programmed. This allows a compiler framework to reason about the dynamic execution behavior of both the compute and memory units for an affine program and its mapping at compile-time, which is extremely important to generate high-performance and energy-efficient binaries for all of the programmable units of the EDDO architectures.

3 Hardware Space-Time Abstraction (HST)

In this section, we introduce our abstraction—*Hardware Space-Time (HST)*—to capture runtime behavior of hardware components of EDDO architectures.

From a programming standpoint, we view an EDDO architecture as a collection of *buffets* and *engines*. For instance, consider the architecture in Figure 4(b). With its deep memory hierarchy and criss-crossing data delivery networks, the topology is reminiscent of neural-network accelerators such

as NVDLA [20] and Eyeriss [5]. Every state element (a rectangle in the figure) is a buffet [23]. The DRAM, on-chip buffers, and even the operand/result registers at the MACC units are assumed to provide buffet semantics. The arrows and the MACC units (×) are transfer and compute engines, respectively. Our job is to program each engine, with the buffets serving as source, drain and synchronization endpoints. Henceforth, we use the terms *buffer* and *buffet* interchangeably in this paper.

We tame the complexity of programming this interconnected distributed system by recognizing that in executing a workload, the engines typically execute in a systematic, hierarchical cadence that can be captured in a *space-time* structure. We now describe this structure by starting with simple examples and gradually growing in complexity until we can describe the architecture in Figure 4.

3.1 Symbolic Hierarchical Space-Time (SHST)

The Symbolic Hierarchical Space-Time (SHST) is a hierarchical space symbolically representing the existence of all hardware units over a period of time steps. Intuitively, this space-time can be thought of as a set of "holes" over space and time into which workload operations can be placed. Because we are dealing with EDDO architectures, a programmer/mapper must reason about the workload mapping at each level of the architecture's hierarchy. To support this, the SHST has the same depth (i.e., number of levels) as the hardware hierarchy.

An example: Figure 3(a) shows an example 3-level hardware architecture with a DRAM (L3) serving as backing store for two L2 buffers (level-2 of the hardware). Each buffer services a set of four L1 MAC units (level-1 of the hardware). Figure 3(b) shows a visual depiction of the space-time hierarchy (for a small number of time steps). Observe that:

- The L3 DRAM only has one space *and* time coordinate. The single space coordinate simply means that there is only one DRAM instance. The single time coordinate means that the entire workload is resident at this DRAM instance throughout its execution.
- The L2 level has two space-coordinate values ($s_2 = 0, 1$) corresponding to the two buffer instances, and two time-coordinate values ($t_2 = 0, 1$) in this example. These time coordinates do not represent wall-clock time instances; they represent logical time *steps*. Several t_1 time steps may occur within each t_2 time step.
- Each of the 4 L1 MACs exists in exactly the same number of time coordinates (3 in this example) within each L2 time coordinate. However, this does not mean they are physically lock-stepped in terms of hardware clock cycles – they can slip with respect to each other at a microarchitectural level.

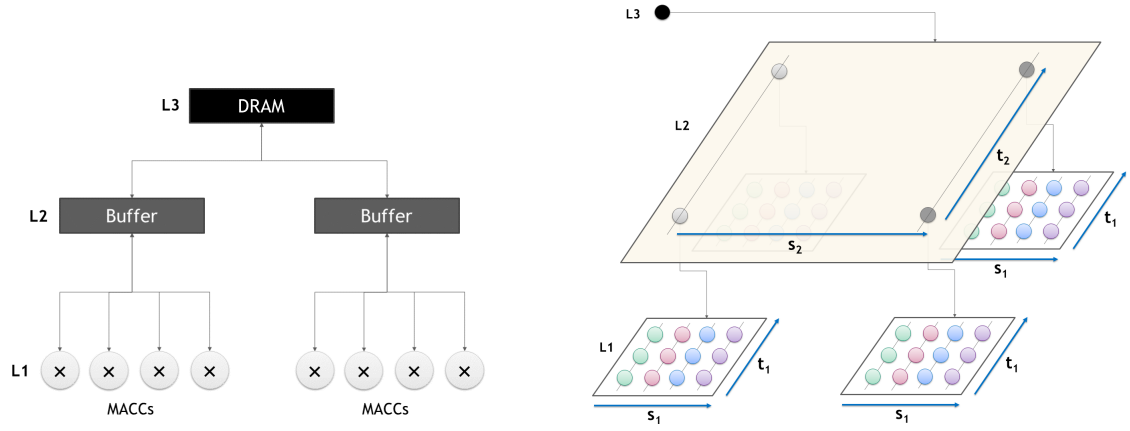
Time coordinates represent logical time *steps* and are not necessarily tied to hardware clock cycles. The presence of a time step provides a slot for mapping a set of workload's iteration-space points. Thus, the passage of time at a hardware unit allows it to hold a different set of iteration-space coordinates from the workload at each time step. If the space-time space at a hardware unit only has one time coordinate $t = 0$ (as in Figure 3), it means that the set of iteration-space points mapped to that unit never changes. This is usually true of the last-level storage unit in a hardware architecture, since it is the source and sink of the entire workload's input and output data sets.

SHST abstraction: The abstraction is a linear chain of *(space, time)* nodes starting from the last level to the first level of the hardware with an intent to symbolically capture the entire space-time hierarchy using the Presburger relations. Each *(space, time)* node corresponds to a level in the hardware, for example, there are three symbolic nodes ($SpaceTime_1(s_1, t_1)$, $SpaceTime_2(s_2, t_2)$, $SpaceTime_3(s_3, t_3)$) corresponding to the 3-levels in the hardware shown in Figure 3. Both the space and time components of a node are multi-dimensional vectors, and the values of a space component don't have an ordering, however, the values of a time component have an ordering and adhere to the classical lexicographic ordering to realize relative ordering among the timestamps.

