SkeleShare: Algorithmic Skeletons and Equality Saturation for Hardware Resource Sharing

Jonathan Van der Cruysse*, Tzung-Han Juang*, Shakiba Bolbolian Khah*, Christophe Dubach*†
*McGill University, Montréal, Canada †Mila, Montréal, Canada

***Abstract*—Compiling functional programs into efficient Field Programmable Gate Array (FPGA) designs is difficult. Hardware resources must be explicitly allocated and shared to maximize resource efficiency. This requires careful orchestration of several transformations to expose and exploit sharing opportunities.**

This paper introduces SkeleShare, a novel approach that automates the problem of resource allocation and sharing. It leverages equality saturation and algorithmic skeletons to expose sharing opportunities across abstraction levels. A solver-based extractor then selects a design that consolidates computations, meeting resource constraints while maintaining performance.

This approach is evaluated on neural networks and image processing targeting a real FPGA. The paper shows how SkeleShare is used to express the various algorithmic patterns and transformation rules inherent in neural network operators. The experimental evaluation demonstrates that SkeleShare's fully automated resource allocation and sharing matches and exceeds the performance of prior work, which involves expert manual extraction of sharing opportunities.

***Index Terms*—FPGA compilation, Functional programming, Resource allocation, Resource sharing, Equality saturation, Algorithmic skeletons, Multilevel intermediate representation, E-Graphs, Solver-based extraction, Neural network hardware**

I. Introduction

Compiling functional programs to Field-Programmable Gate Arrays (FPGAs) is challenging due to the need for explicit hardware resource management. Naive compilation uses resources inefficiently. Two techniques address this: *resource allocation* assigns hardware to program parts within a resource budget, and *resource sharing* consolidates computations into shared hardware units [14, 15]. These techniques interact. To illustrate this, consider the following slice of a neural network:

```
1 mv^(64·2·2)×64 (fcWeights,
2   flatten(conv^1024×1024×64×64×3 (convWeights, input)))
```

This expression applies a convolution, flattens the result, and computes a matrix-vector product representing a fully connected layer. A typical compiler maps each operation to a separate hardware unit [9, 23, 24], forming a pipeline where convolution and matrix-vector product compete for resources.

To mitigate such resource contention, three transformations create sharing opportunities: **Lowering** exposes structure; **padding** aligns dimensions to enable reuse; and **tiling** partitions inputs for reuse [14, 15]. These can be applied in various sequences. The convolution could first be lowered into a `map` with a matrix-vector product (**mv**) over a sliding window:

```
1 let c = map(λws. map(λw.
2   mv^(64·3·3)×64 (convWeights, w), ws), slide2D(3, input))
3 mv^(64·2·2)×64 (fcWeights, flatten(c))
```

then, the inputs to the matrix-vector product on line 3 padded:

```
1 let c = map(λws. map(λw.
2   mv^(64·3·3)×64 (convWeights, w), ws), slide2D(3, input))
3 let paddedFCWeights = map(λrow. pad(320, row), fcWeights)
4 mv^(64·3·3)×64 (paddedFCWeights, pad(320, flatten(c)))
```

This yields duplicate calls to the same matrix-vector product, which can then be shared:

```
1 let sharedMV = mv^(64·3·3)×64
2 let c = map(λws. map(λw.
3   sharedMV(convWeights, w), ws), slide2D(3, input))
4 let paddedFCWeights = map(λrow. pad(320, row), fcWeights)
5 sharedMV(paddedFCWeights, pad(320, flatten(c)))
```

However, finding such transformations is non-trivial and early lowering may unintentionally inhibit back-end optimizations relying on high-level structure. The transformation space is vast, and beneficial sharing patterns often emerge only after coordinated rewrites across abstraction levels. The compiler must therefore select the right sequence of transformations to identify cross-level sharing opportunities while postponing lowering to preserve back-end optimizations.

This paper presents a solution in the form of SkeleShare, the first fully automated technique to jointly solve resource allocation and sharing for high-level functional programs targeting FPGAs. Unlike prior work that relies on manual rewriting [15], SkeleShare integrates a multilevel Intermediate Representation (IR), equality saturation, and a solver-based extractor that selects optimized variants based on cost and resource constraints. Lowering is deferred until after extraction, preserving structure critical for back-end optimization.

SkeleShare is evaluated on the VGG and TinyYolo deep neural networks, a self-attention layer, and an image processing pipeline, each compiled to an Intel Arria 10 FPGA. The system expresses these programs using algorithmic skeletons and applies transformation rules that enable cross-layer reuse. Results show that SkeleShare matches the performance of prior hand-optimized baselines.

The contributions of this paper are as follows:

- Multi-abstraction e-graph rewriting, combining skeletons, multi-level IR, equality saturation, and deferred lowering.
- A solver-based e-graph extraction algorithm that jointly reasons about allocation and sharing.
- An instantiation of the SkeleShare interface for the domain of neural networks and image processing on FPGAs.
- An empirical evaluation of SkeleShare on real hardware, showing strong performance on a range of benchmarks.

979-8-3315-9288-2/26 © 2026 IEEE

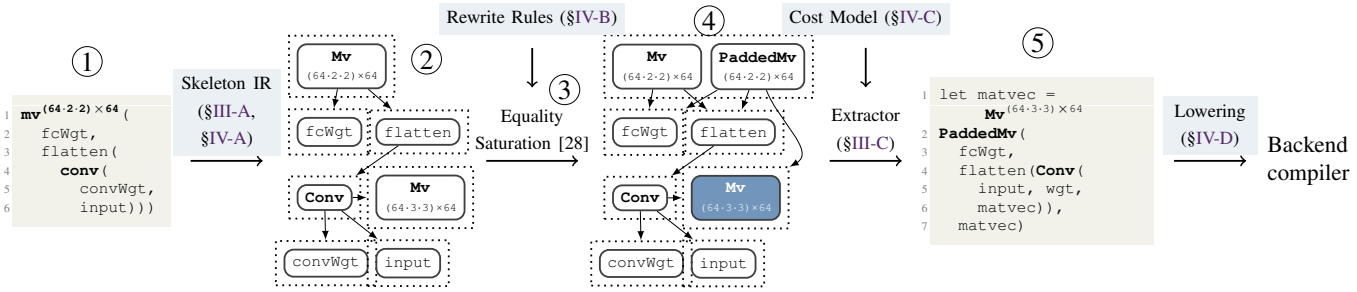


Fig. 1: Overview of SkeleShare, the proposed technique. Program ① is converted to skeleton IR e-graph ②. Equality saturation applies transformation rules ③ to e-graph ②. After rule application, we obtain e-graph ④. The blue-highlighted **`Mv`** has multiple uses and is hence a candidate for sharing. From e-graph ④, an outlining extractor chooses expression ⑤. Superscripts and e-node annotations denote **`Mv`** input dimensions. Light blue boxes are domain-specific components.

II. Overview

The resource allocation and sharing problem in high-level functional programs poses two challenges:

1) **Transformation sequencing:** Finding a sequence of rewrites that maximizes sharing and fits resource budgets.
2) **Cross-level sharing:** Detecting opportunities that span multiple levels of abstraction.

SkeleShare addresses both using *multi-abstraction e-graph rewriting*. Illustrated by Figure 1, rewriting combines a skeleton IR, equality saturation, and solver-guided extraction:

- **Translation:** The input program ① is translated into a multilevel skeleton IR and encoded in an e-graph ② [8, 19, 20, 32]. In the figure, the skeleton equivalents of **`mv`** and **`conv`** are **`Mv`** and **`Conv`**; the relationship between **`Mv`** and **`Conv`** is explicit as the latter takes the former as an argument. This exposes composition while preserving high-level structure needed for back-end optimizations.
- **Saturation:** Rewrite rules ③ are applied to the e-graph, yielding an enriched graph ④ that compactly represents many transformation paths. Concretely, e-graph ④ is enhanced with the finding that the top-level **`Mv`** is equivalent to a padded version of the **`Conv`**'s inner **`Mv`**.
- **Extraction:** A solver-based extractor selects a final expression ⑤, jointly solving for allocation and sharing decisions under resource constraints. In the example, the solver selects the padded version of the top-level **`Mv`** and chooses to share the inner matrix-vector product.
- **Lowering:** The selected program is lowered into back-end–recognizable primitives. Rather than lowering each skeleton in isolation, this process considers composed skeletons to enable further optimization.

This multi-abstraction e-graph rewriting process diverges from standard equality saturation workflows in its use of a multi-level skeleton IR and its consideration of optional sharing during extraction. SkeleShare has a modular interface (light blue in Figure 1) with four domain-specific elements: skeleton IR, rewrite rules, cost model, and lowering strategy.

The following section introduces the core components of SkeleShare. Section IV then applies them to a neural network FPGA compilation use case.

III. The SkeleShare Process

This section presents SkeleShare, a transformation framework that uses algorithmic skeletons to express computations across multiple abstraction levels. Complex operations are structured as compositions of higher-order skeletons, each parameterized by lower-level functions, enabling modular and composable transformations. Equality saturation systematically explores equivalent program variants within an e-graph, deferring decisions until extraction, which jointly optimizes resource allocation and sharing under hardware constraints. A final lowering stage then converts skeletons into concrete implementations.

A. Skeletons

SkeleShare represents computations using a skeleton-based IR, designed to expose transformation opportunities across multiple abstraction levels. The syntax of this IR is:

$$
\begin{array}{rll}
e ::= & x & \text{(variable)} \\
\mid & \lambda x.\, e & \text{(abstraction)} \\
\mid & \mathtt{Sk}(e_1, \ldots, e_n) & \text{(skeleton)} \\
\mid & e_1\, e_2 & \text{(application, post-extraction only)}
\end{array}
$$

Here, x ranges over variables, and **`Sk`** over domain-specific skeletons. Skeletons define high-level operations that can be decomposed and transformed. Function application is disallowed during equality saturation to restrict higher-order functions to skeletons, making cost modeling tractable. After sharing decisions have been made, application nodes are introduced to model `let` bindings and construct the final program. For the neural network compilation use case examined in this paper, the full set of skeletons is introduced in Section IV-A.

