Abstract
Tensor kernels in machine learning (ML) often correspond to pure mathematical expressions, making term rewriting an attractive strategy for optimization and mapping to specialized hardware accelerators. However, existing ML intermediate representations (IRs) tend to either be pure but high-level, making low-level rewrites to hardware targets inexpressible, or low-level but impure, hampering the use of term rewriting altogether.

This paper introduces Glenside, a pure IR whose core abstraction—the access pattern—enables low-level, layout-aware, hardware-centric program rewrites. We demonstrate how term rewriting in Glenside can be used to map program fragments to hardware accelerator invocations and automatically discover classic data layout transformations like im2col. Glenside establishes a new foundation for exploring further term rewriting techniques in optimizing low-level tensor programs.

CCS Concepts:
• Software and its engineering → Domain specific languages;
• Computing methodologies → Machine learning;
• Theory of computation → Equational logic and rewriting.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MAPS ’21, June 21, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8467-4/21/06...$15.00
https://doi.org/10.1145/3460945.3464953

1 Introduction
Machine learning (ML) and other high-performance computing (HPC) applications increasingly rely on specialized hardware accelerators to provide speed and energy efficiency [17, 19, 32]. This trend has highlighted the need for flexible accelerator support in domain-specific compilers like Halide [30], TVM [9], TensorFlow/MLIR [1, 20], and PyTorch [28].

Adding accelerator support to an existing compiler typically uses custom pattern matching to map expensive tensor operations from applications down to accelerator invocations [11, 43]. Pattern matching often additionally relies on various other transformations to canonicalize intermediate representations (IRs) and massage data layouts into formats matching accelerator requirements [26]. Even with these changes, users may need to manually modify their application to help the compiler discover opportunities for dispatching operations to accelerators, such as by changing data types or unrolling loops.

In principle, term rewriting techniques [3] should be able to facilitate many of these transformation and mapping tasks within a compiler. Halide and TVM already rely on extensive rewrite systems for optimizing scalar computations and simplifying loop bounds in order to support further downstream optimizations [15, 24].
Unfortunately, existing IRs in compilers for array/tensor programming DSLs tend to present abstraction and granularity mismatches that hamper term rewriting approaches. Term rewriting is most easily applied in pure (side effect–free) IRs that support equational reasoning. At the same time, mapping to accelerators requires considering low-level hardware details like data layout. Existing pure IRs for ML frameworks are used primarily for high-level transformations (e.g., type elaboration and inlining) and do not expose low-level data layout details [33]. On the other hand, IRs used for crucial lower-level optimizations like operator fusion must support precise reasoning about memory use, and therefore are typically impure, hampering term rewriting.

To help mitigate such impedance mismatches, we present Glenside, a pure tensor program IR that enables hardware-level term rewriting. Glenside is based on a simple access pattern abstraction that supports expressing and reasoning about data layout transformations via syntactic rewrite rules. When combined with standard arithmetic rewrites for per-tensor-element computations, access patterns enable implementing complex transformations for accelerator support as compositions of simple rewrites.

Tensors are traditionally characterized by their shape, an n-tuple of positive integers indicating the size of each of a tensor’s dimensions. Access patterns instead characterize each tensor with two shapes, e.g., \((x, y, z)\), separating the dimensions which are *iterated over* from the dimensions which are *computed on*. Figure 1(c) depicts an example where a 3D tensor’s first dimension is iterated over and some computation applied to each corresponding 2D matrix.

We demonstrate how Glenside enables implementing representative hardware-level transformation via term rewriting, including mapping computations to systolic arrays [17] (a common hardware module in ML accelerators) and automatically discovering the im2col data layout transformation [7], which enables mapping 2D convolutions to matrix multiplication hardware. In particular, by employing equality saturation [42], these transformations “fall out for free” (i.e., without any carefully crafted rewrite orderings [41]), from a handful of general rewrites concerning tensor transposition, Cartesian product, dot product, etc., expressed in terms of access patterns.

To summarize, our contributions include:

- *Access patterns*, a tensor representation that employs a simple, extended tensor shape type to distinguish iteration and computation dimensions
- The Glenside IR, a pure compiler IR that facilitates term rewriting to enable support for specialized accelerators
- A library of generic rewrites over Glenside programs
- Case studies demonstrating how Glenside enables automatically discovering key transformations for mapping applications to custom accelerators via equality saturation with the egg [42] library.