We define our SHST abstraction for a n-level hardware, i.e., $\Theta_n^{SHST}(\overrightarrow{s_n}, \overrightarrow{t_n})$ recursively as follows:

$$\Theta_i^{SHST}(\overrightarrow{s_i}, \overrightarrow{t_i}) = \; SpaceTime_i(\overrightarrow{s_i}, \overrightarrow{t_i}),\; i = 1,$$
$$SpaceTime_i(\overrightarrow{s_i}, \overrightarrow{t_i}) \rightarrow [\Theta_{i-1}^{SHST}(\overrightarrow{s_{i-1}}, \overrightarrow{t_{i-1}})],\; i > 1$$

where $\overrightarrow{s_i}, \overrightarrow{t_i}$ denote space and time components of the Level-i hardware unit. Our SHST abstraction for a single-level hardware is simply a *set*. For a non-single-level hardware, we represent our SHST abstraction as a *map* with the domain or input set symbolically denoting the space-time vectors of the last level unit in the hardware, and the range or output set recursively encloses the SHST abstraction for the remaining levels in the hardware using a zipped notation available in the ISL framework. For example, the SHST abstraction for the entire 3-level hardware (shown in Figure 3) excluding the Presburger constraints is $SpaceTime_3[s_3, t_3] \rightarrow [SpaceTime_2[s_2, t_2] \rightarrow SpaceTime_1\ [s_1, t_1]]$, where $SpaceTime_3[s_3, t_3]$ corresponds to the space-time vectors of the DRAM and $[SpaceTime_2\ [s_2, t_2] \rightarrow SpaceTime_1[s_1, t_1]]$ corresponds to the SHST abstraction for the remaining L2 and L1 levels of the hardware. Presburger constraints for space are derived from the hardware specification, and those for time are derived from the mapping. Although the hardware examples are uniform and all sub-trees are isomorphic at each level, our abstraction allows users to represent non-uniform hardware topologies if desired (e.g., two L2 buffers



$$\Theta_3^{SHST}(\vec{s_3}, \vec{t_3}) = \{SpaceTime_3[\vec{s_3} = s_3, \vec{t_3} = t_3] \rightarrow [SpaceTime_2[\vec{s_2} = s_2, \vec{t_2} = t_2] \rightarrow SpaceTime_1[\vec{s_1} = s_1, \vec{t_1} = t_1]] : (s_3 = 0 \wedge t_3 = 0) \wedge (0 \leq s_2 < 2 \wedge 0 \leq t_2 < 2) \wedge (0 \leq s_1 < 4 \wedge 0 \leq t_1 < 3)\}$$

(c)

Figure 3. Example showing: (a) 3-level hardware architecture with a DRAM (L3) and a buffer (L2) servicing an array of four MACC units (L1). (b) Hardware space-time space representing this architecture (for a small number of time steps). *Space* coordinates representing the hardware units are shown along the X-axis of the figure. *Time* coordinates representing time steps are shown along the Z-axis (into the plane of this document). Bounds for t_1 and t_2 are shown purely for illustration; actual time bounds are derived from the mapping (Section 4). (c) Symbolically capturing the hardware space-time using the Presburger relations, i.e., Symbolic Hardware Space-Time (SHST).

each with a different number of L1s), as long as the hierarchy can be described in a *piece-wise quasi-affine* manner.

We could have alternatively represented the SHST abstraction as a single *(space, time)* node enclosing space and time vectors of all hardware levels. However, this approach would have required a separate bookkeeping mechanism to delineate between multiple hardware levels. We avoid this bookkeeping by using *zipped* relations in ISL to explicitly separate the space and time vectors of different hardware levels.

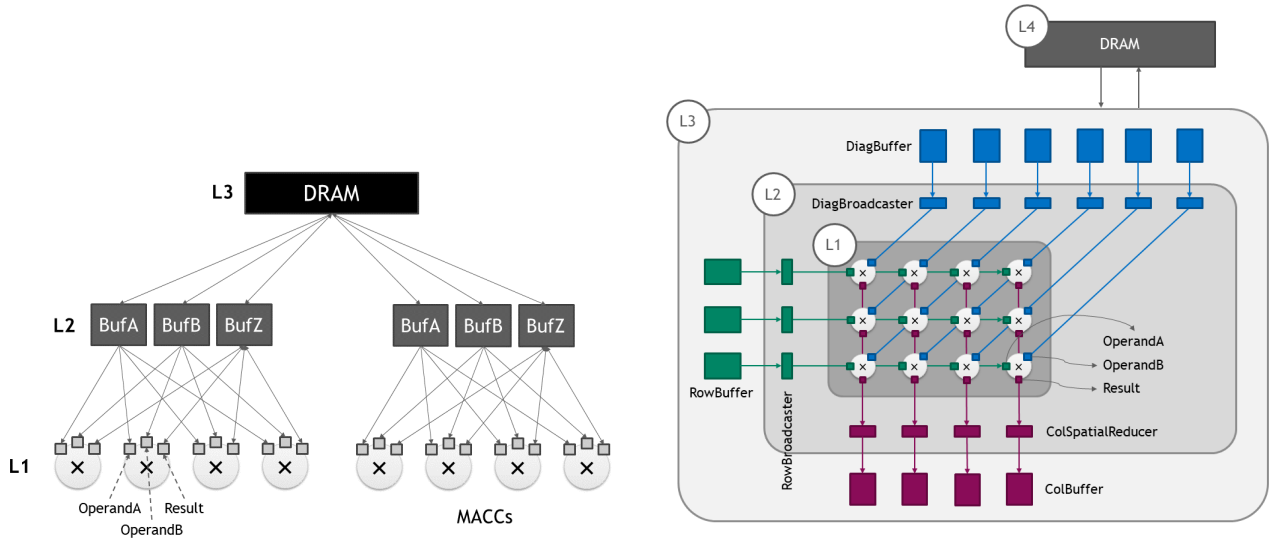
Global space-time vectors: The values of the space and time vectors corresponding to a particular level in the hardware are not global in the overall dynamic execution but relative to its parent nodes. However, global vectors/coordinates are required for the entire mapping analysis and code generation. As a result, we consider the space and time vectors of all ancestor nodes, including the node itself, to construct a global space and time vector for a node. We formally represent the global coordinates for a given level k of a n-level hardware as $\Theta_k^{GST}(\overrightarrow{s_k}, \overrightarrow{t_k})$.