The skeleton IR is derived through a mechanical translation from the input Domain-Specific Language (DSL): each DSL operator is replaced with one or more skeletons. This translation preserves program structure while making intermediate abstractions explicit and transformable.

Although not the focus of this work, the IR is typed using a structural type system. Types are mostly left implicit throughout the paper for readability but are enforced in the implementation.

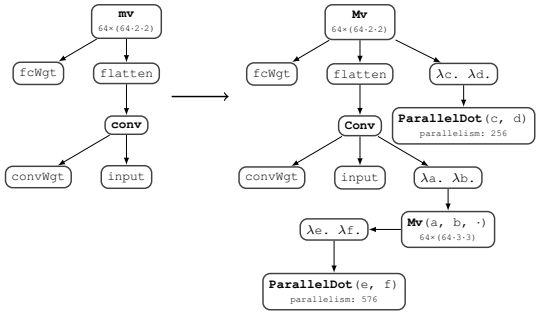


Fig. 2: A tree representation of the running example **mv**(fcWgt, flatten(**conv**(convWgt, input))) on the left, alongside its skeleton IR tree on the right.

The design of skeletons builds on the concept of algorithmic skeletons [1, 6, 36], which capture reusable patterns such as `map` and `fold` in parallel programming. In contrast to traditional skeletons that abstract over *what* is computed, SkeleShare skeletons capture *how* computations are implemented.

Concretely, SkeleShare skeletons are ordinary IR operations that are defined hierarchically: higher-level skeletons take lower-level ones as arguments. This design makes the IR simultaneously higher-level and lower-level, enabling transformations across abstraction boundaries without lowering. For example, **conv** in the DSL translates to skeleton IR:

```
Conv(in, wgt, λa.λb.
    Mv(a, b, λc.λd. ParallelDot[P](c, d)))
```

This form reveals a three-level hierarchy: a convolution is implemented in terms of matrix-vector products, which themselves are decomposed into dot products, run in parallel. Because skeletons are modular, the inner computation can be replaced without changing the surrounding structure. For example, the matrix-vector product can be padded like so:

```
Conv(in, wgt, λa.λb.
    PaddedMv(a, b, λpa.λpb.
        Mv(pa, pb, λc.λd. ParallelDot[P](c, d))))
```

Figure 2 illustrates this translation on a concrete example. The left side shows the original DSL expression, while the right side presents the corresponding skeleton IR. Each high-level operator is replaced by skeletons: `mv` becomes **Mv** and `conv` becomes **Conv**. This structure enables transformation and sharing at multiple levels: rewrites can target skeletons directly rather than reconstructing low-level patterns, and high-level structure is preserved through extraction, avoiding premature decisions that may interfere with back-end optimizations. A flattened IR, by contrast, would require complex, brittle rules to match nested patterns and would preempt back-end optimizations by locking in low-level structure too early.

This skeleton IR enables transformation and sharing at multiple levels. SkeleShare explores rewrites through equality saturation and selects an optimized configuration via solver-guided extraction.

B. Equality Saturation and Transformation

Equality saturation is a general optimization technique that builds an e-graph, a data structure that compactly represents many semantically equivalent expressions [8, 19, 20, 28, 32]. An e-graph consists of e-classes (equivalence classes of terms) and e-nodes (individual operations), where each e-node references child e-classes. Unlike traditional compilers, which apply transformations in a fixed order, equality saturation accumulates alternatives and defers decision-making.

SkeleShare applies this technique to a skeleton-based IR, using domain-specific rewrite rules to expose trade-offs in resource sharing and allocation. Initially, the e-graph mirrors the structure of the input program. As shown in Figure 3, expression tree ① is translated into the skeleton IR and encoded as e-graph ②.

Rewrite rules then expand the e-graph without discarding existing terms, producing a saturated structure that compactly encodes many transformation variants. In Figure 3, the enriched e-graph ③ captures two key opportunities:

1) **Resource sharing via padding:** The top-level **Mv** can be replaced with **PaddedMv**, aligning its shape with another matrix-vector product and enabling hardware reuse.
2) **Resource allocation via parallelism:** Multiple variants of **ParallelDot** express different degrees of parallelism, balancing throughput against DSP usage.

By deferring commitment, the e-graph preserves trade-offs between performance and hardware cost. These choices are resolved during extraction, described next.

C. Extraction

After saturation, the extractor selects a single expression from the e-graph, resolving all allocation and sharing decisions as a joint optimization problem. This section encodes these decisions as Boolean variables and equations, searched by an off-the-shelf solver (Z3 [7]) for a minimal-cost variable assignment. To improve scalability, SkeleShare agglomerates e-classes to reduce e-graph size before extraction.

Following prior work [25, 28, 31], extraction is modeled as a Boolean decision over each e-node and e-class—whether it appears in the final expression. That formulation is extended here to support multidimensional cost models and sharing decisions. As extraction is NP-hard, SkeleShare bounds solver time and returns the best solution found within a timeout.

1) Variables: The extractor introduces the following Boolean variables [10]:

- $select_n$: whether e-node n is selected.
- $select_c$: whether any e-node in e-class c is selected.
- $share_c$: whether c is shared across uses or duplicated.

These variables define the subgraph to be extracted and the corresponding sharing plan. Figure 4 illustrates the result of selection and sharing decisions. Selected components are colored normally; shared e-classes appear in blue; unselected elements are grayed out.

Expression construction is driven by solver output: shared e-classes are outlined via `let` bindings, while unshared ones

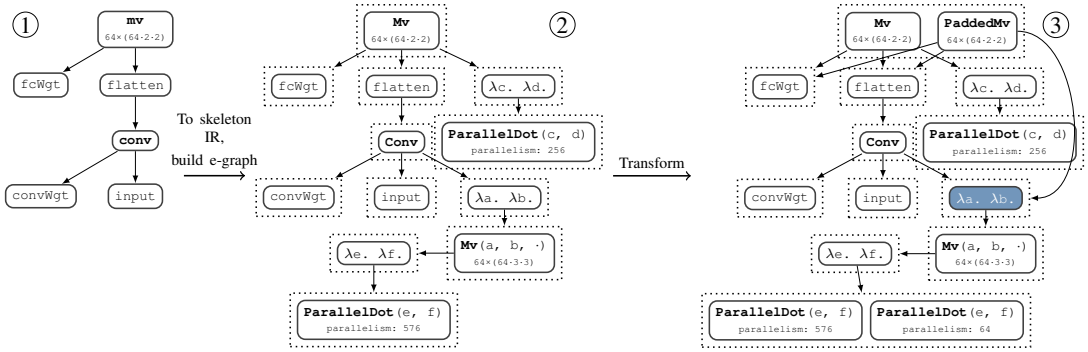


Fig. 3: E-Graph construction and transformation for **mv**(fcWgt, flatten(**conv**(convWgt, input))). Expression tree ① is translated to skeleton IR and encoded as e-graph ②. Rewrite rules expand the graph to ③, introducing a padded **Mv** and multiple **ParallelDot** variants. The top e-class (dotted box) includes both, exposing reuse and performance–cost tradeoffs. Shareable components are in blue; solid boxes are e-nodes, dotted boxes are e-classes.

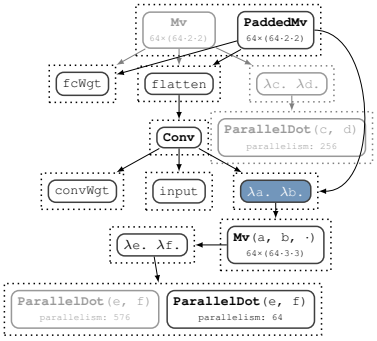


Fig. 4: Example selection and sharing decisions in an e-graph. The sole e-node of a shared e-class is highlighted in blue. Selected e-nodes and e-classes retain their usual color. Non-selected e-nodes and e-classes appear grayed out.

are duplicated. If a shared e-class contains free variables, it is abstracted into a lambda. This process yields a single expression suitable for hardware generation. For the example in Figure 4, extraction produces:

```
let matvec = λa. λb.
  Mv(a, b, λe. λf. ParallelDot[64](e, f))
PaddedMv(
  fcWgt, flatten(Conv(input, convWgt, matvec)), matvec)
```

2) Constraints: Two constraint classes govern extraction: well-formedness and cost.

a) Well-Formedness: Extraction must yield a single acyclic expression. The following constraints enforce this:

- **E-class/e-node consistency:** Every selected e-class must select an e-node:

$$select_c \implies \bigvee_{n \in nodes(c)} select_n$$

- **Argument dependencies:** Selected e-nodes require their arguments to be selected:

$$select_n \implies \bigwedge_{c \in args(n)} select_c$$

- **Root selection:** The root e-class r must be selected:

$$select_r = 1$$

- **Acyclicity:** Expressions must be acyclic, even if the underlying e-graph is not. A topological labeling enforces this [13]:

$$\forall c' \in args(n).\, select_n \implies label_c < label_{c'}$$

b) Cost: A domain-specific cost model governs selection and sharing decisions across one or more resource dimensions. The model associates each e-node n with:

- An $inplace\_cost_i(n)$: cost when n is duplicated.
- A $shared\_cost_i(n)$: cost when n is shared.

From these e-node costs, e-class costs are derived:

$$use\_cost_i(c) = \begin{cases} \sum_n select_n \cdot shared\_cost_i(n) & \text{if } share_c = 1 \\ \sum_n select_n \cdot inplace\_cost_i(n) & \text{otherwise} \end{cases}$$

$$fixed\_cost_i(c) = \begin{cases} \sum_n select_n \cdot inplace\_cost_i(n) & \text{if } share_c = 1 \\ 0 & \text{otherwise} \end{cases}$$

The total cost of each dimension is a domain-specific aggregation (*e.g.,* a sum) of all fixed costs and the use cost of the root e-class. The cost model uses these total costs in its optimization problem, such as minimizing runtime while constraining hardware usage. An example cost model is detailed in Section IV-C.

