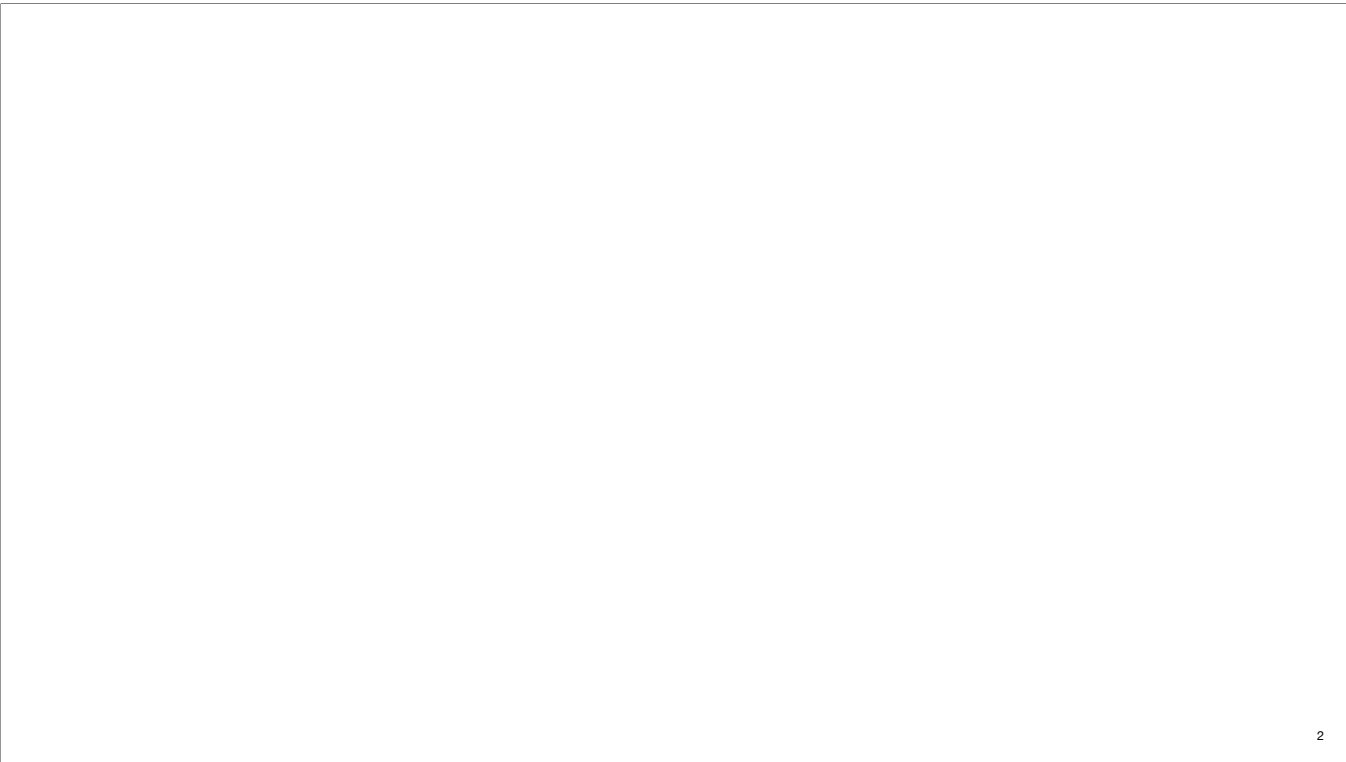


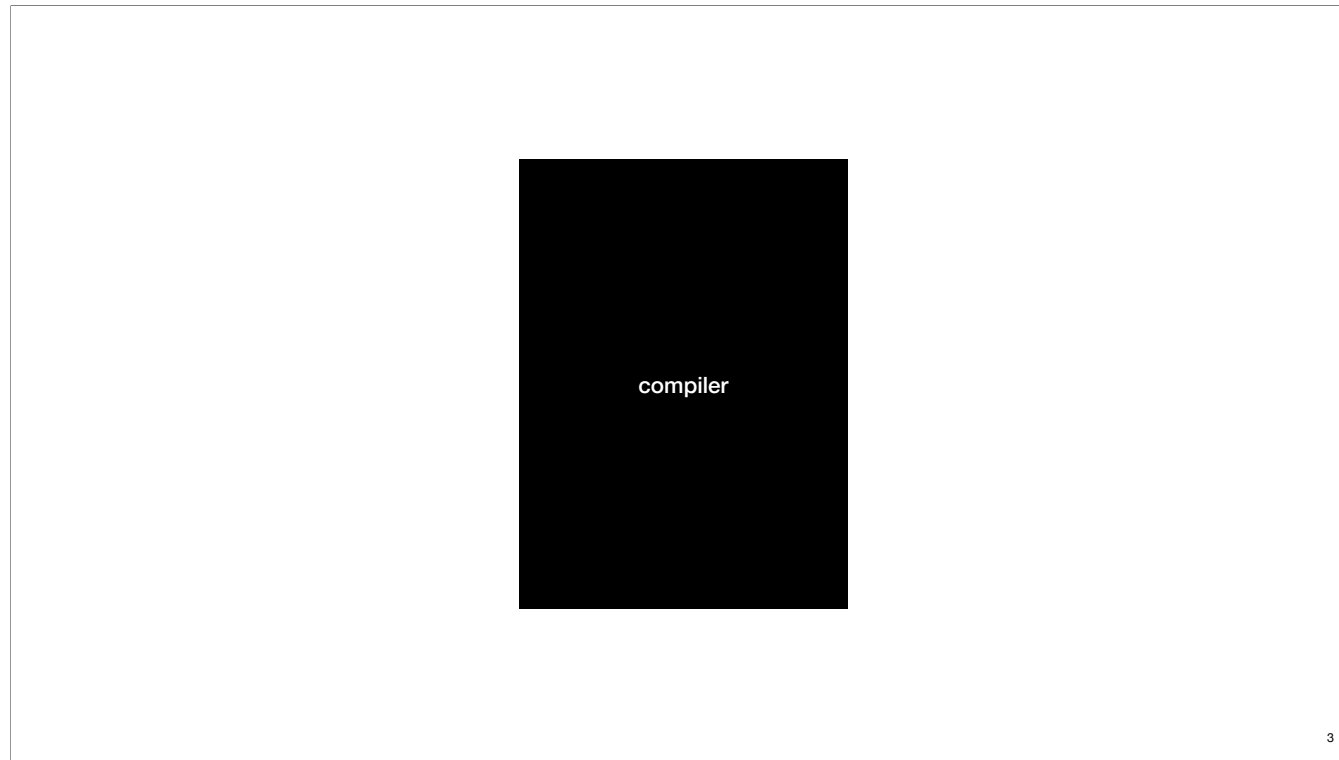
Generating Compiler Backends from Formal Models of Hardware