3.2 Physical Hardware Space-Time (PHST)

The SHST abstraction provides a convenient substrate to project a mapping of the workload onto the strictly hierarchical EDDO architectures and reason about runtime behavior of the hardware components. This strictly hierarchical structure constrains the complexity of mapping problem, while capturing a rich set of interesting hardware topologies.

There is however one critical architectural motif that is commonly used in EDDO architectures, but breaks the strict hierarchy: **hard-partitioned buffers.** Figure 4(a) shows an example architecture (a slight modification of the architecture in Figure 3(a)) in which each L2 Buffer instance has split into 3 partitions. Also, the operand and result registers at the MACC unit have been exposed. The BufA, BufB and BufZ partitions are connected to the OperandA, OperandB and Result registers, respectively.

A more complex example is shown in Figure 4(b). This style of architecture is useful for performing sliding-window filter operations, as seen in convolutional neural networks [5]. In this example, the DRAM is connected to a set of Row, Column and Diagonal buffers, each with a different number of instances. Each Row and Diagonal buffer is attached to a Broadcaster that broadcasts a tensor element along a row or diagonal of the MACC array. Each Column buffer is connected to a hardware reduction unit that collects partial sums from the MACC array and reduction-sums them before placing them in the Column buffer. In this example, the hierarchy underneath each partition at the L2 level diverges into non-isomorphic shapes before re-converging into a common set of arithmetic units.



(a) (b)

$$\Theta_3^{HST}(DRAM) = \{\Theta_3^{GST}(\vec{s_3}, \vec{t_3}) \rightarrow \Theta^{PHST}(DRAM(\overrightarrow{s_{dram}}, \overrightarrow{t_{dram}}))\} = \{SpaceTime_3[0,0] \rightarrow DRAM[0,0]\}$$

$$\Theta_2^{HST}(BufA) = \{\Theta_2^{GST}(\vec{s_2}, \vec{t_2}) \rightarrow \Theta^{PHST}(BufA(\overrightarrow{s_{bufa}}, \overrightarrow{t_{bufa}}))\} = \{[SpaceTime_3[0,0] \rightarrow SpaceTime_2[s_2, t_2]] \rightarrow BufA[s_2, t_2]\}$$
$$\Theta_2^{HST}(BufB) = \{\Theta_2^{GST}(\vec{s_2}, \vec{t_2}) \rightarrow \Theta^{PHST}(BufB(\overrightarrow{s_{bufb}}, \overrightarrow{t_{bufb}}))\} = \{[SpaceTime_3[0,0] \rightarrow SpaceTime_2[s_2, t_2]] \rightarrow BufB[s_2, t_2]\}$$
$$\Theta_2^{HST}(BufZ) = \{\Theta_2^{GST}(\vec{s_2}, \vec{t_2}) \rightarrow \Theta^{PHST}(BufZ(\overrightarrow{s_{bufz}}, \overrightarrow{t_{bufz}}))\} = \{[SpaceTime_3[0,0] \rightarrow SpaceTime_2[s_2, t_2]] \rightarrow BufZ[s_2, t_2]\}$$

$$\Theta_1^{HST}(OperandA) = \{\Theta_1^{GST}(\vec{s_1}, \vec{t_1}) \rightarrow \Theta^{PHST}(OperandA(\overrightarrow{s_{operanda}}, \overrightarrow{t_{operanda}}))\}$$
$$= \{[SpaceTime_3[0,0] \rightarrow [SpaceTime_2[s_2, t_2] \rightarrow SpaceTime_1[s_1, t_1]]] \rightarrow OperandA[2s_2 + s_1, t2, t1]\}$$
....
$$\Theta_1^{HST}(Result) = \{\Theta_1^{GST}(\vec{s_1}, \vec{t_1}) \rightarrow \Theta^{PHST}(Result(\overrightarrow{s_{result}}, \overrightarrow{t_{result}}))\}$$
$$= \{[SpaceTime_3[0,0] \rightarrow [SpaceTime_2[s_2, t_2] \rightarrow SpaceTime_1[s_1, t_1]]] \rightarrow Result[2s_2 + s_1, t2, t1]\}$$

Common constraints for above relations: $0 \leq s_2 < 2, 0 \leq s_1 < 4$

(c)

$$\Theta_4^{HST}(DRAM) = SpaceTime_4[0,0] \rightarrow DRAM[0,0]$$

$$\Theta_3^{HST}(RowBuffer) = [SpaceTime_4[0,0] \rightarrow SpaceTime_3[y, x, t_3]] \rightarrow RowBuffer[y, t_3]$$
$$\Theta_3^{HST}(DiagBuffer) = [SpaceTime_4[0,0] \rightarrow SpaceTime_3[y, x, t_3]] \rightarrow DiagBuffer[y + x, t_3]$$
$$\Theta_3^{HST}(ColBuffer) = [SpaceTime_4[0,0] \rightarrow SpaceTime_3[y, x, t_3]] \rightarrow ColBuffer[x, t_3]$$

$$\Theta_2^{HST}(RowBroadcaster) = [SpaceTime_4[0,0] \rightarrow [SpaceTime_3[y, x, t_3] \rightarrow SpaceTime_2[0, t_2]]] \rightarrow RowBroadcaster[y, t_3, t_2]$$
$$\Theta_2^{HST}(DiagBroadcaster) = [SpaceTime_4[0,0] \rightarrow [SpaceTime_3[y, x, t_3] \rightarrow SpaceTime_2[0, t_2]]] \rightarrow DiagBroadcaster[y + x, t_3, t_2]$$
$$\Theta_2^{HST}(ColSpatialReducer) = [SpaceTime_4[0,0] \rightarrow [SpaceTime_3[y, x, t_3] \rightarrow SpaceTime_2[0, t_2]]] \rightarrow ColSpatialReducer[x, t_3, t_2]$$