D. Lowering

After extraction, the skeleton IR is lowered into back-end–recognizable primitives. A straightforward approach expands each skeleton independently into functional constructs. For instance, a convolution skeleton can be translated mechanically:

```
Conv(in, wgt, mvFun) = map(λws.
  map(λwin. mvFun(wgt, win), ws), slide2D(N, in)).
```

While sufficient for isolated cases, independent lowering can introduce unnecessary buffers in composed transformations (*e.g.,* padding followed by tiling), hurting performance.

To address this, SkeleShare relies on a structured lowering process (Section IV-D) that analyzes groups of composed skeletons. It reorders, fuses and rewrites them to reduce memory usage and expose back-end–optimizable patterns, enabling optimizations such as buffer elimination.

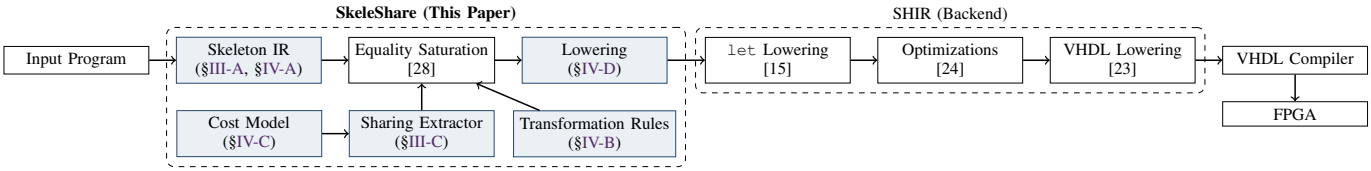


Fig. 5: End-to-end compiler stack integrating SkeleShare with the SHIR pipeline. Blue boxes show SkeleShare's contributions. These produce a skeleton IR that is lowered to SHIR's mid-level primitives and synthesized by a standard FPGA toolchain.

E. Summary

This section presented SkeleShare's end-to-end pipeline for optimizing skeleton-based representations. Computations are expressed as nested skeletons, equality saturation explores equivalent variants, the extractor selects and shares expressions, and lowering generates hardware-efficient code.

SkeleShare departs from standard equality saturation workflows in three ways: 1) the multilevel skeleton IR preserves high-level structure, 2) the extractor reasons jointly about sharing and allocation, and 3) the lowering stage transforms skeletons to lower-level primitives.

IV. Use Case: Neural Networks on FPGAs

This section applies SkeleShare to a common FPGA target: compiling neural networks from functional programs. This setting features hard device-specific limits on Digital Signal Processors (DSPs) and Random Access Memory (RAM) [23, 24], where manual resource sharing can determine whether a program fits [14, 15]. Figure 5 shows how SkeleShare integrates with SHIR [23]. The remainder of this section details its domain-specific components: the skeleton IR (§IV-A), rewrite rules (§IV-B), cost model (§IV-C), and lowering (§IV-D).

A. Linear Algebra Skeletons

As a refinement of the general skeleton-based IR introduced in Section III-A, this section details the specific skeletons used to represent and transform across multiple abstraction levels the linear algebra computations that underpin neural networks. The design follows the same principles laid out earlier: compositional structure, transformation hooks, and abstraction over implementation. As a general rule, skeletons are added for each computation that need to be reshaped by rewrite rules for resource sharing; they serve as hooks for high-level transformations. In this section, there are three groups of linear algebra skeletons—core, data-transformed, and partial computations—summarized in Table I.

1) Core Computations: The first class of skeletons captures four fundamental computations: dot products, matrix-vector products, matrix-matrix products, and neural-network–style three-dimensional convolutions. Dot products serve as the atomic operation, matrix-vector products combine them to implement fully connected layers or matrix-matrix products, and convolutions generalize these patterns across spatial dimensions. The types of these skeletons are:

TABLE I: Overview of skeleton IR operators.

Operator	Description
Core Computations	
ParallelDot[P](a, b)	P parallel dot products
Mv(mat, vec, dotF)	Matrix-vector prod. using dotF
Mm(a, b, mvF)	Matrix-matrix prod. using mvF
Conv(in, wgt, mvF)	3D convolution using mvF
Data-Transformed Computations	
TiledMv(mat, vec, mvF)	Tiled matrix-vector product
PaddedMv(mat, vec, mvF)	Padded matrix-vector product
TiledMmW(a, b, mmF)	Tiled matrix product (width)
TiledMmH(a, b, mmF)	Tiled matrix product (height)
TiledMmK(a, b, mmF)	Tiled matrix product (inner)
TiledConvHW(in, wgt, convF)	Tiled convolution (spatial)
TiledConvICH(in, wgt, convF)	Tiled input channels
TiledConvOCH(in, wgt, convF)	Tiled output channels
PaddedConvHW(in, wgt, convF)	Padded spatial dimensions
PaddedConvICH(in, wgt, convF)	Padded input channels
Partial Dot Products and Parallel Convolutions	
MergePartialSum(psum)	Combines chunked dot results
ParallelPartialDot[P, C](a, b)	Chunked parallel dot products
ParallelConv(in, wgt, mvF)	Convolution with chunked mvF

$$\mathit{ParallelDotT}\ [\mathtt{M},\mathtt{N},\mathtt{T},\mathtt{T}'] = \mathtt{M}\times\mathtt{N}\times\mathtt{T}\to\mathtt{M}\times\mathtt{N}\times\mathtt{T}\to\mathtt{M}\times\mathtt{T}'$$
$$\mathit{MvT}\ [\mathtt{M},\mathtt{N},\mathtt{T},\mathtt{T}'] = \mathtt{M}\times\mathtt{N}\times\mathtt{T}\to\mathtt{N}\times\mathtt{T}\to\mathtt{M}\times\mathtt{T}'$$
$$\mathit{MmT}\ [\mathtt{W},\mathtt{H},\mathtt{K},\mathtt{T},\mathtt{T}'] = \mathtt{H}\times\mathtt{K}\times\mathtt{T}\to\mathtt{W}\times\mathtt{K}\times\mathtt{T}\to\mathtt{H}\times\mathtt{W}\times\mathtt{T}'$$
$$\mathit{ConvT}\ [\mathtt{W},\mathtt{H},\mathtt{ICH},\mathtt{OCH},\mathtt{KS},\mathtt{T},\mathtt{T}'] = \mathtt{H}\times\mathtt{W}\times\mathtt{ICH}\times\mathtt{T}\to \mathtt{OCH}\times\mathtt{KS}\times\mathtt{KS}\times\mathtt{ICH}\times\mathtt{T}\to (\mathtt{H}-\mathtt{KS}+1)\times(\mathtt{W}-\mathtt{KS}+1)\times\mathtt{OCH}\times\mathtt{T}'$$

Here, $\times$ denotes an array type and $\to$ a function type. Bracketed names are explicit type parameters. T and T′ represent arbitrary types; all others are compile-time integers. While types are generally implicit in the paper, they are enforced by a structural type system in the implementation.

SkeleShare leverages the compositional relationship between these skeletons: convolutions and matrix-matrix products are constructed from matrix-vector products, which are built from dot products. This hierarchy is captured in the syntax and semantics of the corresponding operators in Figure 6.

The compositional structure enables nested transformations:

```
Conv(input, wgt, λa. λb.
  Mv(a, b, λc. λd. ParallelDot[64](c, d)))
```

Each component can be independently replaced. For instance, modifying the degree of parallelism in **ParallelDot** changes the hardware instantiation: **ParallelDot**[64] computes 64 dot products in parallel; increasing this to 128 doubles DSP usage and halves latency. Such changes propagate

```
ParallelDot[P](a: M × N × int, b: M × N × int): M × int
Mv(mat: M × N × T, vec: N × T, dotF: ParallelDotT [M, N, T, T′]): M × T′
Mm(a: H × K × T, b: W × K × T, mvF: MvT [W, K, T, T′]): H × W × T′
Conv(in:H × W × ICH × T, wgt:OCH × KS × KS × ICH × T, mvF:MvT [OCH, KS · KS · ICH, T, T′]):(H − KS + 1) × (W − KS + 1) × OCH × T′
```

Fig. 6: Core skeletons for parallel dot products, matrix-vector products and dot products. By convention, type parameters that can be inferred from arguments are kept implicit. **ParallelDot** is defined over integer inputs, the typical scalar type for FPGAs, while the others are polymorphic. This polymorphism enables flexible reuse, used by partial computation skeletons.

through the hierarchy, adjusting the implementation of **Mv** and **Conv** without altering their structure.

2) Data-Transformed Computations: While changing **ParallelDot** parameters controls parallelism, transforming the inputs of **Mv** and **Conv** enables resource sharing through tiling and padding. Tiling partitions data into smaller, fixed-size regions—such as spatial blocks or channel subsets—that are processed independently. This enables a large computation to reuse the same hardware originally designed for a smaller instance, simply by applying it repeatedly across tiles. Padding, by contrast, increases input sizes to match a larger reference shape. This allows a smaller computation to reuse the same hardware as a larger one by embedding itself into the larger input domain.

In SkeleShare, these transformations are expressed as skeletons. Each skeleton mirrors its untransformed counterpart but adds a function parameter that operates over the modified input dimensions. The operators in Figure 7 define the types of these data-transformed skeletons. Highlighted type parameters (*e.g.,* `TM`, `PN`, `TH`, `TW`) define the tiled or padded dimension. Tiled parameters must divide the original size; padded parameters must exceed it. Figure 8 illustrates these transformations.

These skeletons perform data restructuring only: they manipulate input dimensions through tiling or padding but preserve the core computation. Each skeleton applies its transformation and then delegates execution to a user-supplied function that computes the same operation as before, only over a smaller tile or a padded input. This decoupling of transformation from computation supports modular composition, enabling optimizations at different layers.