Gus Smith's Generals Exam

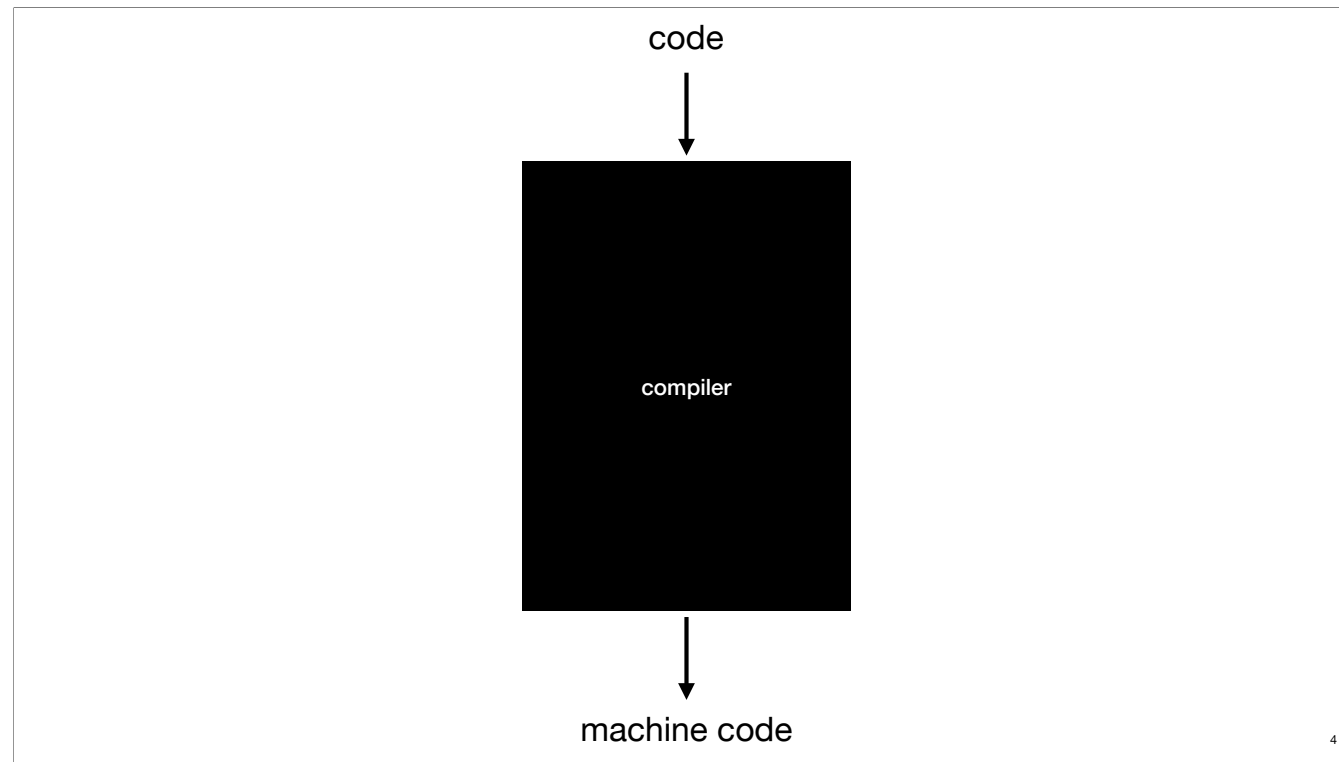
May 6th, 2022



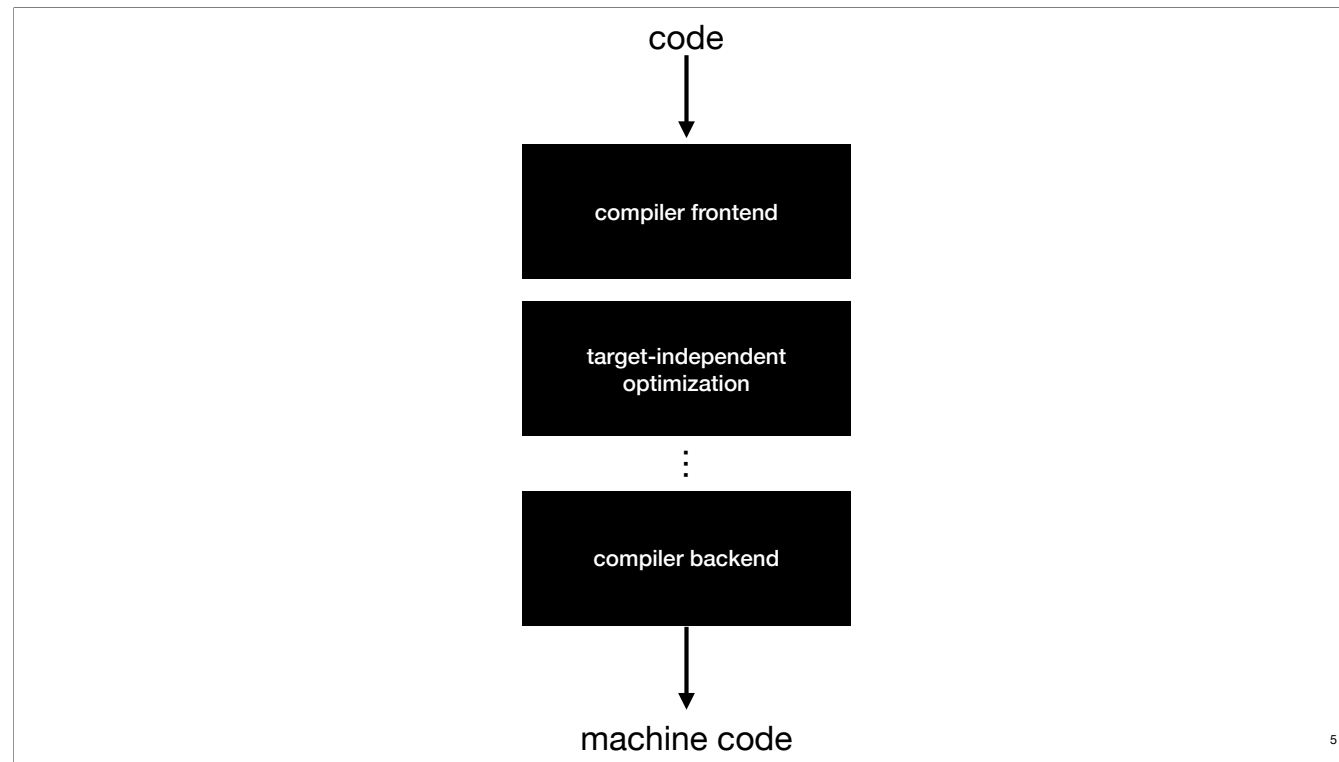
To discuss how we'll generate compiler backends from formal models of hardware, we first need