$$\Theta_1^{HST}(OpndA) = [SpaceTime_4[0,0] \rightarrow [SpaceTime_3[y, x, t_3] \rightarrow [SpaceTime_2[0, t_2] \rightarrow SpaceTime_1[0, t_1]]]] \rightarrow OpndA[y, x, t_3, t_2, t_1]$$
$$\Theta_1^{HST}(OpndB) = [SpaceTime_4[0,0] \rightarrow [SpaceTime_3[y, x, t_3] \rightarrow [SpaceTime_2[0, t_2] \rightarrow SpaceTime_1[0, t_1]]]] \rightarrow OpndB[y, x, t_3, t_2, t_1]$$
$$\Theta_1^{HST}(Result) = [SpaceTime_4[0,0] \rightarrow [SpaceTime_3[y, x, t_3] \rightarrow [SpaceTime_2[0, t_2] \rightarrow SpaceTime_1[0, t_1]]]] \rightarrow Result[y, x, t_3, t_2, t_1]$$

Common constraints for all relations: $(0 \leq x < 4) \wedge (0 \leq y < 3)$

(d)

Figure 4. (a) Example architecture with partitioned L2 buffers and partitioned operand/result registers at the MACC. (b) Example architecture with diverging subtrees underneath partitioned buffers. (c) Symbolic-to-Physical projections for architecture (a). (d) Symbolic-to-Physical projections for architecture (b). The word "Operand" is sometimes shortened to "Opnd" to fit within the page layout.

As with most problems in Computer Science, we solve this problem by introducing an additional level of abstraction: physical hardware units live in distinct *flat* space-time spaces (one space per level per partition). We represent runtime behavior of each physical hardware unit with a single *(space, time)* node, and we represent this collection of nodes for all the units as Physical Hardware Space-Time abstraction (PHST) for the architecture. For example, the PHST abstraction for the hardware unit shown in Figure 4(c) includes the space vector ($\vec{s} = s_2$) and time vector ($\vec{t} = t_2$) for the `BuffA` unit, i.e., $\Theta^{PHST}(\text{BuffA}(\vec{s}, \vec{t}))$.

Note that while the SHST abstraction is a linear chain of space-time vectors of each level of the hardware, the PHST abstraction is a flat collection of space-time vectors of each physical unit in the hardware.

3.3 HST: Connecting SHST with PHST

Our Hardware Space Time (HST) abstraction projects the SHST abstraction onto the PHST abstraction to represent a complete picture of the runtime behavior of all units in a hierarchical hardware along with non-hierarchical components (e.g., hard-partitioned buffers). Additionally, the projection changes the implicit relation between levels of the SHST and physical units to an explicit relationship.

Representation: The major purpose of our HST abstraction is to symbolically capture the runtime behavior of all the physical units in the hardware. Hence, our abstraction includes a projection/map to PHST abstraction of each physical unit from their corresponding level in the SHST abstraction with global space-time vectors. This is mathematically represented as $\Theta_k^{GST}(\overrightarrow{s_k}, \overrightarrow{t_k}) \rightarrow \Theta^{PHST}(P_U(\overrightarrow{s_U}, \overrightarrow{t_U}))$, which denotes the projection onto the PHST for the unit `U` at the level-k in the hardware from its corresponding SHST with global space-time vectors.

Consider the architecture in Figure 4(a). The SHST abstraction for this architecture is *identical* to that in Figure 3. The projections from the SHST onto the PHST is shown in 4(c). Given a global space-time coordinates of a level of the hardware and a string identifying the physical unit name, these projections return a specific hardware unit's physical space-time coordinate. The SHST and PHST separation and their projection elegantly capture the more complex architecture in 4(b) as well. The HST abstraction, i.e., projection for this architecture is shown in Figure 4(d) (we omit a visual depiction of the SHST for this example for brevity). Observe the $y + x$ projection in the $\Theta_3^{HST}(DiagBuffer)$ and $\Theta_2^{HST}(DiagBroadcaster)$. $y + x$ is a hyperplane along the diagonal of the 2D array, and the DiagBuffer and DiagBroadcasters sit along this diagonal hyperplane. This hardware is carefully designed to cater to the specific algorithmic shape of a sliding-window convolution.

The SHST abstraction provides a fabric onto which an algorithm's execution order can be *mapped*. For a given mapping, the symbolic-to-physical relations via the HST abstraction and the workload's tensor-access relations together can precisely identify what set of tensor coordinates must be present at each space-time coordinate in each hardware unit to honor the mapping.

4 PolyEDDO: a Mapping Analysis and Code-Generation Framework

In this section we describe our work-in-progress on implementing a mapping analysis and code-generation framework for EDDO architectures based on the hardware abstraction (HST) presented in Section 3. An overview of our framework is summarized in Figure 5, which takes architecture description in HST abstraction, workload and its mapping, and outputs the decoupled programs for each of the hardware units, along with the data movement and computation activity counts for performance analysis. An in-depth coverage of the flow is outside of the scope of this submission. However, we provide a high-level overview of the process in this section supported with some examples. Our objective is to help the reader build an intuition on how our hardware abstraction can help navigate the challenges of targeting EDDO architectures.

Note that PolyEDDO is a code generator but not an optimizer (or *mapper*). However, its data-movement analysis can serve as a critical step in guiding an optimizer.

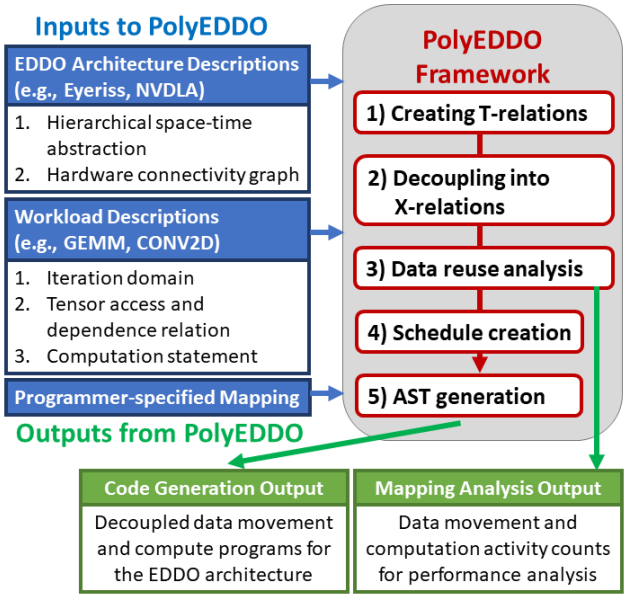


Figure 5. Overview of our PolyEDDO framework.