The effect of that structure is illustrated by this example:

```
let matvec = λa: 64 × (64 · 3 · 3) × int. λb.
  Mv(a, b, λc. λd. ParallelDot[64](c, d))
let matvec′ = λa: 64 × (64 · 2 · 2) × int. λb.
  Mv(a, b, λc. λd. ParallelDot[64](c, d))
matvec′(fcWgt, flatten(Conv(input, convWgt, matvec)))
```

This program contains two nearly identical matrix-vector operations with different input shapes. Compiled directly, they would yield separate hardware designs. To promote reuse, we can rewrite `matvec′` to call `matvec` on padded inputs:

```
let matvec = λa: 64 × (64 · 3 · 3) × int. λb.
  Mv(a, b, λc. λd. ParallelDot[64](c, d))
PaddedMv(
  fcWgt, flatten(Conv(input, convWgt, matvec)), matvec)
```

This reuse reduces hardware duplication and illustrates how padding can unify divergent dimensions into a shared structure.

3) Partial Dot Products and Parallel Convolutions: A common hardware sharing opportunity arises when two convolutional layers differ in input channel count by an integer multiple. For instance, consider a layer with 16 input channels and another with 4: the computation for the larger layer can be decomposed into smaller chunks that align with the smaller one. Indeed, a 16-channel window can be interpreted as four 4-channel windows, enabling a 16-channel dot-product unit to compute multiple 4-channel products in parallel and thereby serve both layers.

To express this, we introduce three skeletons in Figure 9: **ParallelPartialDot** computes chunked dot products, **MergePartialSum** aggregates them, and **ParallelConv** applies a shared matrix-vector function over grouped windows.

These operators satisfy:

```
MergePartialSum(ParallelPartialDot[P, C](a, b))
  = ParallelDot[P](a, b).
```

Moreover, **MergePartialSum** can either directly merge the output of a **ParallelPartialDot** as above or be applied to any expression yielding an array of partial sums, such as a matrix-vector product configured to use a **ParallelPartialDot**:

```
MergePartialSum(Mv(a, b, λc. λd.
    ParallelPartialDot[P, C](c, d))
  = Mv(a, b, λc.λd. ParallelDot[P](c, d)).
```

This delayed merging is well-typed due to the polymorphism of **Mv**, and is useful when convolution channel counts differ by a multiple. For example, consider two convolutions:

```
let c = Conv(input, wgt, λa. λb.
  Mv(a, b, λc. λd. ParallelDot[4](c, d)))
let c′ = Conv(input′, wgt′, λa. λb.
  Mv(a, b, λc. λd. ParallelDot[4](c, d)))
```

Assume `input` has shape $128 \times 128 \times 16 \times$ `int` and `input′` has $128 \times 128 \times 4 \times$ `int`, differing only in channels. Their dot products differ in dimensionality: $3 \times 3 \times 16$ vs. $3 \times 3 \times 4$.

We rewrite both using partial dot products:

```
let shared = λa. λb.
  Mv(a, b, λc. λd. ParallelPartialDot[4, 4](c, d))
let c = Conv(input, wgt,
  λe. λf. MergePartialSum(shared(e, f)))
let c′ = ParallelConv(input′, wgt′, shared)
```

TiledMv(mat: $M \times N \times \texttt{int}$, vec: $N \times \texttt{int}$, mvF: $\mathit{MvT}\,[\texttt{TM}, N, \texttt{int}, \texttt{int}]$): $M \times \texttt{int}$

PaddedMv(mat: $M \times N \times \texttt{int}$, vec: $N \times \texttt{int}$, mvF: $\mathit{MvT}\,[M, \texttt{PN}, \texttt{int}, \texttt{int}]$): $M \times \texttt{int}$

TiledMmW(a: $H \times K \times \texttt{int}$, b: $W \times K \times \texttt{int}$, mmF: $\mathit{MmT}\,[\texttt{TW}, H, K, \texttt{int}, \texttt{int}]$): $H \times W \times \texttt{int}$

TiledMmH(a: $H \times K \times \texttt{int}$, b: $W \times K \times \texttt{int}$, mmF: $\mathit{MmT}\,[W, \texttt{TH}, K, \texttt{int}, \texttt{int}]$): $H \times W \times \texttt{int}$

TiledMmK(a: $H \times K \times \texttt{int}$, b: $W \times K \times \texttt{int}$, mmF: $\mathit{MmT}\,[W, H, \texttt{TK}, \texttt{int}, \texttt{int}]$): $H \times W \times \texttt{int}$

TiledConvHW(in: $H \times W \times \texttt{ICH} \times \texttt{int}$, wgt: $\texttt{OCH} \times \texttt{KS} \times \texttt{KS} \times \texttt{ICH} \times \texttt{int}$, convF: $\mathit{ConvT}\,[\texttt{TW}, \texttt{TH}, \texttt{ICH}, \texttt{OCH}, \texttt{KS}, \texttt{int}, \texttt{int}]$):
$(H - \texttt{KS} + 1) \times (W - \texttt{KS} + 1) \times \texttt{OCH} \times \texttt{int}$

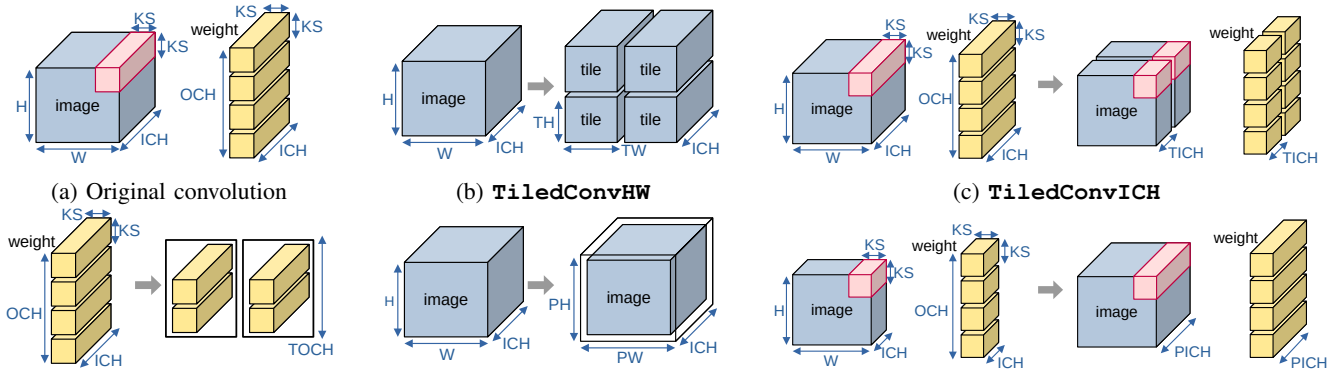
TiledConvICH(in: $H \times W \times \texttt{ICH} \times \texttt{int}$, wgt: $\texttt{OCH} \times \texttt{KS} \times \texttt{KS} \times \texttt{ICH} \times \texttt{int}$, convF: $\mathit{ConvT}\,[W, H, \texttt{TICH}, \texttt{OCH}, \texttt{KS}, \texttt{int}, \texttt{int}]$):
$(H - \texttt{KS} + 1) \times (W - \texttt{KS} + 1) \times \texttt{OCH} \times \texttt{int}$

TiledConvOCH(in: $H \times W \times \texttt{ICH} \times \texttt{int}$, wgt: $\texttt{OCH} \times \texttt{KS} \times \texttt{KS} \times \texttt{ICH} \times \texttt{int}$, convF: $\mathit{ConvT}\,[W, H, \texttt{ICH}, \texttt{TOCH}, \texttt{KS}, \texttt{int}, \texttt{int}]$):
$(H - \texttt{KS} + 1) \times (W - \texttt{KS} + 1) \times \texttt{OCH} \times \texttt{int}$

PaddedConvHW(in: $H \times W \times \texttt{ICH} \times \texttt{int}$, wgt: $\texttt{OCH} \times \texttt{KS} \times \texttt{KS} \times \texttt{ICH} \times \texttt{int}$, convF: $\mathit{ConvT}\,[\texttt{PW}, \texttt{PH}, \texttt{ICH}, \texttt{OCH}, \texttt{KS}, \texttt{int}, \texttt{int}]$):
$(H - \texttt{KS} + 1) \times (W - \texttt{KS} + 1) \times \texttt{OCH} \times \texttt{int}$

PaddedConvICH(in: $H \times W \times \texttt{ICH} \times \texttt{int}$, wgt: $\texttt{OCH} \times \texttt{KS} \times \texttt{KS} \times \texttt{ICH} \times \texttt{int}$, convF: $\mathit{ConvT}\,[W, H, \texttt{PICH}, \texttt{OCH}, \texttt{KS}, \texttt{int}, \texttt{int}]$):
$(H - \texttt{KS} + 1) \times (W - \texttt{KS} + 1) \times \texttt{OCH} \times \texttt{int}$

Fig. 7: Data-transformed skeleton definitions. Padding, tiling and transposition type parameters are highlighted in yellow.



(a) Original convolution (b) **TiledConvHW** (c) **TiledConvICH**

(d) **TiledConvOCH** (e) **PaddedConvHW** (f) **PaddedConvICH**

Fig. 8: Visualization of tiling and padding opportunities on convolutions. Tiling and padding can be applied to any of the four dimensions of a convolution. Data-transformed skeletons cover tiling on all four dimensions, splitting up larger convolutions into smaller tiles. The skeletons also describe padding on the height, width, and input channel dimensions, extending input sizes to share hardware with an existing larger convolution.

The function `shared`, of type $\mathit{MvT}[4, 3 \times 3 \times 16, \texttt{int}, 4 \times \texttt{int}]$, is reused across both cases. It computes partial dot products in $\texttt{c}$, and four grouped convolutions in $\texttt{c}'$, enabling hardware reuse across mismatched channel counts.

B. Transformation Rules

To derive transformed programs, SkeleShare applies equality saturation using a set of rewrite rules. These rules fall into three categories: dot product parallelism, data transformations, and partial dot product/parallel convolution rules.

1) Dot Product Parallelism Rules: Skeleton IR expressions initially assign each **ParallelDot** maximal parallelism for throughput. To reduce DSP usage, SkeleShare applies a rule that halves resource usage while doubling latency:

$$\textbf{ParallelDot}\,[2 \cdot \texttt{P}]\,(\texttt{a},\ \texttt{b}) \rightarrow \textbf{ParallelDot}\,[\texttt{P}]\,(\texttt{a},\ \texttt{b}) \quad \text{(R-HALVE-PARALLELISM)}$$

TABLE II: Additional constraints for pruning rewrites.