to talk about compilers themselves.



So...this is a compiler.
What does a compiler do?



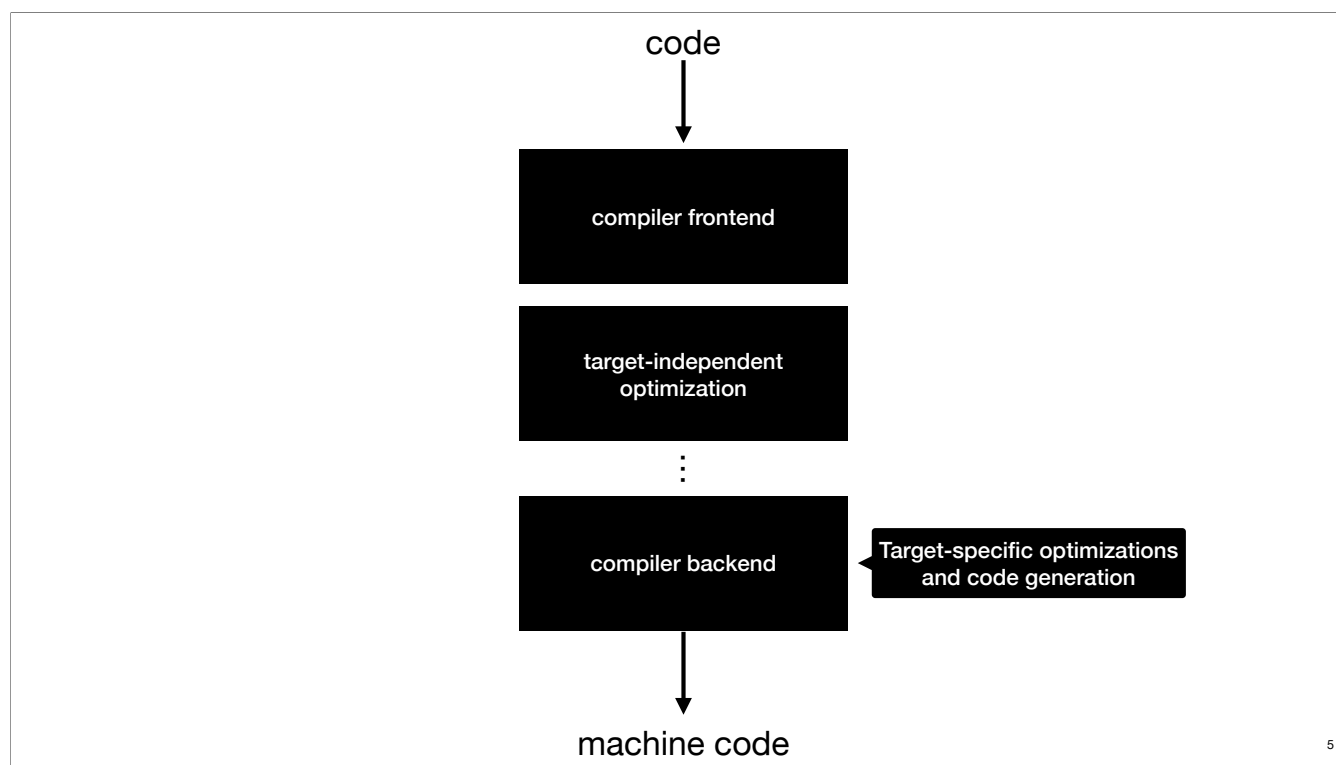
A compiler translates high-level code down to machine code that can be run on hardware.