The rest of the paper is organized as follows: Section 2 provides background and briefly surveys closely related work. Section 3 motivates Glenside via a running example exploring pure matrix multiplication. Section 4 details the design and implementation of Glenside. Section 5 details case studies showing the potential benefits of Glenside’s term rewriting approach to low-level tensor program transformations.

## 2 Background and Related Work

Glenside is designed to help target tensor hardware accelerators and builds on past work in tensor IRs and term rewriting.

### 2.1 Machine Learning Accelerators

A variety of accelerators [12, 17, 21, 22, 25] have been developed to provide efficient implementations of tensor operators for ML applications. These devices accelerate tensor operators through hardware parallelism, simultaneously applying related operations across many tensors in the accelerator’s memory (which are often laid out according to custom rules that facilitate hardware optimization). Tensor program compilers must translate expensive application code fragments down to accelerator invocations that adhere to these layout rules, which often involves both (a) higher-level transformations like tensor reshaping to match accelerator size bounds and loop unrolling to expose optimization opportunities, and (b) lower-level transformations like operator fusion and im2col to match accelerator calling conventions and even implement different operations using the same accelerator, e.g., on systolic arrays [7, 16].

### 2.2 Tensor IRs and Compilers

Tensor compilers for ML and HPC applications strive to balance clear, high-level operator semantics and support for the low-level optimizations necessary to target specialized accelerators. Halide [31] achieves this balance by separating operator specifications (what is computed) from schedules (how, when, and where each output element is generated). This style of separation has proven highly effective across both application domains and hardware targets; numerous
compilers including TVM [8], FireIron [14], LIFT [35], and Accelerate [5] follow variations of this strategy.

The specification/schedule separation approach allows the same high-level program (specification) to be flexibly optimized for and mapped to different hardware targets by applying different schedules. From this perspective, schedules represent different rewriting strategies to explore various loop ordering and memory layouts; in LIFT and Accelerate these take the form of functional combinators closely related to Glenside’s approach. As in classic term rewriting, experts must often carefully craft schedules for each target to achieve the best performance and mitigate phase ordering challenges [41], though recent projects have produced promising results in automatic scheduling [2, 10, 45].

Other tensor IRs like TACO [18], Keops [6], and COMET [37] rely on index notation\(^2\) to concisely express tensor operators and simplify optimization by uniformly representing per-output-element computations. These approaches also rely on rewriting passes to generate kernel implementations specialized to tensor sparsity/density, operator combinations arising in the source application, and details of the target hardware. In Section 3 we discuss some of the tradeoffs of these approaches with respect to other rewriting strategies.

Finally, polyhedral compilers [34] like Tensor Comprehensions [39] and Tiramisu [4] optimize loop-filled programs by modeling loop nests as polyhedra and applying geometric transformations. The polyhedral approach exploits regular loop structure, but is also restricted to geometrically affine transformations. In contrast, term rewriting is neither guided nor restricted by geometric constraints, making these approaches broadly complementary.

### 2.3 Term Rewriting and Equality Saturation

Term rewriting is a classic program optimization technique [3] that relies on iteratively applying rewrite rules of the form \(t \rightarrow r\): when part of a program matches the pattern \(t\) under substitution \(\sigma\), it is rewritten into \(\sigma(r)\). This approach is ubiquitous, frequently used in both mainstream and DSL compilers to implement features including preprocessing, strength reductions, and peephole optimizations [13].

Classic term rewriting systems where rewrites are applied destructively suffer phase ordering problems [41]: the order in which rewrites are applied can enhance or severely diminish performance. Recent work has shown how program synthesis can help address this challenge in peephole optimizers like Halide’s scalar expression rewriter [24].

Advances in alternate rewriting techniques like equality saturation [36] also mitigate phase ordering by exploiting the e-graph data structure from SMT solvers to repeatedly apply all rewrites simultaneously, thus obviating rule ordering considerations. In particular, the egg library [42] has been used to develop new synthesis and optimization tools across diverse domains [23, 27, 40], including DSP compiler vectorization [38] and tensor computation graphs [44].

Glenside provides the first tensor IR amenable to equality saturation by introducing access patterns to provide pure, higher order tensor kernel combinators that support rank-polymorphism without the need for binding structures like anonymous functions or index notation.