Rule	Avoids	Constraint
R-TILE-CONV-HW	Tiny tiles	$\texttt{TH}, \texttt{TW} \geq 6$
R-PAD-CONV-HW	Excessive padding	$\texttt{PH} - H, \texttt{PW} - W \leq 6$
R-PAD-PAR-CONV	Excessive padding	$\texttt{ICH} \cdot \texttt{KS}^2 < \texttt{CACHELN}$
R-PAD-MV	Excessive padding	$\texttt{PN} - P \leq \texttt{CACHELN}$

2) Data Transformation Rules: Each data-transformed skeleton gives rise to a corresponding rewrite with shape constraints that must be satisfied, captured in Figure 10.

Rules use canonical functions **cMvF**, **cMmF**, **cConvF** to preserve semantics under transformation. Only dimensions already present in the e-graph are considered for instantiation, reducing combinatorial explosion. Additional constraints in Table II further prune unhelpful transformations.

3) Partial Dot Product and Parallel Convolution Rules: When convolutional layers differ in input channel count, padding or tiling can harmonize them, but with per-

$$\textbf{MergePartialSum}(\texttt{elements: } M \times C \times \text{int}) : M \times \text{int}$$
$$\textbf{ParallelPartialDot}[P, C](\texttt{a: } M \times N \times \text{int}, \texttt{ b: } M \times N \times \text{int}) : M \times C \times \text{int}$$
$$\textbf{ParallelConv}(\texttt{in: } H \times W \times ICH \times T, \texttt{ wgt: } OCH \times KS \times KS \times ICH \times T, \texttt{ mvF: } \mathit{MvT}\,[OCH, C \cdot KS \cdot KS \cdot ICH, T, C \times T']) : (H - KS + 1) \times (W - KS + 1) \times OCH \times T'$$

Fig. 9: Partial dot product and parallel convolution skeletons.

$$\textbf{Conv}(\texttt{in: } H \times W \times ICH \times \text{int}, \texttt{ wgt: } OCH \times KS \times KS \times ICH \times \text{int}, \_) \to \textbf{TiledConvHW}(\texttt{in, wgt, } \textbf{cConvF}\,[TH, TW, ICH, OCH, KS]) \text{ where } H = 0 \bmod TH \text{ and } W = 0 \bmod TW \quad \text{(R-Tile-Conv-HW)}$$
$$\textbf{Conv}(\texttt{in: } H \times W \times ICH \times \text{int}, \texttt{ wgt: } OCH \times KS \times KS \times ICH \times \text{int}, \_) \to \textbf{TiledConvICH}(\texttt{in, wgt, } \textbf{cConvF}\,[H, W, TICH, OCH, KS]) \text{ where } ICH = 0 \bmod TICH \quad \text{(R-Tile-Conv-ICH)}$$
$$\textbf{Conv}(\texttt{in: } H \times W \times ICH \times \text{int}, \texttt{ wgt: } OCH \times KS \times KS \times ICH \times \text{int}, \_) \to \textbf{TiledConvOCH}(\texttt{in, wgt, } \textbf{cConvF}\,[H, W, ICH, TOCH, KS]) \text{ where } OCH = 0 \bmod TICH \quad \text{(R-Tile-Conv-OCH)}$$
$$\textbf{Conv}(\texttt{in: } H \times W \times ICH \times \text{int}, \texttt{ wgt: } OCH \times KS \times KS \times ICH \times \text{int}, \_) \to \textbf{PaddedConvHW}(\texttt{in, wgt, } \textbf{cConvF}\,[PH, PW, ICH, OCH, KS]) \text{ where } PH > H \text{ and } PW > W \quad \text{(R-Pad-Conv-HW)}$$
$$\textbf{Mm}(\texttt{a: } H \times K \times \text{int}, \texttt{ b: } W \times K \times \text{int}, \_) \to \textbf{TiledMmW}(\texttt{a, b, } \textbf{cMmF}\,[TW, H, K]) \text{ where } W = 0 \bmod TW \quad \text{(R-Tile-MmW)}$$
$$\textbf{Mm}(\texttt{a: } H \times K \times \text{int}, \texttt{ b: } W \times K \times \text{int}, \_) \to \textbf{TiledMmH}(\texttt{a, b, } \textbf{cMmF}\,[W, TH, K]) \text{ where } H = 0 \bmod TH \quad \text{(R-Tile-MmH)}$$
$$\textbf{Mm}(\texttt{a: } H \times K \times \text{int}, \texttt{ b: } W \times K \times \text{int}, \_) \to \textbf{TiledMmK}(\texttt{a, b, } \textbf{cMmF}\,[W, H, TK]) \text{ where } K = 0 \bmod TK \quad \text{(R-Tile-MmK)}$$
$$\textbf{Mv}(\texttt{mat: } M \times N \times \text{int}, \texttt{ vec: } N \times \text{int}, \_) \to \textbf{TiledMv}(\texttt{mat, vec, } \textbf{cMvF}\,[M, TN]) \text{ where } N = 0 \bmod TN \quad \text{(R-Tile-Mv)}$$
$$\textbf{Mv}(\texttt{mat: } M \times N \times \text{int}, \texttt{ vec: } N \times \text{int}, \_) \to \textbf{PaddedMv}(\texttt{mat, vec, } \textbf{cMvF}\,[M, PN]) \text{ where } PN > N \quad \text{(R-Pad-Mv)}$$

where $\textbf{cMvF}\,[M, N] = \lambda\texttt{mat: } M \times N \times \text{int}.\ \lambda\texttt{vec: } N \times \text{int}.\ \textbf{Mv}(\texttt{mat, vec, } \lambda\texttt{a. } \lambda\texttt{b. } \textbf{ParallelDot}\,[M](\texttt{a, b}))$
$\textbf{cMmF}\,[W, H, K] = \lambda\texttt{a: } H \times K \times \text{int}.\ \lambda\texttt{b: } W \times K \times \text{int}.\ \textbf{Mm}(\texttt{a, b, } \textbf{cMvT}[W, K])$
$\textbf{cConvF}\,[H, W, ICH, OCH, KS] = \lambda\texttt{in: } H \times W \times ICH \times \text{int}.\ \lambda\texttt{wgt: } OCH \times KS \times KS \times ICH \times \text{int}.$
$\quad \textbf{Conv}(\texttt{in, wgt, } \textbf{cMvF}\,[OCH, KS \cdot KS \cdot ICH, \text{int}, \text{int}])$

Fig. 10: Data transformation rules.

$$\textbf{Conv}(\texttt{in: } H \times W \times ICH \times \text{int}, \texttt{ wgt: } OCH \times KS \times KS \times ICH \times \text{int}, \_) \to \textbf{PaddedConvICH}(\texttt{in, wgt, } \lambda\texttt{pi: } H \times W \times PICH \times \text{int}.\ \lambda\texttt{pw: } OCH \times KS \times KS \times PICH \times \text{int}.\ \textbf{ParallelConv}(\texttt{pi, pw, } \textbf{pMvF}\,[C, OCH, C \times PICH])), \quad \text{(R-Pad-Par-Conv)}$$

where $\textbf{pMvF}\,[C, M, N] = \lambda\texttt{mat: } M \times N \times \text{int}.\ \lambda\texttt{vec: } N \times \text{int}.\ \textbf{Mv}(\texttt{mat, vec, } \lambda\texttt{a. } \lambda\texttt{b. } \textbf{ParallelPartialDot}\,[C, M](\texttt{a, b}))$

Fig. 11: Partial parallel convolution padding rule.

formance trade-offs. Alternatively, SkeleShare can use the rule in Figure 11 to replace **ParallelDot** with **ParallelPartialDot**, allowing reuse across convolution layers without sacrificing performance.

C. Extractor Cost Model

The extractor cost model minimizes algorithmic time while ensuring the design fits within target FPGA resource limits. Building on Section III-C, where selection and sharing decisions are encoded as Boolean variables, the model evaluates these decisions along two dimensions:

1) **Algorithmic Time $\mathcal{T}$:** Total computation steps required.
2) **DSP Usage $\mathcal{D}$:** Measures the number of DSPs consumed, inferred from the size and number of multiplications and dot products.

Cost estimation is tractable due to language restrictions: SkeleShare targets a functional DSL without recursion or recursive types, and with higher-order functions limited to skeleton composition. All computation is over statically sized arrays, enabling static inference of throughput and DSP usage.

DSP usage is treated as a hard constraint; algorithmic time is minimized. The design restricts hardware convolution units to one, improving routing and reflecting FPGA best practices around reuse. The optimization problem, for root e-class r, is:

$$\begin{aligned}
&\underset{vars}{\text{minimize}}\ use_cost_{\mathcal{T}}(r) \\
&\text{subject to } use_cost_{\mathcal{D}}(r) + \sum_{c \in classes} fixed_cost_{\mathcal{D}}(c) \le \texttt{MAX_DSPS} \\
&\qquad \text{and } conv_count \le 1, \\
&\text{where } vars = \{select_n \mid n \in nodes\} \cup \{select_c \mid c \in classes\} \\
&\qquad\qquad \cup \{share_c \mid c \in classes\}.
\end{aligned}$$

The optimization problem above involves the domain-agnostic e-class cost functions *use_cost* for the root and *fixed_cost* for all classes selected, as each shared, selected class gives rise to a shared hardware component whose hardware resource usage is not accounted for by the *use_cost* of the root class. As seen in Section III-C, *use_cost* and

fixed_cost are defined in terms of the domain-specific e-node cost functions *inplace_cost* and *shared_cost*. The definition of *shared_cost* for the two dimensions is:

$$\begin{aligned} &\mathit{shared\_cost}_{\mathcal{T}}(n) = \mathit{inplace\_cost}_{\mathcal{T}}(n) + \texttt{COMMUNICATION\_COST}, \\ &\mathit{shared\_cost}_{\mathcal{D}}(n) = 0. \end{aligned}$$

where `COMMUNICATION_COST` is a small integer constant (5 in our implementation) that discourages unnecessary sharing without blocking beneficial reuse.