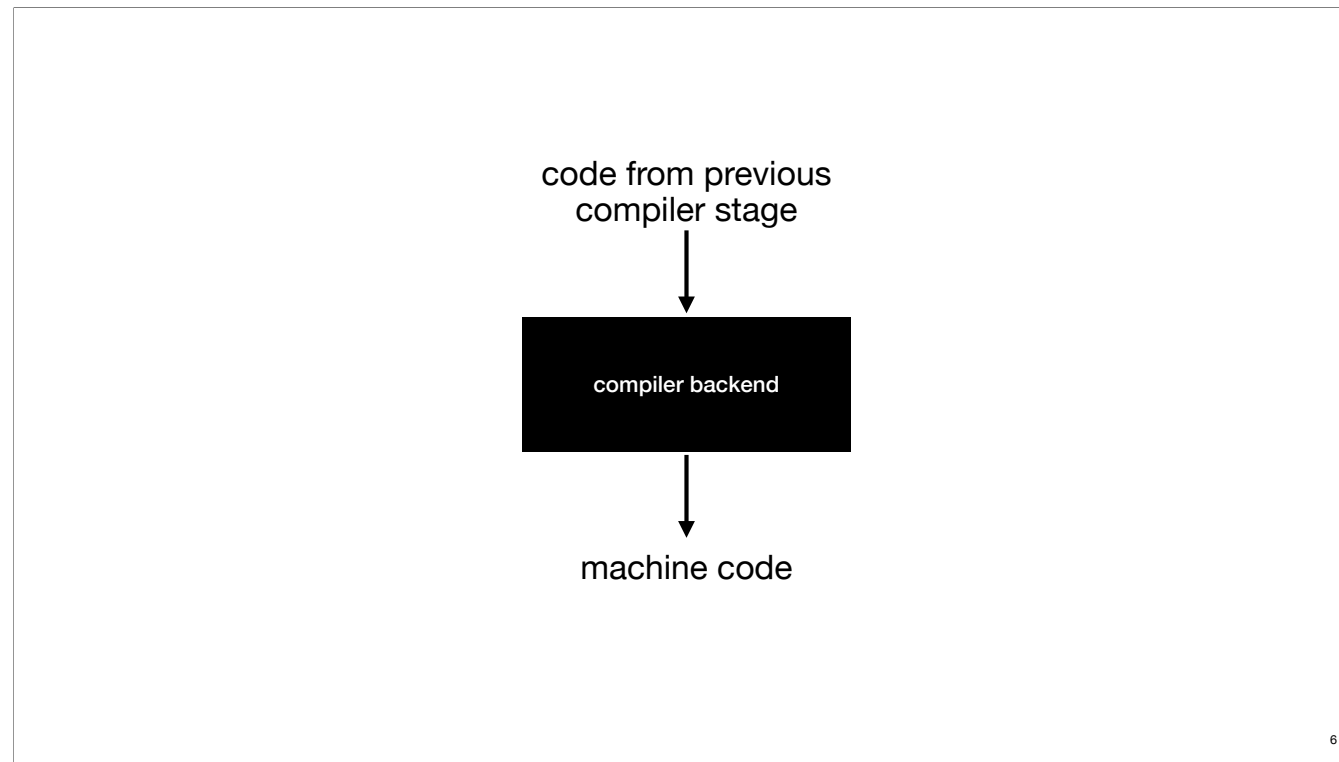


Compilers are generally composed of a few layers: a frontend, some target independent optimization, and finally, a backend.

(Build)

The compiler backend is what does target-specific optimization and code generation for the target hardware.





The compiler backend produces machine code,

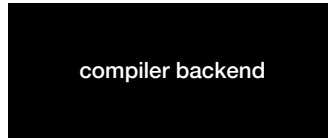
(Build)

Which is what runs on the target hardware.

(Build)

Our focus today will be on compiler backends and the hardware they target.