4.1 PolyEDDO Inputs

1) Architecture description: An EDDO architecture is completely specified by:

(1)—its Hierarchical space-time abstraction[2]. The input includes the Presburger relations only on the *space* vectors, and the *time* vectors will be constructed with a given mapping and workload. For example, the architecture in Figure 4(b) can be described using the Θ^{HST} relations in Figure 4(d), but without any of the $\vec{t}$ vectors.

(2)—a relation H representing the hardware's connectivity graph. This relation includes entries of the form $P_S[\vec{s}] \rightarrow P_D[\vec{d}]$ if the physical hardware unit S at space vector $\vec{s}$ is allowed to send a message to the unit D at space vector $\vec{d}$.

2) Workload description A workload is described using the standard components of the polyhedral model, i.e., iteration domain ($\mathcal{D}$), a set of tensor access read and write relations ($\mathcal{A}^r, \mathcal{A}^w$), and a set of dependence relations. These can be parsed in from a reference (or *unmapped*) code block, or entered directly in the form of sets and relations.

```
for(int k = 0; k < K; k++)
 for(int p = 0; p < P; p++)
  for(int r = 0; r < R; r++)
   Outputs[k][p] = Outputs[k][p] + \
       Inputs[p+r] * Weights[k][r]; //s
```

Figure 6. Running example – 1D convolution workload with multiple weights (`k-loop`).

In this rest of section, we use the running example shown in Figure 6, i.e., 1D convolution workload with multiple weights, to illustrate the various steps in our framework. The iteration domain and access relations for the statement `s` in the running example are as follows:

$\mathcal{D}(s(\vec{i_s})) = \{[k,p,r] \mid (0 \leq k \leq K-1) \wedge (0 \leq p \leq P-1) \wedge (0 \leq r \leq R-1)\}$

$\mathcal{A}^r_W(s(\vec{i_s})) = \{[k,p,r] \rightarrow Weights[k,r]\}$

$\mathcal{A}^r_I(s(\vec{i_s})) = \{[k,p,r] \rightarrow Inputs[p+r]\}$

$\mathcal{A}^r_O(s(\vec{i_s})) = \{[k,p,r] \rightarrow Outputs[k,p]\}$

$\mathcal{A}^w_O(s(\vec{i_s})) = \{[k,p,r] \rightarrow Outputs[k,p]\}$

3) Mapping description: We view mapping as an assignment of program's statement instances onto the HST abstraction, i.e., an assignment of work to a physical hardware unit's space-time coordinates. Critically, because we are programming an EDDO architecture in a hierarchical fashion, the work assigned to each space-time coordinate is *not* necessarily a single statement instance; each space-time coordinate is assigned a *tile*'s worth of work. These tiles are constructed by hierarchically decomposing the iteration space. This is valid for the workloads that we currently assume in the implementation, i.e., perfectly nested affine loops without dependencies violating multi-level tiling. Concretely, a user provides three sets of information to specify a mapping:

(1)—A set of tiling factors for each level of the hardware hierarchy. We assume a 3-level hardware architecture as an example, and as result, it results in 3-levels of tiling the original iteration space. We symbolically use K_i, P_i and R_i to represent user-provided tiling factors for the level-i of the loops in the running example. Also, (k_i, p_i, r_i) denotes the iteration vector over the level-i tiles in the running example.

(2)—A set of projections at each level from a tiled iteration space to a symbolic space-time coordinate/vector at that level. For example, the following relation projects a tiled iteration space at level 3 to a $SpaceTime_3$ node in the SHST of the architecture in Figure 4(b)/(d).

$$\{\ [k_3,p_3,r_3] \rightarrow SpaceTime_3[\vec{s_3},\vec{t_3}] : \vec{s_3} = (y,x) \wedge \vec{t_3} = (t) \wedge k_3 = t \wedge p_3 = x \wedge r_3 = y\ \}$$

Note that the loop order over a tiled iteration space is represented in this projection using the time vector (e.g., $\vec{t_3}$). Certain projections may violate dependencies, but in our present implementation correctness of projections is the responsibility of the programmer or automated mapper generating mappings for PolyEDDO. In future, we hope to implement a static validation step that checks whether the mapping is legal on the hardware and implements the intended semantics of the problem. We also plan to integrate the mapping description with schedule trees [28] to describe program execution order of imperfectly nested affine loops and also a broader sequence of affine loop nests onto our HST abstraction.

(3)—A set of *bindings* at each level that bind each workload tensor to a physical hardware partition. For example:

$$\begin{aligned} binding_4(\text{Weights}) &= \text{DRAM} \\ binding_4(\text{Inputs}) &= \text{DRAM} \\ binding_4(\text{Outputs}) &= \text{DRAM} \\ binding_3(\text{Weights}) &= \text{RowBuffer} \\ binding_3(\text{Inputs}) &= \text{DiagBuffer} \\ binding_3(\text{Outputs}) &= \text{ColBuffer} \end{aligned}$$

Observe that all tensors are bound to the DRAM at level 4 while each tensor is bound to a distinct buffer partition at level 3. We could have chosen a more exotic binding at level 3, with two tensors bound to (and sharing) a single partition with the third tensor solely occupying a second partition. Such a binding may or may not be legal, depending on the ability of the interconnection network to route data between hardware units.