### 3 From Pure matMul to IR Design Goals

Applying functional techniques and term rewriting to tensor IRs requires careful design. For example, we must ensure that operators be compositional with respect to tensor shapes and that the representation support generic rules within the target rewrite engine. To highlight such constraints and motivate access patterns in Glenside, this section illustrates potential pitfalls with a simple matrix multiplication example.

#### 3.1 Pure Matrix Multiplication

We write \(f64\) for the type of 64-bit floats and \([A]\) for vectors over type \(A\). Using this notation, we can specify operators like dot product and 2D matrix transpose as:

\[
\begin{align*}
\text{dotProd} & : [f64] \times [f64] \rightarrow f64 \\
\text{trans2} & : [[f64]] \rightarrow [[f64]]
\end{align*}
\]

Implementing 2D matrix multiplication on inputs \(P\) and \(Q\) requires computing an output matrix \(R\) where \(R_{ij} = \sum_k P_{ik} Q_{kj}\). The need to compute \text{dotProd} for every pair of a row from \(P\) and a column from \(Q\) suggests map and Cartesian product operators which we might specify with:

\[
\begin{align*}
\text{map} & : (A \rightarrow B) \times [A] \rightarrow [B] \\
\text{cartProd} & : [A] \times [B] \rightarrow [A \times B]
\end{align*}
\]

Naively, we can almost implement matrix multiplication as:

\[
\text{matMul}(P, Q) := \text{map}(\text{dotProd}, \text{cartProd}(P, \text{trans2}(Q)))
\]

However, the result type will have been flattened to just \([f64]\), making it impossible to compose with other matrix operators that expect \(([f64])\) inputs.

Our first problem is that the \text{cartProd} specification above “forgets” the shape of its arguments. We could change this specification to arrange the output as a matrix:

\[
\text{cartProd2D} : [A] \times [B] \rightarrow [[A \times B]]
\]

But this result type prevents directly mapping \text{dotProd}.

\[^3\text{This simple type does not specify how \text{cartProd2D} orders its output relative to its input vectors. We assume the order expected for matrix multiplication.}\]
iterating over the first (outermost) dimension of a tensor. If we specialize map to iterate over the second dimension:

\[
\text{mapAt2} : (A \to B) \times [[A]] \to [[B]]
\]

then we can implement a compositional \(\text{matMul}\) operator that correctly produces results of type \([[f64]]\) as:

\[
\text{matMul}(P, Q) := \\
\text{mapAt2}(\text{dotProd}, \text{cartProd2D}(P, \text{trans2}(Q)))
\]

### 3.2 Glenside Design Constraints and Goals

This style of pure, higher-order functional program representation enables term rewriting and equational reasoning via rules like:

\[
\text{dotProd}(P, Q) \leftrightarrow \text{dotProd}(Q, P)
\]

\[
\text{trans2}(\text{trans2}(P)) \leftrightarrow P
\]

\[
\text{map}(f, \text{map}(g, P)) \leftrightarrow \text{map}(f \circ g, P)
\]

\[
\text{mapAt2}(f, \text{trans2}(P)) \leftrightarrow \text{trans2}(\text{mapAt2}(f, P))
\]

However, some of these rules depend on the shapes of dimension-specific operators aligning. What happens when we need to support higher-dimensional tensors? Without a mechanism to abstract which dimensions of a tensor are being iterated as opposed to computed over, we would have to generate versions of each rule for every combination of dimensions. Worse, these problems do not only affect rewrite rules; they also lead to code blowup just to specify all the variants of tensor kernels that arise in practice.

One strategy to address these challenges is adding support for anonymous functions ("lambdas"), currying, and closures to the tensor program representation. These features can provide sufficient flexibility to handle shape alignment issues that otherwise may require dimension-specific operators like \(\text{cartProd2D}\) and \(\text{mapAt2}\) above. For example, given curried versions of \(\text{dotProd}\) and \(\text{map}\), we could have used such features to implement a curried \(\text{matMul}\) as:

\[
\text{matMul'}(P, Q) := \\
\text{map'}(\lambda r \Rightarrow \text{map'}(\text{dotProd'} r) (\text{trans2 } Q)) P
\]

Alternatively, some IRs rely on index notation for even pitier implementations like:

\[
\text{matMul}(P, Q)[i, j] := \text{dotProd}(P[i], \text{trans2}(Q)[j])
\]

Unfortunately, these approaches all rely on some form of name binding which can significantly complicate term rewriting. Rewriting under binders, whether explicitly in the form of lambdas or implicitly with index notation, requires additionally analyzing the potential contexts (what names are bound to) of every subexpression. While it is still technically possible to apply state-of-the-art rewrite engines like egg [42] via explicit variable substitution rules and free variable analyses, we have found the additional complexity and rewrite search space blow up substantially eliminate the potential advantages of term rewriting in such IR designs.