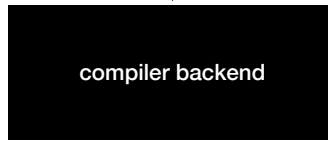
code from previous
compiler stage



machine code



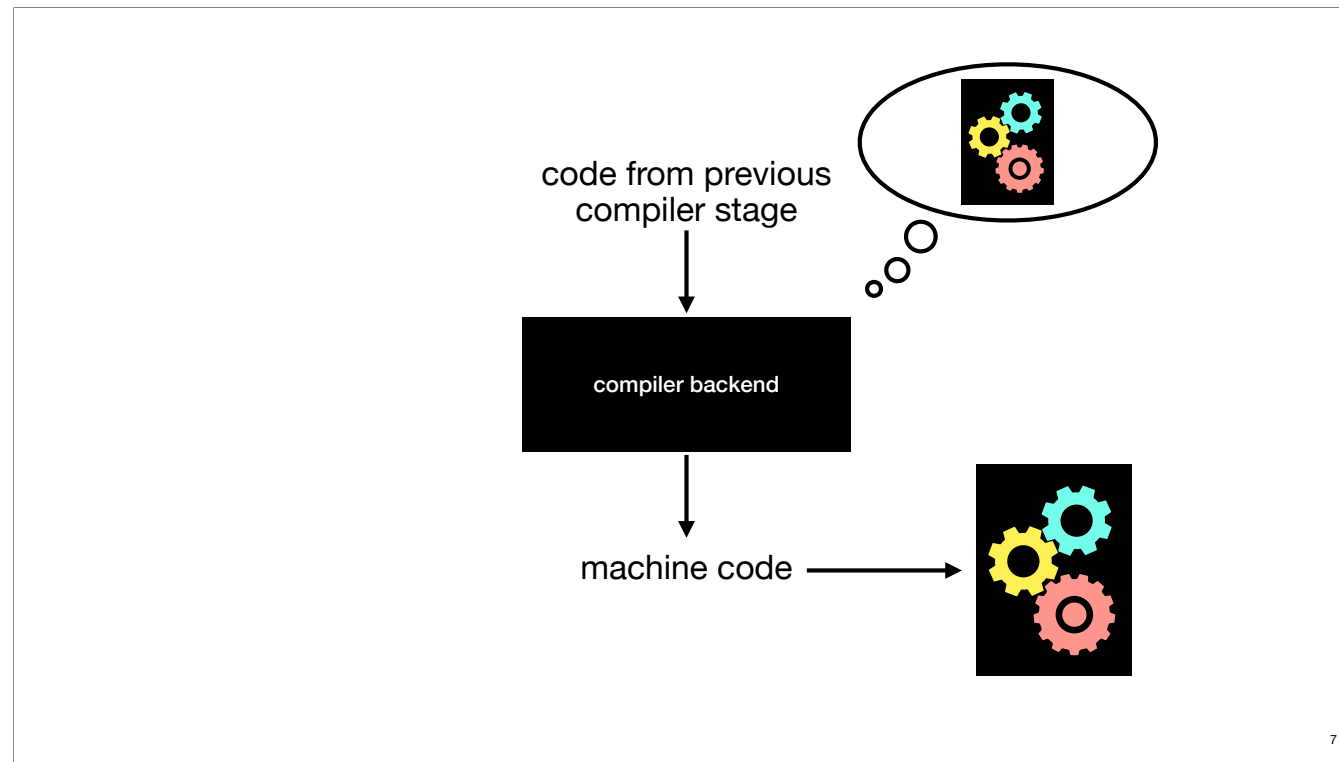
code from previous
compiler stage



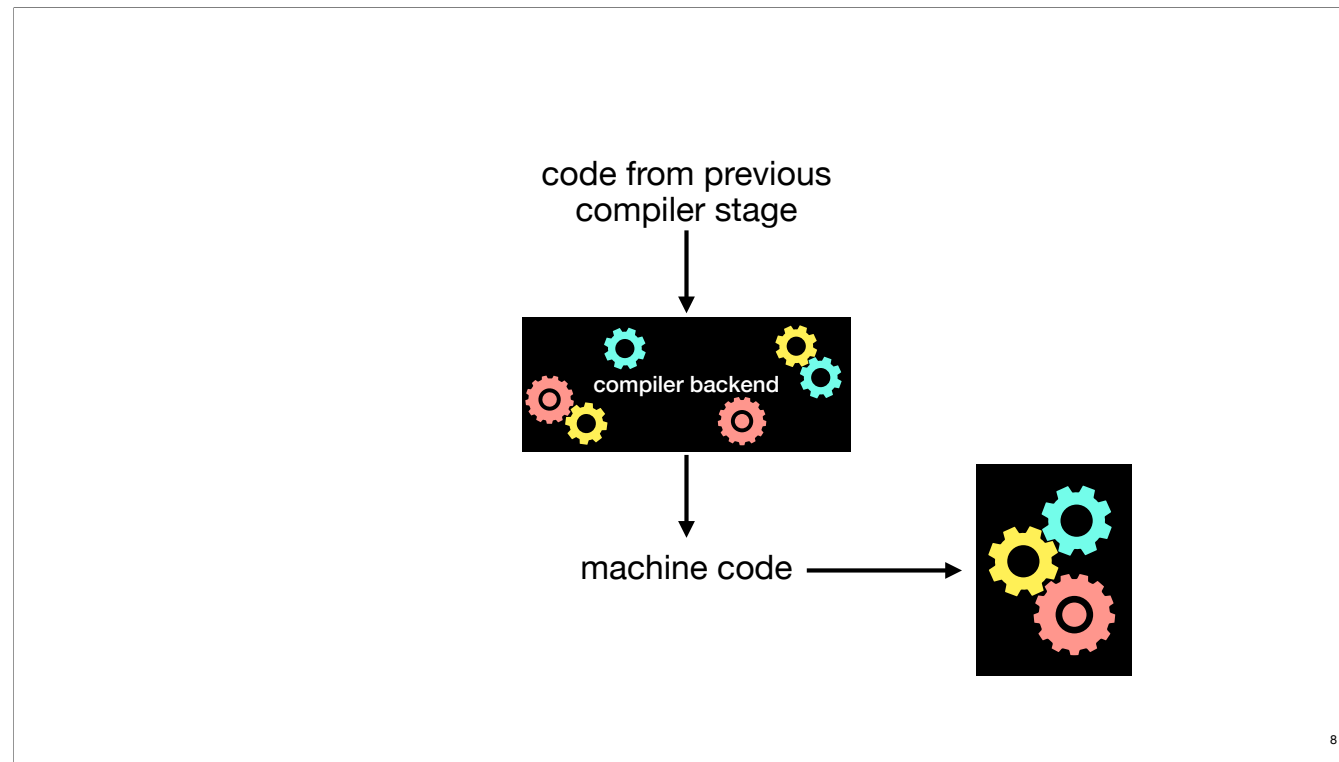
Our focus!

machine code





To compile to hardware, the compiler backend needs to know about the underlying hardware its compiling to.