4.2 PolyEDDO Workflow

4.2.1 Creating T-relations. Given an architecture, workload, and mapping specifications, PolyEDDO first constructs

[2]However, this can be deduced even from a high-level specification of the hardware. For now, we explicitly take this as an input to the PolyEDDO

a set of *tiling relations* (or *T-relations*). Given a hierarchical symbolic space-time global coordinate, a T-relation identifies the set of tensor or iteration-space coordinates participating in a specific activity at that space-time coordinate. For example, the T-relation:

$$\begin{aligned} T_2^{Weights} &= \{\Theta_2^{GST}(\vec{s}_2, \vec{t}_2) \rightarrow Weights[k, r]\} \\ &= \{(\vec{s}_4, \vec{t}_4, \vec{s}_3, \vec{t}_3, \vec{s}_2, \vec{t}_2) \rightarrow Weights[k, r] : (\vec{s}_4 = 0 \wedge \vec{t}_4 = 0) \\ &\quad \wedge (\vec{s}_3 = (y, x) \wedge \vec{t}_3 = 0) \wedge (\vec{s}_2 = 0 \wedge \vec{t}_2 = t_2) \\ &\quad \wedge (y = r) \wedge (0 \leq x < 4) \wedge (t_2 = k)\} \end{aligned}$$

identifies which *Weights* coordinates must be *read* at the given level-2 hierarchical space-time global coordinate, i.e., $\Theta_2^{GST}(\vec{s}_2, \vec{t}_2)$ (refer to the architecture in Figure 4(b)/(d)). Observe that (a) a distinct filter coordinate r is assigned to each hardware row y (condition: $y = r$), (b) the filters k are delivered over time t_2 (condition: $t_2 = k$), and (c) all hardware columns x share the same tensor coordinates, opening up the possibility of a broadcast. T-relations are constructed for data movement and compute activities.

4.2.2 Decoupling into X-relations. Next, each hierarchical T-relation is *decoupled* (i.e., decomposed) into a set of parent-child *data-transfer relations* (or *X-relations*). X-relations specify the data that must be transferred between parents and children at a given set of space-time coordinates in order to execute the provided mapping. Thus, they form the basis for the explicit, decoupled data orchestration that we are trying to describe.

The decoupling process involves three transformations: (a) symbolic space-time is projected into physical hardware space-time, (b) the hierarchy is collapsed into pairwise parent-child data-transfer relations, and (c) hardware space-time coordinates are transformed into flat/absolute coordinates. The resulting X-relations look like the following:

$$\begin{aligned} X_2^{Weights} &= [\Theta^{PHST}(RowBroadcaster(\overrightarrow{s_{RB}}, \overrightarrow{t_{RB}})) \rightarrow \\ &\quad \Theta^{PHST}(OperandA(\vec{s}_A, \vec{t}_A))] \rightarrow Weights[k, r] \\ &= \{[RowBroadcaster[\overrightarrow{s_{RB}}, \overrightarrow{t_{RB}}] \rightarrow \\ &\quad OperandA[\vec{s}_A, \vec{t}_A]] \rightarrow Weights[k, r] \\ &\quad : \overrightarrow{s_{RB}} = y \wedge \overrightarrow{t_{RB}} = t \wedge \vec{s}_A = (y, x) \wedge \vec{t}_A = t \\ &\quad \wedge (y = r) \wedge (t = k) \wedge (0 \leq x < 4)\ \} \end{aligned} \tag{1}$$

The X-relation in this example performs buffet *read* operations at the parent and a buffet *fill* operation at the child. Similarly, X-relations can perform buffet *drain* and *update* operations. Computation is expressed as a transformation of data as it flows through an X-relation. These relations form the basis for the set of heterogeneous, distributed programs that work in concert to execute the algorithm's mapping. Each X-relation represents an independent decoupled program that synchronizes with other programs via buffets.

4.2.3 Reuse Analysis. The X-relations do not take reuse into account; they insist on delivering a full tensor data tile from a parent to each child coordinate. However, data tiles often overlap, which means parent-to-child data movement can be elided thanks to *reuse* opportunities. EDDO architectures support exploitation of various forms of reuse: local temporal reuse, forwarding from peers, multicasting data over wires from parents, and spatial reduction of partial outputs via dedicated reduction networks.

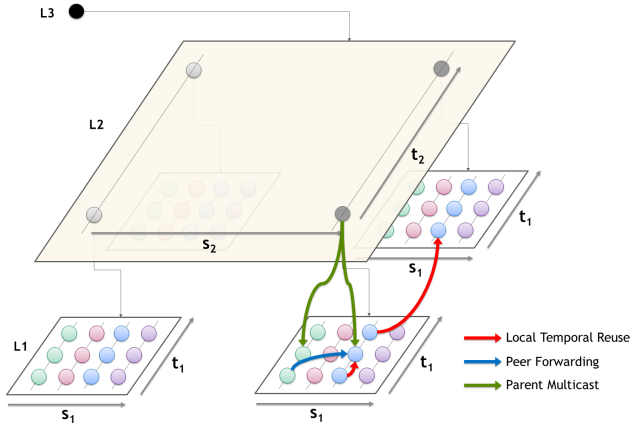


Figure 7. Illustration of different forms of reuse shown over space-time. Spatial reduction is treated as a form of multicast for tensor updates.