All the above constraints inform Glenside’s key design goal: providing an IR that flexibly supports specifying and composing higher-order tensor operators\(^4\) over arbitrary dimensions while still enabling high-performance term rewriting techniques like equality saturation. In the rest of this paper, we show how access patterns enable achieving these goals with a focus on applications to mapping application fragments down to specialized hardware accelerators.

### 4 Glenside

This section details Glenside’s implementation, focusing on its core abstraction, access patterns. We use Section 3’s matrix multiplication as a running example throughout.

#### 4.1 Access Patterns

Access patterns encode common tensor IR patterns where some tensor dimensions are iterated over (accessed) while others are computed on.\(^5\) Section 3’s \(\text{matMul}\) example iterates over dimension 0 of input \(P\), while computing on dimension 1, effectively viewing \(P\) as a 1D vector of 1D vectors.

Access patterns are specified by their shape — a pair of tuples of positive integers \((S_A, S_C)\). An access pattern of shape \((S_A, S_C)\) is, in turn, a tensor \(T\) whose shape is given by the concatenation of the access pattern shape tuples \(S_A \uplus S_C\); we refer to \(S_A\) and \(S_C\) as the access and compute dimensions of \(T\), respectively.

Access patterns represent the view of an \((|S_A| + |S_C|)\)-dimensional tensor as a tensor of shape \(S_A\), each of whose elements has shape \(S_C\). For an access pattern \(T\) of shape \((S_A, S_C)\) where \(|S_A| = n_A\), we use the syntax (access \(T\ n_A\)) to represent \(T\) in Glenside. For example, if a 2D matrix \(T\) has shape \((m, n)\), then the Glenside expression (access \(T\ 1\)) yields an access pattern of shape \((m), (n)\).

The matrix multiplication example from Section 3 directly accesses the rows of \(P\), but uses \(\text{trans2}\) to iterate over the columns of \(Q\). Instead of requiring an explicit transpose operator, Glenside provides access pattern transformers.

#### 4.2 Access Pattern Transformers

Access pattern transformers manipulate one or more access patterns to produce a new access pattern, allowing Glenside to support more complex patterns like slicing, transposing, and interleaving. Table 1 lists Glenside’s transformers.

To produce an access pattern representing the columns of \(Q\) for matrix multiplication, we employ the \(\text{transpose}\) transformer. It takes an access pattern and a list of dimension indices, and rearranges the dimensions of the access pattern in the order specified by the indices. If \(Q\) has shape \((N, O)\), (\(\text{transpose} (\text{access } Q \ 1 \ (\text{list } 1 \ 0)\)) produces an access pattern of shape \((\text{O}), (\text{N})\).

\(^4\)As \(\text{map}\) and \(\text{mapAt2}\) in Section 3 illustrate, an IR can support higher-order operators without necessarily providing lambdas, currying, or closures.

\(^5\)This is similar to NumPy’s concept of universal functions.
where the second shape represents a 2-tuple of a row from $A$.

cation example: we have accessed the rows of $P$.

of the input access pattern. Table 2 lists the operators in

calculate.

are the only Glenside constructs which perform

transformer takes access patterns of shapes

The $\text{cartProd}$ transformer takes access patterns of shapes

The different access patterns of shapes

The $\text{slice}$ operator (see Table 2) does just that.

To compute the dot product over our row-column pairs, we need only to apply $\text{compute dotProd}$ to our access pattern, to produce an access pattern with final shape $((M, N), ())$.

The entire Glenside specification of matrix multiplication is shown in Figure 2b.

5 Case Studies

To demonstrate Glenside’s utility, we first show how it enables concise specifications of several critical ML kernels (Section 5.1). We then show how Glenside’s pure, binder-free representation enables mapping kernels to an example accelerator via direct application of generic rewrite rules (Section 5.2). Finally, we highlight how Glenside enables the flexible mapping of larger, more diverse kernels to our accelerator, utilizing the power of equality saturation to automatically discover a variety of program transformations. Specifically, we show how Glenside can automatically map convolutions to matrix multiplications (Section 5.3) and automatically map large matrix multiplications into a sequence of smaller matrix multiplications (Section 5.4).