The shared cost definition encodes two intuitions: 1) interfacing with a shared component incurs communication overhead in addition to the time required by that component to perform the desired computation, 2) using a shared component does not allocate DSPs at the use site.

The in-place DSP cost model reflects back-end limits, such as a 64-DSP cap per dot product:

$$\begin{aligned} &\mathit{inplace\_cost}_{\mathcal{D}}(\textbf{ParallelDot}\,[\mathtt{P}]\ (\mathtt{a}\colon\ \mathtt{M}\times\mathtt{N}\times\mathtt{int},\ \mathtt{b})) \\ &\quad = \mathtt{P}\cdot\min(\mathtt{N},64), \\ &\mathit{inplace\_cost}_{\mathcal{D}}(\textbf{ParallelPartialDot}\,[\mathtt{P},\mathtt{C}]\ (\mathtt{a}\colon\ \mathtt{M}\times\mathtt{N}\times\mathtt{int},\ \mathtt{b})) \\ &\quad = \mathtt{P}\cdot\min(\mathtt{N},64), \\ &\mathit{inplace\_cost}_{\mathcal{D}}(\mathbf{f}\,(\mathtt{a}_1,\mathtt{a}_2,\ldots,\mathtt{a}_n)) = \sum_n \mathtt{a}_i \text{ for any other primitive } \mathbf{f}. \end{aligned}$$

The in-place algorithmic time cost function $\mathit{inplace\_cost}_{\mathcal{T}}$ formalizes intuitions such as: the cost of a padding skeleton is equivalent to cost of the padded computation while the cost of a tiling skeleton is defined as the cost of processing a tile multiplied by the number of tiles. Moreover, $\mathit{inplace\_cost}_{\mathcal{T}}$ accounts for the back-end's DSP allocation strategy. This can be observed in its definition for **`ParallelDot`**:

$$\begin{aligned} &\mathit{inplace\_cost}_{\mathcal{T}}(\textbf{ParallelDot}\,[\mathtt{P}]\ (\mathtt{a}\colon\ \mathtt{M}\times\mathtt{N}\times\mathtt{int},\ \mathtt{b})) \\ &\quad = \frac{\mathtt{M}\cdot\mathtt{N}}{\mathtt{P}\cdot\min(\mathtt{N},64)} + \max(\mathcal{T}(a),\mathcal{T}(b)) \end{aligned}$$

Section A defines the full algorithmic time function.

D. Lowering

This final stage of SkeleShare lowers the optimized skeleton IR into hardware-level constructs. Each skeleton is expanded using a template mapped to SHIR's mid-level primitives [23]. Template selection is guided by the input dimensions, as different shapes induce different memory access patterns; each template is optimized accordingly.

When skeletons are composed—*e.g.,* **`PaddedConvICH`** followed by **`TiledConvHW`**—lowering them in isolation introduces intermediate buffers, degrading performance. To avoid this, the lowering process is to treat composed skeletons as a unit, reordering and fusing them to eliminate temporary buffers. A preferred transformation order also unlocks further back-end rewrites that improve code quality [24].

Once lowered to mid-level primitives, the result is passed to the SHIR compiler, which applies further optimizations and maps each primitive to a VHDL template. The composed templates are then passed to Intel Quartus, an FPGA toolchain.

V. Use Case: Image Processing on FPGAs

We now illustrate how the approach from the previous section extends to image processing pipelines. Such pipelines are dominated by 2D convolutions, separable into 1D convolutions. This section describes new skeletons and rewrite rules to support this use case. Extraction and lowering are as before.

A. New Skeletons

Two new core operations support image processing: **`Conv1D_W`**`(img, wgt, mvF)` and **`Conv1D_H`**`(img, wgt, mvF)`, which implement 1D convolution across the width and height of the input image. `mvF` expresses 1D convolution in terms of matrix-vector multiplication using the classical im2col implementation technique. A new data-transformed skeleton expresses height-wise 1D convolution as a width-wise convolution: **`TransposedConv1D_W`**`(img, wgt, conv1D_WF)`. While not needed for the application that will be evaluated later, other data-transformed computations (*e.g.,* padding or tiling) can easily be added.

B. Transformation Rules

In the context of 1D convolutions, the only additional transformation rule required is to express that convolving along the height is equivalent to convolving a transposed image along the width. The following rule expresses this:

$$\begin{aligned} &\textbf{Conv1D\_H}(\mathtt{img}\colon\ \mathtt{H}\times\mathtt{W}\times\mathtt{int},\ \mathtt{wgt}\colon\ \mathtt{K}\times\mathtt{int},\ \_) \rightarrow \\ &\quad \textbf{TransposedConv1D\_W}(\mathtt{img},\ \mathtt{wgt},\ \textbf{cConv1D\_WF}[\mathtt{H},\mathtt{W},\mathtt{K}]) \end{aligned}$$

where $\textbf{cConv1D\_WF}[\mathtt{W},\mathtt{H},\mathtt{K}] = \lambda\mathtt{img}\colon\ \mathtt{H}\times\mathtt{W}\times\mathtt{int}.\ \lambda\mathtt{wgt}\colon\ \mathtt{K}\times\mathtt{int}.\ \textbf{ConvW}(\mathtt{a},\ \mathtt{b},\ \textbf{cMvF}[\mathtt{W}-\mathtt{K}+1,\mathtt{K}])$.

C. Summary

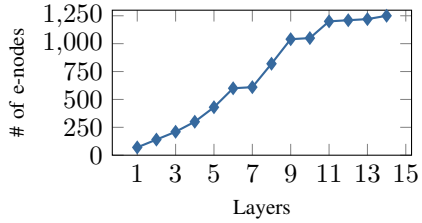
To summarize, adding new skeletons for a different domain like image convolution involves: 1) defining the main operators for the domain, and 2) adding skeletons and rewrite rules to express operators as a function of each other to enable sharing.

VI. Evaluation

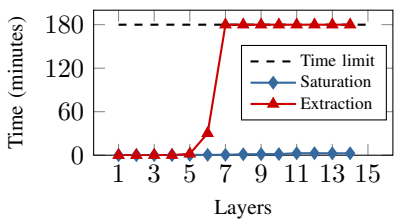
SkeleShare is evaluated on the VGG-CIFAR and TinyYolo v2 convolutional networks as well as on a self-attention layer and a four-stage stencil pipeline. These benchmarks assess its impact on resource usage, scalability, and performance. Experiments include: 1) a comparison with prior hand-optimized work on real hardware, 2) a scalability study on progressively larger network slices, 3) a comparison of predicted and measured performance, 4) an ablation study to verify the necessity of each core transformation for correctness, and 5) an analysis to test robustness under increasingly limited DSP budgets.

A. Experimental Setup

We implement SkeleShare in Scala using SHIR [23] as the back-end. Equality saturation uses De Bruijn indices [16, 29], with extraction via Z3 [7] and a timeout-driven refinement loop. Experiments run on an Intel Arria 10 FPGA, a resource-constrained platform representative of embedded deployments.



(a) Number of e-nodes for different numbers of layers.



(b) Compilation time for different numbers of layers.

Fig. 12: Scaling behavior of SkeleShare on increasingly large slices of VGG-CIFAR. x-axis indicates the number of layers (*e.g.,* $x = 5$ uses first five layers).

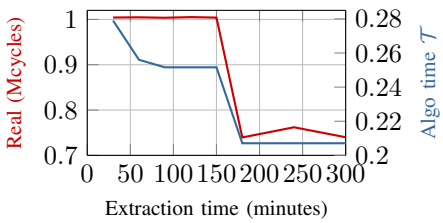


Fig. 13: Cost model predictions and measured hardware runtime for VGG under different solver timeouts.

Input and weights are preloaded on the host; runtime includes PCIe overhead. Benchmarks are VGG-CIFAR and TinyYolo v2, models widely used in low-power applications such as edge inference and mobile vision. Running them on a constrained FPGA mirrors deployment conditions and tests SkeleShare's ability to optimize under tight budgets. VGG comprises 13 convolutional, 5 pooling, and one fully connected layer, with spatial sizes from 32×32 to 1×1. TinyYolo has 9 convolutional and 5 pooling layers, with inputs shrinking from 416×416 to 13×13 and channels scaling to 1024. Together, they exercise sharing across diverse and regular structures.

A self-attention layer benchmark confirms generalizability to non-convolutional networks. A stencil benchmark consisting of 3×3 Gaussian blur followed by 3×3 edge detection on a 64×64 image shows that SkeleShare generalizes to use cases other than neural networks. The filters in the stencil benchmark are decomposed into 1D convolutions over a 2D image, one convolution working horizontally; the other vertically.

B. Comparison with Prior Work on an FPGA

Experiments 1–7 in Table III compare SkeleShare to hand-optimized baselines from prior work [15, 18, 33], which involved expert-tuned manual padding and tiling of layers to create shared components. Prior work achieves 166 GOPS on VGG, 500 to 608 GOPS on TinyYolo, and 583 GOPS on self-attention. SkeleShare achieves 163 GOPS on VGG, 647 GOPS on TinyYolo and 648 GOPS on self-attention, matching the baseline on VGG and surpassing it on the others. These results show that automated sharing and allocation can deliver expert-level designs with drastically less human effort, at times even surpassing human baselines. The slightly higher resource usage on VGG stems from differences in RTL implementation rather than approach: synthesis is sensitive to low-level details.

C. Stencil Pipeline Performance

Experiments 8–9 compare SkeleShare's 4-stage stencil benchmark solution to an optimized custom-engineered baseline without sharing, meaning the baseline consists of four pipeline stages of which only one is active at a time. While this is the ideal solution in terms of pure throughput, it uses its DSPs inefficiently. When prompted to share filter hardware by lowering the DSP budget to a quarter of the baseline design's resources, SkeleShare finds a solution that meets this constraint and delivers a solution that, at 61 GOPS, is only slightly slower than the baseline's at 63 GOPS, where the slowdown is due to the overhead of shared resources.