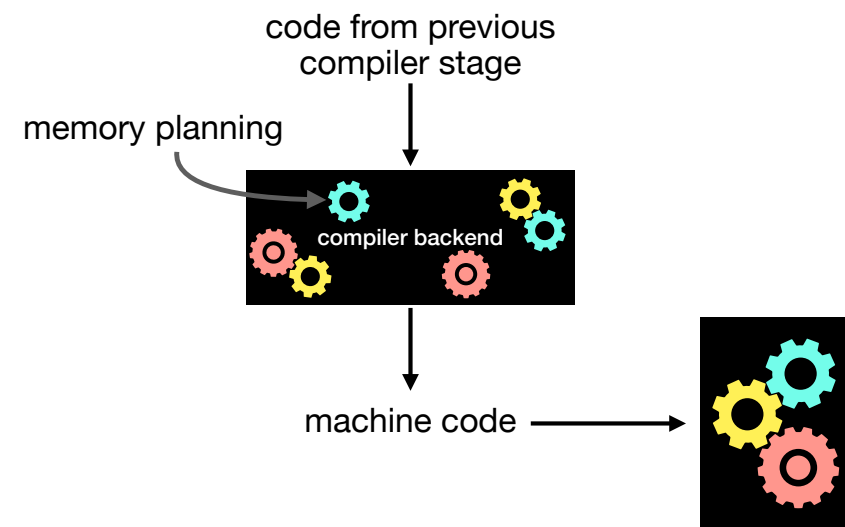
In reality, this is implemented by building models of the target hardware into the components of the compiler.

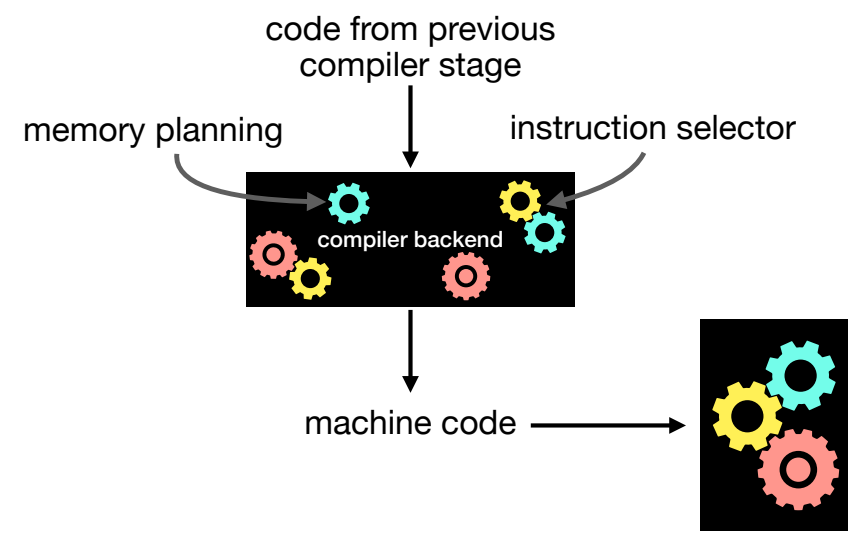
(Build)

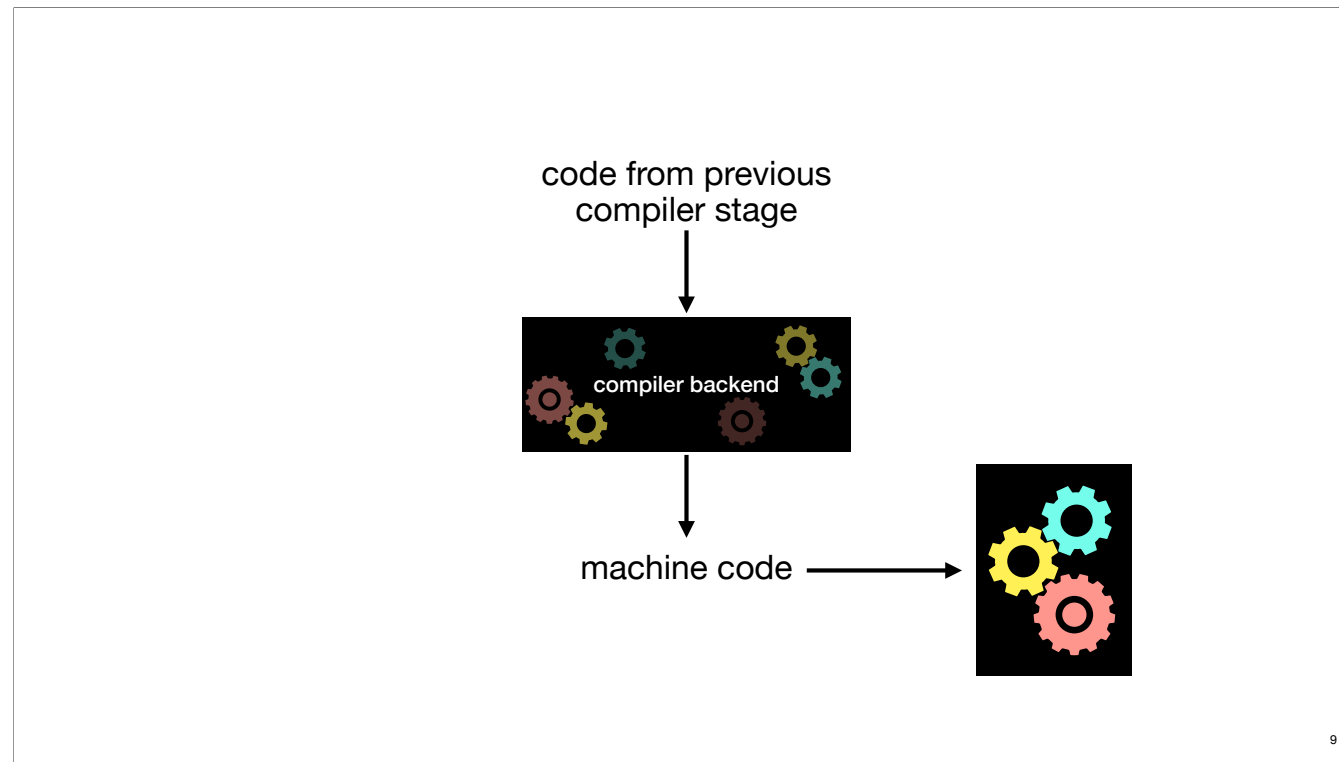
For example, the memory planning component of the compiler will contain a model of the hardware's memory hierarchy,