5.1 Representation of Common ML Kernels

Figure 2 lists the Glenside specifications of three common ML kernels: 2D convolution, matrix multiplication, and max pooling. Below, we discuss the specifications of 2D convolution and max pooling; see Section 4 for a description of matrix multiplication.

2D Convolution. 2D convolution (conv2d) is a core kernel in deep learning, defined element-by-element over tensors storing activations $A$, strides $S$, and weights $W$ as:

$$\text{out}[n, o, x, y] = \sum_{dx,dy} (A[n, c, S[0] \cdot x + dx, S[1] \cdot y + dy] \cdot W[o, c, dx, dy])$$

Table 1. Glenside’s access pattern transformers.

<table>
<thead>
<tr>
<th>Transformer</th>
<th>Input(s)</th>
<th>Output Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>$((a_0, \ldots, a_n))$ and non-negative integer $i$</td>
<td>$((a_0, \ldots, a_{i-1}), (a_i, \ldots, a_n))$</td>
</tr>
<tr>
<td>transpose</td>
<td>$((a_0, \ldots, a_n), \ell$ (a permutation of $(0, \ldots, n-1))</td>
<td>$((a_0, \ldots, a_n))$</td>
</tr>
<tr>
<td>cartProd</td>
<td>$((a_0, \ldots, a_n), (c_0, \ldots, c_p))$, $(b_0, \ldots, b_m)$, $(c_0, \ldots, c_p)$</td>
<td>$((a_0, \ldots, a_n, b_0 \ldots, b_m, 2, c_0, \ldots, c_p))$</td>
</tr>
<tr>
<td>windows</td>
<td>$(a_0, \ldots, a_n), (b_0, \ldots, b_n))$, window shape $(w_0, \ldots, w_n)$, strides $(s_0, \ldots, s_n)$</td>
<td>$((a_0, \ldots, a_n, b_0', \ldots, b'_{m'}, (w_0, \ldots, w_n'), where b'_i \equiv [(b_i - (k_i - 1))/s_i]</td>
</tr>
<tr>
<td>slice</td>
<td>$(a_0, \ldots, a_n))$</td>
<td>$(a'_0, \ldots, a'_n))$</td>
</tr>
<tr>
<td>squeeze</td>
<td>$(a_0, \ldots, a_n), index d where a_d = 1</td>
<td>$((a_0, \ldots, a_n))$ with $a_d$ removed</td>
</tr>
<tr>
<td>flatten</td>
<td>$(a_0, \ldots, a_m), (b_0, \ldots, b_n))$</td>
<td>$((a_0 \cdots a_m), (b_0 \cdots b_n))$</td>
</tr>
<tr>
<td>reshape</td>
<td>$(a_0, \ldots, a_m), (b_0, \ldots, b_n), access pattern shape literal $(c_0, \ldots, c_p), (d_0, \ldots, d_q)$</td>
<td>if $a_0 \cdots a_m = c_0 \cdots c_p$ and $b_0 \cdots b_n = d_0 \cdots d_q$</td>
</tr>
<tr>
<td>pair</td>
<td>two access patterns of shape $(a_0, \ldots, a_n)$</td>
<td>$((a_0, \ldots, a_n))$</td>
</tr>
</tbody>
</table>

Table 2. Glenside’s access pattern operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduceSum</td>
<td>$(\ldots) \rightarrow ()$</td>
<td>sum values</td>
</tr>
<tr>
<td>reduceMax</td>
<td>$(\ldots) \rightarrow ()$</td>
<td>max of all values</td>
</tr>
<tr>
<td>dotProd</td>
<td>$(t, s_0, \ldots, s_n) \rightarrow ()$</td>
<td>eltwise mul; sum</td>
</tr>
</tbody>
</table>

Now to compute the dot product of these row-column pairs.

In Glenside, the dotProd operator (see Table 2) does just that.
(transpose)
(squeeze)
(compute dotProd)
(cartProd)
(windows)
(access activations 1)
(shape C Kh Kw)
(access weights 1)
(list 0 3 1 2))

(a) 2D convolution.