TABLE III: Performance evaluation of SkeleShare on VGG and TinyYolo. Logic, RAM, and DSPs show resource use. Throughput metric: GOPS (Giga-Operations Per Second). Prior work does not report logic/RAM use for TinyYolo [15].

Experiment	Logic	RAM	DSPs	GOPS
1. VGG, SkeleShare	49%	35%	76%	163
2. VGG, [15]	48%	33%	76%	166
3. TinyYolo, SkeleShare	38%	24%	76%	647
4. TinyYolo, [15]	N/A	N/A	76%	608
5. TinyYolo, [33]	N/A	N/A	58%	500
6. Self-attention, SkeleShare	35%	29%	67%	648
7. Self-attention, [18][1]	10%	7%	14%	583
8. 4-stage stencil, SkeleShare	10%	7%	6%	61
9. 4-stage stencil, baseline	11%	7%	25%	63
10. VGG, SkeleShare, no sharing	No solution found			
11. VGG, SkeleShare, no padding	No solution found			
12. VGG, SkeleShare, no tiling	No solution found			
13. VGG, baseline, no sharing	Not synthesizable			
14. VGG, SkeleShare, 1 abstr.	45%	35%	72%	125
15. VGG, SkeleShare, 1/4 DSPs	42%	35%	19%	65
16. VGG, [15], 1/4 DSPs	40%	33%	19%	69
17. VGG, SkeleShare, 1/2 DSPs	40%	35%	38%	123
18. VGG, [15], 1/2 DSPs	42%	33%	38%	136

[1] Evaluated on Xilinx Virtex Ultrascale+ with more hardware resources than Intel Arria 10. Both self-attention designs use 1024 DSPs.

D. Scalability

To understand how SkeleShare maintains this level of performance as models grow, we examine its scalability using increasingly deep slices of VGG-CIFAR. Figure 12 shows that the number of e-nodes grows near-linearly with layer count, plateauing when layer shapes repeat (*e.g.,* layers 6 and 7). Although equality saturation remains efficient, extraction time increases steeply due to NP-hardness of the optimal extraction problem that the solver faces.

To ensure tractability, SkeleShare relaxes the extraction problem by querying the solver for progressively better solutions, returning the best solution found before a timeout. As shown in Figure 13, hardware runtime improves with longer solver timeouts and saturates around the 180-minute mark,

when all DSPs are fully shared and more solver time loses its practical benefit on VGG. Cost model predictions track this trend closely, with minor deviations due to PCIe traffic (Section VI-A). Since VGG is the largest model in the data set, we impose a 180 minute timeout on extraction, applying to all results in Table III and Figure 12b. This time budget is shorter than the synthesis time; as shown by the aforementioned table, SkeleShare's delivers strong results even with the timeout.

E. Verifying the Necessity of Transformations

The ablation experiments in Table III validate that each transformation is essential for mapping large models like VGG to an FPGA. Disabling sharing, padding, or tiling (Experiments 10–12) consistently results in no feasible solution. This reflects an established fact prior work that these transformations are required to fit deep networks onto FPGAs [15], further confirmed by Experiment 13, which shows the non-synthesizability of a design that directly implements VGG.

Intuitively, padding and tiling expose sharing opportunities across repeated layers. Without them, resource demands exceed FPGA capacity, preventing any valid design from emerging. Similarly, disabling sharing removes the ability to reuse common subcomputations across layers.

F. Quantifying the Benefit of Multi-Abstraction Rewriting

To quantify the performance gains of multi-abstraction rewriting, which reuses hardware across different abstractions, Experiment 14 disables such sharing for the VGG benchmark while still permitting the full spectrum of transformations. Intuitively, this is equivalent to running SkeleShare on a DSL rather than a skeleton IR, while still considering tiling, padding, and same-abstraction sharing.

The design produced by SkeleShare for this experiment is one where the convolutional layers are transformed such that their hardware is shared. Due to the constraints of the experiment, the fully connected layer is implemented using separate resources. This limited reuse leads to performance of 125 GOPS compared to 163 GOPS with cross-abstraction sharing; hence, multi-abstraction rewriting produces a speedup of 1.3× for VGG.

G. Scalability Across Hardware Constraints

To assess how well SkeleShare performs under varying resource constraints, we evaluate its throughput on devices with progressively smaller DSP budgets. Experiments 15–18 evaluate the technique under reduced DSP budgets, using one-quarter and one-half of the available DSPs, respectively. SkeleShare adapts gracefully to both, with throughput nearly doubling as the budget increases (65 vs. 123 GOPS). Additional gains are limited by board bandwidth, as high DSP usage exceeds the memory system's capacity to supply data.

VII. Related Work

SkeleShare builds on prior work in functional hardware design, optimization, and equality saturation, reviewed below.

Functional HLS and Resource Sharing: Mapping high-level functional programs to FPGAs traditionally involves manual tuning of allocation and sharing [14, 15], with High-Level Synthesis (HLS) frameworks such as Intel OpenCL, LegUp [4], and Xilinx HLS offering vendor-specific sharing directives. Functional languages offer abstractions that naturally express parallelism and reuse. Chisel [3], CλaSH [2], and Spatial [17] promote modular design. SHIR [23, 24] separates algorithmic structure from hardware control, and serves as this paper's compiler back-end.

Design Space Exploration in HLS: Design space exploration automates hardware optimization. ScaleHLS [35] applies dynamic programming and evolutionary algorithms to explore transformation sequences. AutoPhase [12] learns LLVM pass sequences for circuit generation. COSMOS [21] co-optimizes datapath and memory components, and Sensei [11] predicts post-synthesis area. These approaches differ from SkeleShare, which performs structural program rewrites.

Program Transformations: Several prior systems focus on performance optimization through rewrites and algorithmic tuning. PetaBricks [1] enables algorithm selection at runtime for adaptive tuning. Spiral [22] generates code for linear transforms using rewrites and performance models. Lift [27] compiles functional parallel patterns to optimized GPU code using equational reasoning. These systems optimize for throughput and portability; SkeleShare targets hardware reuse through solver-backed sharing and allocation.

Equality Saturation: Equality saturation [28, 32] uses e-graphs [8, 19, 20] to represent equivalent programs. SEER [5] and Churchroad [26] apply e-graph rewriting to optimize control flow and datapaths. Glenside [25] rewrites data layouts for accelerators. Tensat [34] optimizes tensor expressions. These projects operate on a flat abstraction; SkeleShare leverages multi-abstraction e-graph rewriting for sharing and allocation.

Algorithmic Skeletons: Algorithmic skeletons abstract reusable parallel patterns to express *what* is computed, for systematic program structuring [6, 36]. PetaBricks [1] applies algorithmic skeletons to determine *how* to efficiently compute results, similar to how SkeleShare transforms computations with padding and tiling. Like SkeleShare, PetaBricks can work across levels of abstraction. SkeleShare builds on these ideas and combines them with e-graphs to implement multi-abstraction e-graph rewriting for the challenge of resource allocation and sharing, which PetaBricks does not consider.

VIII. Conclusion

Sharing and allocation in functional FPGA compilation are challenging due to the combinatorial space of implementation variants. SkeleShare tackles this by representing programs with algorithmic skeletons and using equality saturation to expose all sharing opportunities. A solver-based extractor, guided by a cost model, selects a resource-conforming variant. SkeleShare matches expert-tuned baselines on VGG and outperforms them on TinyYolo, showing that principled automation can rival and surpass manual designs.

Data-Availability Statement

Additional data related to this publication may be found in the artifact's Zenodo repository [30].

Acknowledgments

We thank Christof Schlaak for implementing the core SHIR project, used as the back-end compiler for this work. This work was supported by the Fonds de Recherche du Québec–Nature et Technologies' 3rd cycle scholarship, awards #304858, #346530 and #353820. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

Artifact Appendix

A. Abstract

This artifact contains a Dockerfile that assembles the SkeleShare implementation and its dependencies into a container. The container is designed to reproduce Table III, which contains the main experimental findings of this paper.

B. Artifact check-list (meta-information)

- **Algorithm:** An equality-saturation-based, multi-abstraction e-graph rewriting algorithm with a solver-based cost-model-guided extractor for FPGA resource sharing.
- **Compilation:** Docker version 26.1.4 builds and runs the artifact.
- **Data set:** VGG-CIFAR, TinyYolo v2, Self-Attention, and Stencil benchmarks.
- **Hardware:** For *container execution* only, any x86-64 machine capable of running Docker. For *reproducing FPGA resource/throughput* numbers, an Intel Arria 10 FPGA connected to a compatible x86-64 host via PCI-Express and a compatible toolchain. The reference setup used for the paper is a dual-socket Intel Xeon Gold 6254 server with an Arria 10 GX1150 FPGA.
- **Metrics:** FPGA resource usage (logic, RAM, DSPs), and throughput (GOPS).
- **Output:** VHDL files of the final design.
- **Experiments:** Scaling behavior of SkeleShare on increasing VGG layers, EqSat exploration for benchmarks, VHDL generation for chosen extracted programs, evaluation of neural network models (VGG, TinyYolo), self-attention, and stencil pipelines, ablation studies, and experiments under constrained DSP budgets.
- **How much time is needed to complete experiments?:** Approximately 3 hours with the *quick* version (precomputed solutions and pre-synthesized FPGA design), and several days with the *complete* version (recomputing solutions and synthesizing designs).
- **Publicly available?:** Yes
- **Code licenses?:** MIT
- **Archived?:** https://doi.org/10.5281/zenodo.17635866

C. Description

1) How delivered: This artifact is available as a pre-built Docker image, containing all mentioned experiments. Download as below. [1]

```
$ docker pull \
    ghcr.io/jonathanvdc/skeleshare-cgo26-artifact:latest
```

[1]Depending on the Docker installation, `docker` may require `sudo`.

Alternatively, the container can be compiled from source code, available on Zenodo [30], by running the following command:

```
$ docker build -t skeleshare-cgo26-artifact .
```

2) Hardware dependencies: An x86-64 CPU, an Intel Programmable Acceleration Card (with Intel Arria 10 GX1150 FPGA) connected to the host CPU via PCI-Express.