(Build)

While the instruction selector will contain a model of the hardware interface or ISA.







Importantly, though, these models of the underlying hardware are generally *implicit*. They are not explicitly coded descriptions or simulators of the underlying hardware, but instead, implicit descriptions like heuristics and hard-coded optimizations.

An Empirical Study of the Effect of Source-Level Loop Transformations on Compiler Stability

ZHANGXIAOWEN GONG, University of Illinois at Urbana-Champaign, USA

ZHI CHEN, University of California, Irvine, USA

JUSTIN SZADAY, University of Illinois at Urbana-Champaign, USA

DAVID WONG, Intel, USA

ZEHRA SURA, IBM, USA

NEFTALI WATKINSON, University of California, Irvine, USA

SAEED MALEKI, Microsoft, USA

DAVID PADUA, University of Illinois at Urbana-Champaign, USA

ALEXANDER VEIDENBAUM, University of California, Irvine, USA

ALEXANDRU NICOLAU, University of California, Irvine, USA

JOSEP TORRELLAS, University of Illinois at Urbana-Champaign, USA

Modern compiler optimization is a complex process that offers no guarantees to deliver the fastest, most efficient target code. For this reason, compilers struggle to produce a stable performance from versions of code that carry out the same computation and only differ in the order of operations. This instability makes compilers much less effective program optimization tools and often forces programmers to carry out a brute

10

In this paper, the authors perform a study of how various minor code transformations affect the quality of output of major compilers GCC, ICC, and Clang. They find that small changes in the input program lead to large changes in the performance of the compiled result. One of their conclusions is that

(Build)

Inaccurate vectorization models in these major compilers are a primary source of low-quality compiler results.

These vectorization models, which are heuristics for when code should be vectorized for a specific architecture, are a great example of implicit hardware models that exist in major compilers.

An Empirical Study of the Effect of Source-Level Loop Transformations on Compiler Stability

ZHANGXIAOWEN GONG, University of Illinois at Urbana-Champaign, USA

ZHANGXIAOWEN GONG, University of Illinois at Urbana-Champaign, USA
J. Inaccurate vectorization models are a primary source of low-quality compiler results!

DAVID WONG, Intel, USA

ZEHRA SURA, IBM, USA

NEFTALI WATKINSON, University of California, Irvine, USA

SAEED MALEKI, Microsoft, USA

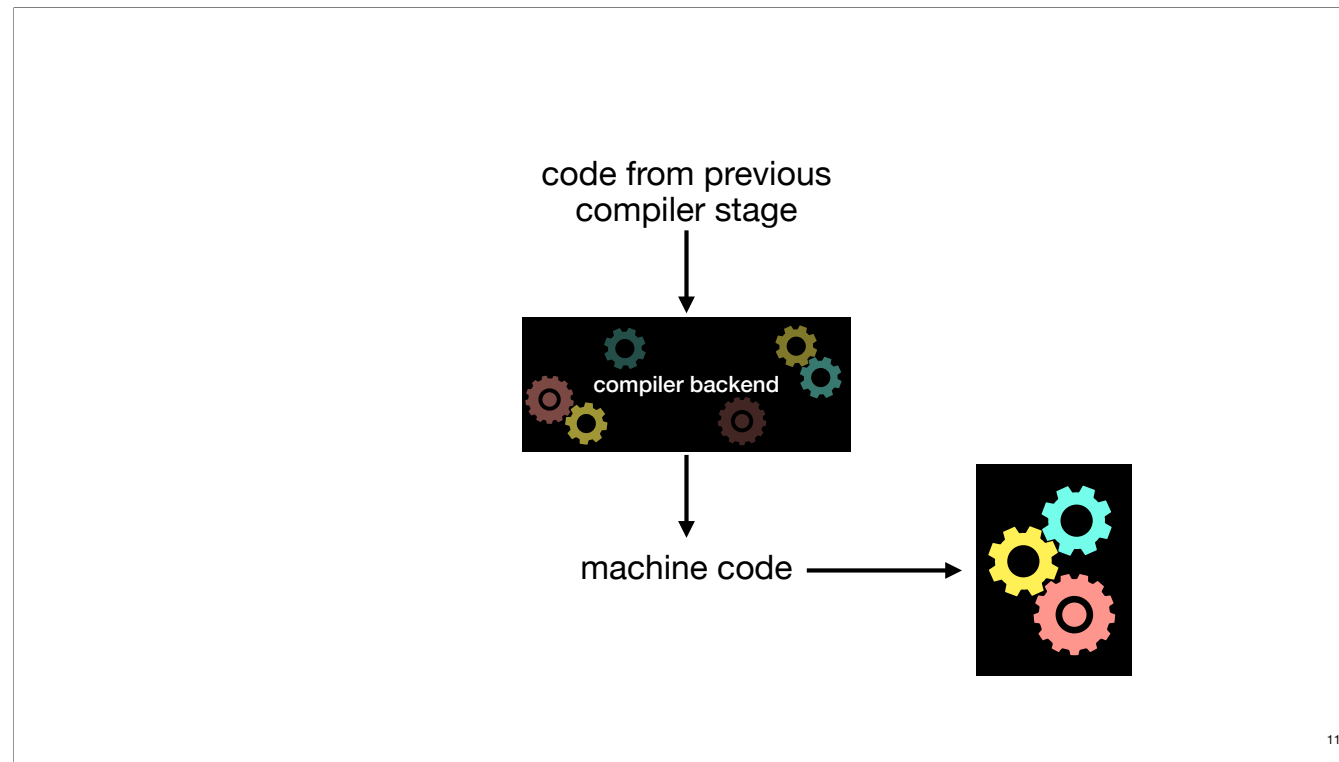
DAVID PADUA, University of Illinois at Urbana-Champaign, USA