Figure 2. Common tensor kernels from machine learning expressed in Glenside. Lines containing access patterns are annotated with their access pattern shape. \(N\) is batch size; \(H/W\) are spatial dimension sizes; \(C/O\) are input/output channel count; \(K_h/K_w\) are filter height/width; \(S_h/S_w\) are strides.

where \(n\) indexes the output batch, \(o\) indexes output channels, \(x/y\) index spatial dimensions, \(dx/dy\) index the convolutional window spatial dimensions, and \(c\) indexes input channels. 2D convolution slides each of the \(o\) filters of shape \((c, dx, dy)\) through each possible \((c, dx, dy)\)-shaped window of the input images. At each of these locations, an elementwise multiplication and reduction sum is computed.

The Glenside specification of conv2d is shown in Figure 2a. We access the weights as a vector of \(O\) filters and the activations as a vector of \(N\) images. We leave the filters as they are, but form windows of shape \((C, K_h, K_w)\) over the activations using the windows access pattern transformer (Table 1). This produces an access pattern of shape \((N, 1, H', W')\), \((C, K_h, K_w))\), i.e., a batch of “images” of new spatial shape \((H', W')\), where every location is a window of the original input. Finally, we take the Cartesian product of the filters and the windows, compute their dot product, and squeeze and transpose the output into the correct layout.

Max Pooling. Max pooling, commonly used in ML to condense intermediate activations (“act” below), is defined as:

\[
\text{out}[n, c, x, y] = \max_{dx, dy} (\text{act}[n, c, \text{strides}[0] \cdot x + dx, \text{strides}[1] \cdot y + dy])
\]

Max pooling slides a window of shape \((dx, dy)\) over all possible locations within the spatial (i.e., \(x\) and \(y\)) dimensions. At each window location, it reduces the window to a scalar with the max operator. The Glenside specification merely applies reduceMax over each two-dimensional window.

Discussion. Glenside separates the computation from the data access patterns in these kernels while exposing the simplicity of their computation—and the relative complexity of their data access. In all three kernels, the computation can be described with a single operator; most of the specification entails setting up the data access pattern.

Furthermore, Glenside exposes similar structure between kernels; for example, both conv2d and matrix multiplication feature the expression \((\text{compute dotProd} (\text{cartProd } \text{?a0 } \text{?a1})) \implies \text{(systolicArray } \text{?rows } \text{?cols}) \text{?a0} (\text{access (transpose } \text{?a1 (list 1 0))} 0)\) where \?a0 is of shape \((\text{?batch}, (\text{?rows}))\) and \?a1 is of shape \((\text{?cols}, (\text{?rows}))\)

Figure 3. Our rewrite rewriting matrix multiplication to a systolic array invocation.

5.2 Mapping matMul to Accelerators

Glenside can be used to uncover opportunities to invoke accelerator components. Consider a weight-stationary systolic array, a common matrix multiplication architecture. A weight-stationary systolic array with \(r\) rows and \(c\) columns takes two lists of length-\(r\) vectors (the activations and
we show how a few general rewrites within Glenside lead to which enables computing conv2d we print below our rewrites.

weights, respectively), pairing each vector from one list with each vector from the other, and computes a dot product over each pair. The second list contains c vectors, while the first can be of any length.

Glenside’s purity allows us to implement this hardware mapping task using a term rewriting system, in which we rewrite a matching program pattern to an invocation of our systolic array. Our rewrite is shown in Figure 3, mimicking egg’s rewrite syntax. Tokens starting with a question mark (such as ?a0 in Figure 3) are variables in the pattern, bound by the left-hand side (LHS), and then used on the right-hand side (RHS). egg also allows for conditions on rewrites, which we print below our rewrites.

To design our rewrite, we first must design the LHS to match program patterns that resemble the data access pattern and compute pattern of our systolic array. Glenside is eminently suitable for this task, as it can express exactly the data access and computation pattern we described for the systolic array. Pairing all vectors from one list with all vectors from another and computing the dot product of the pairs is represented as (compute dotProd (cartProd ?a0 ?a1)), binding ?a0 and ?a1 to the input access patterns. We encode the systolic array’s constraints on the input shapes as a condition on the rewrite. Patterns which match the LHS are mapped to the RHS; in this case, we introduce a new systolicArray construct to represent the functioning of our systolic array. The shape of the systolic array is given by the ?rows and ?cols parameters, and the inputs are given as access patterns. Note how we also transform the second access pattern to more accurately convey how the actual systolic array hardware accesses the weight tensor: it reads it all at once (hence, (access ... 0)), and expects it to be laid out in transposed form in memory. This added information—enabled by Glenside’s access patterns—provides richer data layout information, potentially helping future rewrites or code generation steps.