Fortunately, the space-time abstraction makes it intuitive to reason about all these forms of reuse. This is illustrated in Figure 7. This leads to straightforward transformations to account for reuse. For example, consider the X-relation in Equation 1. To exploit *local temporal reuse* in the `OperandA` buffer, we derive a modified *delta-transfer relation* (or Δ-*relation*) that takes into account the fact that data that was already present in the buffer at time t need not be fetched from anywhere at time $t + 1$. We first define:

$$\begin{aligned} R &= \{[\Theta^{PHST}(P(\vec{s}_p, \vec{t}_p)) \rightarrow \Theta^{PHST}(C(\vec{s}_c, \vec{t}_c))] \\ &\quad \rightarrow [\Theta^{PHST}(P(\vec{s}'_p, \vec{t}'_p)) \rightarrow \Theta^{PHST}(C(\vec{s}'_c, \vec{t}'_c))] \\ &\quad : \quad (\vec{s}_p = \vec{s}'_p) \wedge (\vec{s}_c = \vec{s}'_c) \wedge (\vec{t}_c \prec \vec{t}'_c)\ \} \\ &= \{[P[\vec{s}_p, \vec{t}_p] \rightarrow C[\vec{s}_c, \vec{t}_c]] \rightarrow [P[\vec{s}'_p, \vec{t}'_p] \rightarrow C[\vec{s}'_c, \vec{t}'_c]] \\ &\quad : \quad (\vec{s}_p = \vec{s}'_p) \wedge (\vec{s}_c = \vec{s}'_c) \wedge (\vec{t}_c \prec \vec{t}'_c)\} \end{aligned}$$

where P and C represent the parent and the child in X-relations, respectively. Thus, P represents `RowBroadcaster` and C represents `OperandA` in the above $X_2^{Weights}$ relation. The operator $\prec$ represents the closest time vector in their lexicographic orderings.

$$\Delta_2^{Weights} = X_2^{Weights} - (R^{-1} \circ X_2^{Weights})^{-1} \tag{2}$$

The above approach of computing temporal reuse elements considering the nearest time vector in the lexicographic ordering is inspired from the prior work on analyzing the cache behavior of affine programs by Wen et al. [2]. We have similar Δ-relations for the other forms of reuse such as spatial and spatio-temporal reuses. It is an interesting problem because there is sometimes a choice in the source for a particular piece of data: it may be available at multiple peers along with a parent. The availability of multicast capabilities makes this choice a non-trivial problem where the globally-optimal solution cannot be arrived at with a greedy approach. At this time, we believe that a heuristic solution will be required. Note that this space of choices is within the scope of a single user-provided mapping—the mapping search is itself a distinct (and hard) optimization problem. It is also possible to expose the reuse choices to the programmer or mapper as part of this mapping space.

Note that the cardinality of Δ-relations can be used to determine reuse factors.

4.2.4 Data Tile Delivery Schedule Creation. While the Δ-relations specify a schedule for moving tiles from one unit to another (after accounting for reuse), they do not prescribe an order to move the individual elements within a data tile (except at the leaf levels of the space-time hierarchy where tiles are unit-sized). Choosing the delivery order for data elements within a data tile has no impact on reuse factors, but can impact performance and energy efficiency due to (a) spatial locality of accesses to blocks/lines in buffers, affecting performance and energy efficiency, and (b) pipelining of data delivery with consumption, affecting performance.

A schedule maps a point in an iteration space to a time-coordinate. In a Δ-relation, each element is essentially a point in a *data movement iteration space* that represents a data-movement action. A schedule for a Δ-relation as in Equation 2 needs to create a total order for these points by interleaving the $\vec{s_p}$, $\vec{t_p}$, $\vec{s_c}$, $\vec{t_c}$ and tensor coordinates. We are effectively writing a program that a parent node will execute to read data from its local storage and deliver to a set of children. Here is an example schedule for a Δ-relation:

$$[[P[\vec{s_p}, \vec{t_p}] \rightarrow C[\vec{s_c}, \vec{t_c}]] \rightarrow Tensor[\vec{m}]] \rightarrow [\vec{s_p}, \vec{t_p}, \vec{t_c}, \vec{m}, \vec{s_c}]$$

Informally, this schedule behaves as follows (refer to the space-time diagram in Figure 7 as you read this text). $\vec{s_p}$ identifies a distinct parent node and is unrolled. At each parent unit $\vec{s_p}$, the following program plays out: For each parent time-step $\vec{t_p}$, for each child time-coordinate $\vec{t_c}$, for each tensor coordinate $\vec{m}$, read the tensor value, and send it to all children $\vec{s_c}$ that need it.

In addition to determining delivery orders, we also perform some required program-order serialization of different Δ-relations during the scheduling step. Although buffets perform most of the heavy-lifting for synchronizing Δs, there are some operations that need to be explicitly serialized. We hope to elaborate on this in a future publication.

Once schedules are determined, we can derive *activity counts* representing the workload's execution, which can be fed into analytical models for performance and energy estimation.

4.2.5 AST Generation. Abstract Syntax Trees (ASTs) for each Δ schedule are automatically generated using the polyhedral framework. We reiterate that unlike traditional architectures, these ASTs (of which there can be tens or even hundreds) represent a distributed, heterogeneous collection of synchronized programs operating in concert to execute the workload's dataflow.

These ASTs are the final PolyEDDO output. Generating the actual *binary* program or configuration for the individual state machines on the hardware from each AST is a highly machine-dependent but mechanical process that is outside of the scope of this work.

5 Related work

In this section, we restrict our attention to describing related work around our HST abstraction, and we skip the discussion about mapping analysis and code generation for different instances of EDDO architectures. However, we believe that our HST abstraction, mapping analysis, and code generation techniques can be useful for targetting EDDO architectures in popular ML frameworks leveraging MLIR [19] or TVM [4]. In the rest of this section, we briefly contrast our abstraction with prior work on multi-dimensional timestamps [11, 15], schedule trees [28], Maestro data-centric abstraction [17], Timeloop abstraction [22], and Hierarchical place trees (HPTs) [30].

Multi-dimensional timestamps and Schedule trees: Multi-dimensional timestamps abstraction (in the form of Kelly's notation [15] or Girbal notation [11]) is a classical way of encoding a program execution order in the polyhedral model via timestamps. Recently, schedule trees abstraction [28] is introduced to explicitly capture the program execution order in the form of trees, and it has been shown as a more natural, more practical, and more comfortable abstraction to understand by programmers. Both of these representations aim to capture the program execution order for enabling optimizations and code generation, where as our abstraction is introduced to *explicitly* capture the dynamic behavior of EDDO-architecture components at compile-time.