3) Software dependencies: RHEL 7.6, Intel Quartus Prime Pro V19.2, Intel FPGA OPAE SDK V1.3.0, Intel Acceleration Stack for Intel Arria 10 V1.2.1

D. Experiment workflow

Table III presents the core findings of this paper. To replicate these results, we recommend the following workflow. Additional hardware-specific details, including the synthesis instructions, are provided in the artifact README. [2] This workflow involves generating VHDL code for each benchmark and running it on the FPGA.

1) Pull the Docker image: As explained in Section VIII-C1.

2) Run the container: First, create a local directory for intermediate results. Then, invoke the container's evaluation flow, which generates VHDL code for all experiments. The generated VHDL files for each experiment will be stored in `./results/<experiment-id>/lowering/`:

```
$ mkdir -p results
$ docker run --rm -it --mount \
    type=bind,src=./results,dst=/workspace/results \
    ghcr.io/jonathanvdc/skeleshare-cgo26-artifact:latest
```

3) Synthesize the design: To obtain logic, RAM, DSP and GOPS measurements, synthesize the VHDL files using Quartus and execute them on the FPGA with the Intel OPAE SDK. Because synthesis and execution are hardware/software dependent, the README describes the exact procedure for the reference evaluation platform.

4) Quick versus Complete experiment: Optimizing benchmarks with equality saturation and synthesizing FPGA designs can take many hours, so the artifact includes two modes. The quick mode uses precomputed solutions and pre-synthesized FPGA designs, enabling faster replication. The complete mode recomputes the EqSat solutions, and comes with synthesis instruction to fully reproduce the end-to-end flow.

The commands above use a precomputed solution embedded in the container. To recompute the solution and place it in `./results/<experiment-id>/eqsat/`, run:

```
$ docker run --rm -it --mount \
    type=bind,src=./results,dst=/workspace/results \
    ghcr.io/jonathanvdc/skeleshare-cgo26-artifact:latest \
    python3 evaluation.py --phase eqsat
```

To generate VHDL code and save it under `./results/<experiment-id>/lowering/`, run:

```
$ docker run --rm -it --mount \
    type=bind,src=./results,dst=/workspace/results \
    ghcr.io/jonathanvdc/skeleshare-cgo26-artifact:latest \
    python3 evaluation.py --phase lowering
```

Pre-synthesized designs are available on servers with access for evaluators; detailed instructions for using those designs and for running synthesis on the reference setup are provided in the README.

Finally, Figure 12 presents the scaling behavior of SkeleShare on increasing number of VGG layers. The artifact README provides details to extract e-node counts and saturation curve to reproduce Figure 12a and Figure 12b.

E. Evaluation and expected result

Running the experiment workflow as provided will lead to replicating the results presented in Table III. The table's rows can be cross-referenced with the resource usage and throughput measurements collected after executing each benchmark on the FPGA board.

[2]The artifact README can be found at https://github.com/jonathanvdc/skeleshare-cgo26-artifact/blob/main/README.md

References

[1] Jason Ansel et al. "PetaBricks: a language and compiler for algorithmic choice". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 38–49. ISBN: 9781605583921. DOI: 10.1145/1542476.1542481. URL: https://doi.org/10.1145/1542476.1542481.

[2] Christiaan Baaij and Jan Kuper. "Using rewriting to synthesize functional languages to digital circuits". In: *International Symposium on Trends in Functional Programming*. Springer. 2013, pp. 17–33.

[3] Jonathan Bachrach et al. "Chisel: constructing hardware in a Scala embedded language". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. URL: https://doi.org/10.1145/2228360.2228584.

[4] Andrew Canis et al. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. Monterey, CA, USA: Association for Computing Machinery, 2011, pp. 33–36. ISBN: 9781450305549. DOI: 10.1145/1950413.1950423. URL: https://doi.org/10.1145/1950413.1950423.

[5] Jianyi Cheng et al. "SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 1029–1044. ISBN: 9798400703850. DOI: 10.1145/3620665.3640392. URL: https://doi.org/10.1145/3620665.3640392.

[6] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[7] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[8] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. "Variations on the Common Subexpression Problem". In: *J. ACM* 27.4 (Oct. 1980), pp. 758–771. ISSN: 0004-5411. DOI: 10.1145/322217.322228. URL: https://doi.org/10.1145/322217.322228.

[9] David Durst et al. "Type-directed scheduling of streaming accelerators". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 408–422.

[10] Mike He et al. "Improving term extraction with acyclic constraints". In: *EGRAPHS 2023 workshop*. 2023.

[11] Hsuan Hsiao and Jason H. Anderson. "Sensei: An area-reduction advisor for FPGA high-level synthesis". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Mar. 2018, pp. 25–30. DOI: 10.23919/DATE.2018.8341974.

[12] Qijing Huang et al. "AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 308–308. DOI: 10.1109/FCCM.2019.00049.

[13] Mikolas Janota, Radu Grigore, and Vasco Manquinho. "On the quest for an acyclic graph". In: *arXiv preprint arXiv:1708.01745* (2017).

[14] Tzung-Han Juang and Christophe Dubach. "Maximizing Data and Hardware Reuse for HLS with Early-Stage Symbolic Partitioning". In: *ACM Trans. Archit. Code Optim.* (Jan. 2025). ISSN: 1544-3566. DOI: 10.1145/3711926. URL: https://doi.org/10.1145/3711926.

[15] Tzung-Han Juang, Christof Schlaak, and Christophe Dubach. "Let Coarse-Grained Resources Be Shared: Mapping Entire Neural Networks on FPGAs". In: *ACM Trans. Embed. Comput. Syst.* 22.5s (Sept. 2023). ISSN: 1539-9087. DOI: 10.1145/3609109. URL: https://doi.org/10.1145/3609109.

[16] Thomas Koehler et al. "Guided Equality Saturation". In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024). DOI: 10.1145/3632900. URL: https://doi.org/10.1145/3632900.

[17] David Koeplinger et al. "Spatial: a language and compiler for application accelerators". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 296–311. ISBN: 9781450356985. DOI: 10.1145/3192366.3192379. URL: https://doi.org/10.1145/3192366.3192379.

[18] Tianyang Li et al. "Unified accelerator for attention and convolution in inference based on FPGA". In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2023, pp. 1–5.

[19] Charles Gregory Nelson. "Techniques for program verification". AAI8011683. PhD thesis. Stanford, CA, USA, 1980.

[20] Greg Nelson and Derek C. Oppen. "Fast Decision Procedures Based on Congruence Closure". In: *J. ACM* 27.2 (Apr. 1980), pp. 356–364. ISSN: 0004-5411. DOI: 10.1145/322186.322198. URL: https://doi.org/10.1145/322186.322198.

[21] Luca Piccolboni et al. "COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators". In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017). ISSN: 1539-9087. DOI: 10.1145/3126566. URL: https://doi.org/10.1145/3126566.

[22] Markus Puschel et al. "SPIRAL: Code generation for DSP transforms". In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275.

[23] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. "Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators". In: *ACM Trans. Archit. Code Optim.* 19.2 (Jan. 2022). ISSN: 1544-3566. DOI: 10.1145/3501768. URL: https://doi.org/10.1145/3501768.

[24] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. "Optimizing data reshaping operations in functional IRs for high-level synthesis". In: *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 61–72. ISBN: 9781450392662. DOI: 10.1145/3519941.3535069. URL: https://doi.org/10.1145/3519941.3535069.

[25] Gus Henry Smith et al. "Pure tensor program rewriting via access patterns (representation pearl)". In: *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 21–31. ISBN: 9781450384674. DOI: 10.1145/3460945.3464953. URL: https://doi.org/10.1145/3460945.3464953.

[26] Gus Henry Smith et al. "Scaling Program Synthesis Based Technology Mapping with Equality Saturation". In: *arXiv preprint arXiv:2411.11036* (2024).

[27] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 74–85.

[28] Ross Tate et al. "Equality Saturation: A New Approach to Optimization". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 264–276. ISBN: 9781605583792. DOI: 10.1145/1480881.1480915. URL: https://doi.org/10.1145/1480881.1480915.

[29] Jonathan Van der Cruysse and Christophe Dubach. "Latent Idiom Recognition for a Minimalist Functional Array Language using Equality Saturation". In: *Proceedings of the 22nd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '24. Edinburgh, United Kingdom, 2024, pp. 270–282. URL: https://jonathanvdc.github.io/files/2024-cgo-latent-idiom-recognition.pdf.

[30] Jonathan Van der Cruysse, Tzung-Han Juang, and Dubach Christophe Bolbolian Khah Shakiba. *SkeleShare: Algorithmic Skeletons and Equality Saturation for Hardware Resource Sharing*. Zenodo, Nov. 2025. DOI: https://doi.org/10.5281/zenodo.17925912.

[31] Yisu Remy Wang et al. "SPORES: sum-product optimization via relational equality saturation for large scale linear algebra". In: *Proceedings of the VLDB Endowment* 13.12 (2020).

[32] Max Willsey et al. "`egg`: Fast and Extensible Equality Saturation". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

[33] Ke Xu et al. "A dedicated hardware accelerator for real-time acceleration of YOLOv2". In: *Journal of Real-Time Image Processing* 18 (2021), pp. 481–492.

[34] Yichen Yang et al. "Equality saturation for tensor graph superoptimization". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 255–268.

[35] Hanchen Ye et al. "ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. San Francisco, California: Association for Computing Ma-

chinery, 2022, pp. 1355–1358. ISBN: 9781450391429. DOI: 10.1145/3489517.3530631. URL: https://doi.org/10.1145/3489517.3530631.

[36] Mani Zandifar et al. "Composing Algorithmic Skeletons to Express High-Performance Scientific Applications". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: Association for Computing Machinery, 2015, pp. 415–424. ISBN: 9781450335591. DOI: 10.1145/2751205.2751241. URL: https://doi.org/10.1145/2751205.2751241.