5.3 Flexible Mapping: Discovering im2col

The im2col transformation is a data layout optimization which enables computing conv2d on matrix multiplication hardware. The transformation involves instantiating the convolutional windows over the input activations directly in memory [7]. This leads to data duplication, but the resulting speedup more than offsets that overhead. In this case study, we show how a few general rewrites within Glenside lead to the automatic rederivation of the im2col transformation.

![Figure 4](image1.png)

Figure 4. Rewrites enabling the discovery of the im2col transformation.

![Figure 5](image2.png)

Figure 5. An im2col-transformed conv2d, after the application of the rewrites in Figure 4 and just before the application of the systolic array rewrite.

Glenside’s representation underscores the structural similarity between conv2d and matrix multiplication, reflected also by the shared (compute dotProd (cartProd ...)) between conv2d and the LHS of the systolic array rewrite in Figure 3. Using this rewrite on conv2d would permit mapping it to the systolic array; however, the restrictions on the shape of ?a0 and ?a1 prevent its application. The systolic array has an activation access pattern of shape ((N,1,H',W',O'), ()) and a weight access pattern of shape ((c), (d)), while conv2d operates over access patterns of shape ((N,1,H',W'), (C,Kh,Kw)) and of ((O), (C,Kh,Kw)), respectively. Transforming the access pattern into a lower-dimensional form would enable the systolic array rewrite.

Figure 4 shows the rewrites which enable this transformation. We call the first rewrite an exploratory rewrite as it optimistically matches any access pattern expression. It flattens an access pattern and immediately reshapes it back to its original shape, thus preserving equality (see Table 1 for formal definitions). This exploratory rewrite introduces the flattening necessary to convert the higher-dimensional access patterns of conv2d into the access patterns matched by the systolic array rewrite. However, the reshape operator will still need to be moved before we can fire Figure 3’s systolic array rewrite. The second and third rewrites in Figure 4 take care of this; they implement composition commutativity of reshape with cartProd and compute dotProd, which “bubble” reshape operators up and out of expressions. These rewrites express general properties of these operators and are not specific to this task.
if \(?\dim\) is an access dimension

\[
\text{concat} \ (\text{cartProd} \ ?a \ ?b \ ?c) \ \Rightarrow \\
\text{concat} \ (\text{cartProd} \ ?a) \ (\text{cartProd} \ ?b) \ (\text{cartProd} \ ?c)
\]

if \(?\dim\) and \(?\dim1\) are the same shape dimension

\[
\text{compute dotProd} \ (\text{concat} \ ?a \ ?b) \ \Rightarrow \\
\text{compute dotProd} \ (\text{concat} \ ?a) \ (\text{concat} \ ?b)
\]

if \(?\dim\) is an access dimension

\[
\text{compute dotProd} \ (\text{concat} \ ?a \ ?b) \ \Rightarrow \\
\text{compute reduceSum} \ (\text{pair} \ (\text{compute dotProd} \ ?a) \ (\text{compute dotProd} \ ?b))
\]

if \(?\dim\) is a shape dimension

\textbf{Figure 6.} Rewrites for blocking matMul.

These three rewrites work in concert to map conv2d to a systolic array. First,\textsuperscript{6} the exploratory rewrite flattens and reshapes all access pattern expressions. This includes the inputs to conv2d’s cartProd subexpression, which are flattened to shapes \((N \cdot H' \cdot W'\), \((C' \cdot K_h' \cdot K_w'))\) and \((O), \((C' \cdot K_h \cdot K_w))\) and reshaped back to their original shapes. Next, the composition commutativity rewrites for cartProd and compute dotProd fire in sequence, bubbling the reshape up through the cartProd and dotProd expressions (shown in Figure 5). Finally, the systolic array rewrite completes the im2col transform. Glenside’s equality saturation based rewrite engine discovers these rewrites as the exploratory rewrite fires on every term and no rewrites are missed due to the absence of phase ordering.