Maestro data-centric abstraction: Recently, Kwon et al. [17] introduced Maestro data-centric abstraction to explicitly capture data movement behavior of a workload and its mapping on spatial DNN accelerators for compute-intensive kernels such as CONV2D and GEMM. This abstraction enabled their cost model to estimate reuse factors, execution time, and energy efficiency of a mapping relatively faster

than inferring from the loop-nest specification of a mapping. The followup work from Chatarasi et al. [3] characterized the set of programs that are conformable with the Maestro abstraction, and these set of programs are found to be a strict subset of affine loop nests, i.e., perfectly nested loops with affine subscripts, only reduction-dependences, and rectangular iteration spaces. However, our HST abstraction leveraging the polyhedral model can cover a more extensive set of programs relative to the maestro data-centric abstraction. Furthermore, the Maestro abstraction is currently limited to only considering hierarchical hardware, unlike our abstraction covering non-hierarchical and non-uniform hardware.

Timeloop: Timeloop [22] is a mapper/cost model framework for EDDO architectures which uses a hierarchical architectural abstraction. HST improves on Timeloop's abstraction by providing a clear separation between symbolic and physical hierarchies, which allows for a more natural representation of partitioned topologies, especially those with non-isomorphic subtrees as in Figure 4(b). Thanks to its polyhedral underpinnings, PolyEDDO supports a wider set of workloads and mappings than Timeloop's analysis module and is likely to be easier to extend compared to Timeloop's custom implementation [21]. In future we hope to integrate PolyEDDO with Timeloop and take advantage of its mapper, analytical microarchitectural models, and integration with the Accelergy [29] infrastructure.

Hierarchical place trees: The HPT abstraction [30] is introduced to provide a portable abstraction for task parallelism and data movement. The abstraction also supports the co-allocation of data and computation at multiple memory/-compute hierarchy levels. The HPT abstraction is limited to capturing only the hierarchical units, and it lacks the support for non-hierarchical units, and the "time" dimensions to capture hardware behavior across the time.

Recently, Kong [16] introduced an approach to explicitly capture the complex topologies such as Manhattan connections using the Polyhedral model, and such approaches are complementary and can enhance our abstractions.

6 Conclusions and Future Work

In this paper, we presented an abstraction called *Hierarchical Space-Time* (HST) that symbolically captures the runtime behavior of all memory and compute units of an EDDO architecture, given a workload and its mapping. The abstraction is rich enough to describe all the unique attributes of EDDO architectures (arbitrarily deep memory hierarchies, decoupled data movement and compute engines, nested parallelism, non-hierarchical components), while also providing a convenient target for mapping analysis and automated code-generation. Via examples, we also illustrated the use of HST in the context of an in-progress analysis/code-generation framework called PolyEDDO. We hope that the abstraction is either directly useful, or provides useful insights, to other frameworks targeting EDDO architectures.

Although this work focused on HST, we hope to present the entire PolyEDDO infrastructure in the future. We also hope to extend this infrastructure to support imperfectly nested affine loops, a sequence of affine loops, compressed-sparse tensors and EDDO architectures with optimized structures to support their traversal, possibly leveraging recent work on sparse polyhedral frameworks [1, 26].

References

[1] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. 2019. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 625–639. https://doi.org/10.1145/3314221.3314615

[2] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. 2018. Analytical modeling of cache behavior for affine programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 32:1–32:26. https://doi.org/10.1145/3158120

[3] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2020. Marvel: A Data-centric Compiler for DNN Operators on Spatial Accelerators. arXiv:2002.07752 [cs.DC]

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[5] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*.

[6] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[7] J. Cong and J. Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240838

[8] Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. 1991. On estimating and enhancing cache effectiveness. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 328–343.

[9] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C. Chen, P. Chuang, T. Fox, G. Gristede, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S. Lo, G. Maier, M. Scheuermann, S. Venkataramani, C. Vezyrtzis, N. Wang, F. Yee, C. Zhou, P. Lu, B. Curran, L. Chang, and K. Gopalakrishnan. 2018. A Scalable Multi- TeraOPS Deep Learning Processor Core for AI Trainina and Inference. In *2018 IEEE Symposium on VLSI Circuits*. 35–36. https://doi.org/10.1109/VLSIC.2018.8502276

[10] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. ACM, New York, NY, USA, 84–93. https://doi.org/10.1145/3289602.3293906

[11] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl J. of Parallel Programming* 34 (2006), 2006.

[12] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A Fast Analytical Model of Fully Associative Caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 816–829. https://doi.org/10.1145/3314221.3314606

[13] K Hegde, R Agrawal, Y Yao, and CW Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-Based Video Understanding. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 933–946.

[14] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[15] W. Kelly and W. Pugh. 1995. A unifying framework for iteration reordering transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 1. 153–162 vol.1. https://doi.org/10.1109/ICAPP.1995.472180

[16] Martin Kong. 2019. Abstractions for Polyhedral Topology-Aware Tasking. In *International Workshop on Languages and Compilers for Parallel Computing*.

[17] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 754–768. https://doi.org/10.1145/3352460.3358252

[18] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 461–475. https://doi.org/10.1145/3173162.3173176

[19] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020). arXiv:2002.11054 https://arxiv.org/abs/2002.11054

[20] NVIDIA. 2018. NVIDIA Deep Learning Accelerator (NVDLA). https://nvldla.org.

[21] Angshuman Parashar, Anurag Mukkara, Yannan Wu, Po-An Tsai, Priyanka Raina, Lillian Pentecost, Yakun Sophia Shao, and Joel Emer. 2019. Timeloop. https://github.com/NVlabs/timeloop.

[22] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 304–315.

[23] M. Pellauer, Y. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. Keckler, C. Fletcher, and J. Emer. [n.d.]. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *ASPLOS'19*.

[24] Vivek Sarkar, William Harrod, and Allan E. Snavely. 2010. Software Challenges in Extreme Scale Systems. (Jan 2010). Special Issue on Advanced Computing: The Roadmap to Exascale.

[25] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 14–27.

[26] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[27] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and Exact Analysis for LRU Caches. *Proc. ACM Program. Lang.* 3, POPL, Article 54 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290367

[28] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*.

[29] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[30] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2010. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing*, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 172–187.