This example highlights how, with straightforward, generally applicable rewrites defined over Glenside, equality saturation can emergently discover useful transformations that previously required expert insights to apply.

5.4 Flexible Mapping: matMul Blocking

Equality saturation can also be used with Glenside to emergently discover a matrix multiplication blocking scheme. Matrix multiplication blocking is the common strategy of breaking up a single, large matrix multiplication into smaller multiplications, by multiplying subsets of the input matrices and assembling the results to form the output matrix. This is essential in practice, as systolic arrays are small (often between 16 × 16 and 256 × 256) while matrices in ML and HPC applications can be much larger.

As in Section 5.3, this transformation follows from an exploratory rewrite and associated “cleanup” rewrites. The exploratory rewrite used for blocking is shown at the top of Figure 6. Given an access pattern, this rewrite slices the access pattern into two pieces along a dimension and then concatenates them back together. The dimension as well as the division strategy are configurable. For this example, we assume for simplicity that we run this rewrite on every available dimension, that we divide each dimension perfectly in half, and that all dimensions are powers of 2 in size. Figure 6 gives rewrites for bubbling the introduced concat operators up through the expression, namely the compositional commutativity of concat with cartProd and compute dotProd. Starting from the matrix multiplication in Figure 2b, assuming input shapes of \((32, 32)\), the exploratory rewrite first slices and concatenates the access patterns at the input of cartProd. Then, using the commutativity rewrites, the resulting concatenates are bubbled up to produce the final expression in Figure 7. The effect of these rewrites is that the single \(32 \times 32\) matMul becomes eight separate \(16 \times 16\) matMuls, which are summed and concatenated to form the full output matrix. This case study demonstrates yet again that Glenside’s expressiveness allows a small set of rewrites to produce interesting and useful emergent transformations.

\textbf{Figure 7.} A \(32 \times 32\) matMul blocked into \(16 \times 16\) matMuls. Only two of the eight total multiplications are shown.

\textsuperscript{6}Since equality saturation explores rewrites non-destructively, the rewriting order here is purely for explanatory purposes.
6 Conclusion

In this paper, we proposed access patterns as an abstraction to enable equality saturation style term rewriting for low-level tensor program mapping to hardware accelerators. Crucially, access patterns support specifying and composing higher-order, arbitrary dimension tensor operators without the need for binding structures like anonymous functions or index notation. We demonstrated the potential utility of access patterns in the Glenside IR through case studies showing how rewrites in Glenside can automatically uncover common layout transformations like im2col used for accelerator mapping. We are excited for the community to join in further exploring the potential applications of access patterns and to build additional optimizations on Glenside’s foundations.

Acknowledgments

This work was sponsored by the Real Time Machine Learning (RTML) DARPA project, and the Applications Driving Architectures (ADA) and Center for Research in Intelligent Storage and Processing in Memory (CRISP) JUMP Research Centers, co-sponsored by SRC and DARPA. We thank Chandrakana Nandi for her extensive and enthusiastic editing. We also thank Sorawee Porncharoenwase, Luis Vega, and Max Willsey for their helpful comments.

A Broader Impact Statement

The ability to develop effective compiler support for specialized hardware accelerators in ML, and HPC more broadly, has generally been restricted to a handful of elite, well-resourced teams. This restriction slows hardware development and creates barriers to entry for teams in less privileged environments to contribute to and help guide the development of the field.

We believe that the access pattern abstraction and Glenside’s approach to term rewriting for improving compiler support for custom accelerators will help advance both near- and longer-term principled approaches to building flexible compiler infrastructure. In turn, we hope that this infrastructure will help contribute to a broader, more diverse, and more inclusive community of folks working together to build efficient technologies for social good.

Of course, all technology is political and it can be difficult to anticipate how future researchers and practitioners may apply Glenside. While the most obvious consequence of more efficient hardware utilization is better performance for users and lower environmental impact via decreased power consumption, it is also possible that access patterns and Glenside would enable the rapid obsoleting of current hardware platforms and therefore contribute to harmful electronic waste. This work could also stimulate demand for hardware customization by removing compiler development–related overheads and ultimately lead to higher negative environmental impact similar to the situation with respect to custom ASICs for bitcoin mining [29].

Also, any improvement to ML efficiency or applicability may contribute to economic and privacy concerns arising from increased technology company monopolization as discussed in Zuboff’s The Age of Surveillance Capitalism [46].

References