ALEXANDER VEIDENBAUM, University of California, Irvine, USA

ALEXANDRU NICOLAU, University of California, Irvine, USA

JOSEP TORRELLAS, University of Illinois at Urbana-Champaign, USA

Modern compiler optimization is a complex process that offers no guarantees to deliver the fastest, most efficient target code. For this reason, compilers struggle to produce a stable performance from versions of code that carry out the same computation and only differ in the order of operations. This instability makes compilers much less effective program optimization tools and often forces programmers to carry out a brute

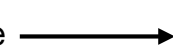


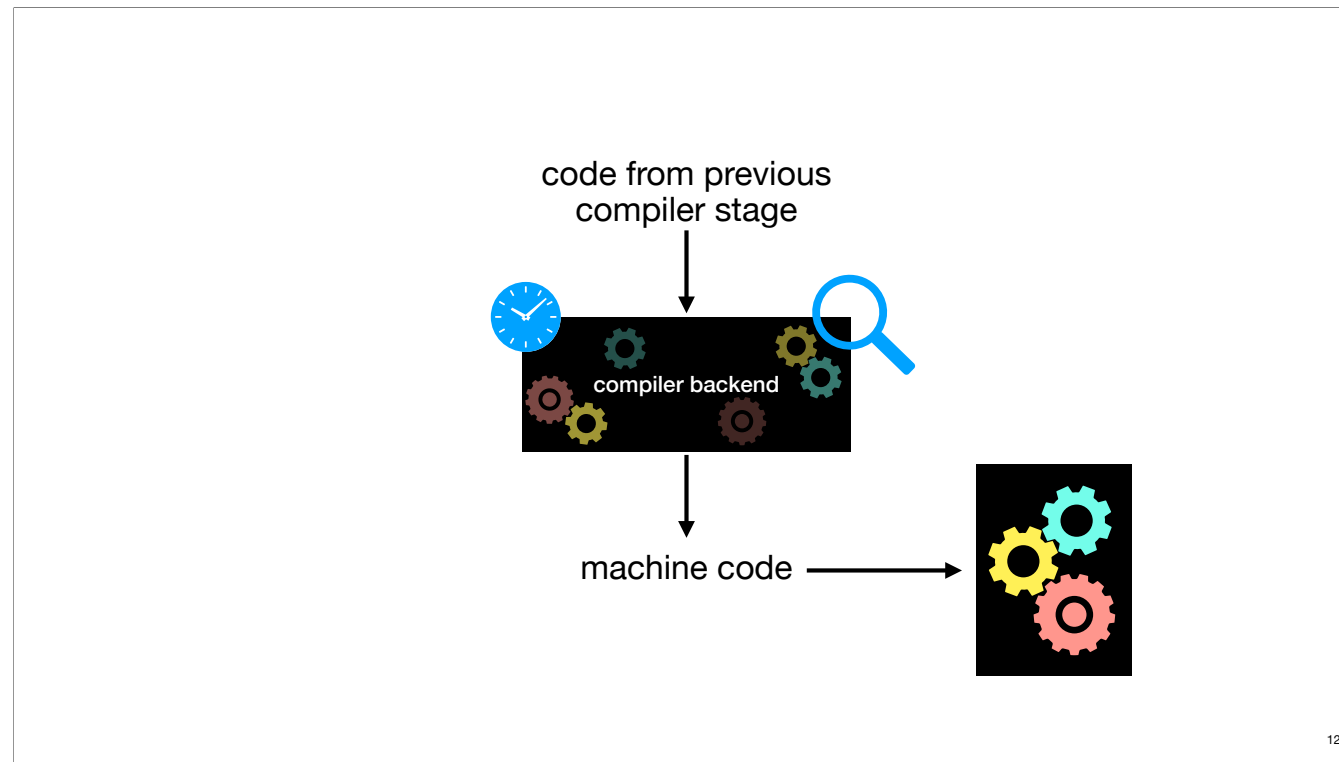
So, as this paper highlights, relying on implicit models in compilers can produce
(Build)
Imperfectly optimized, non-performant code.
But this is not the only downside of implicit models.

code from previous
compiler stage

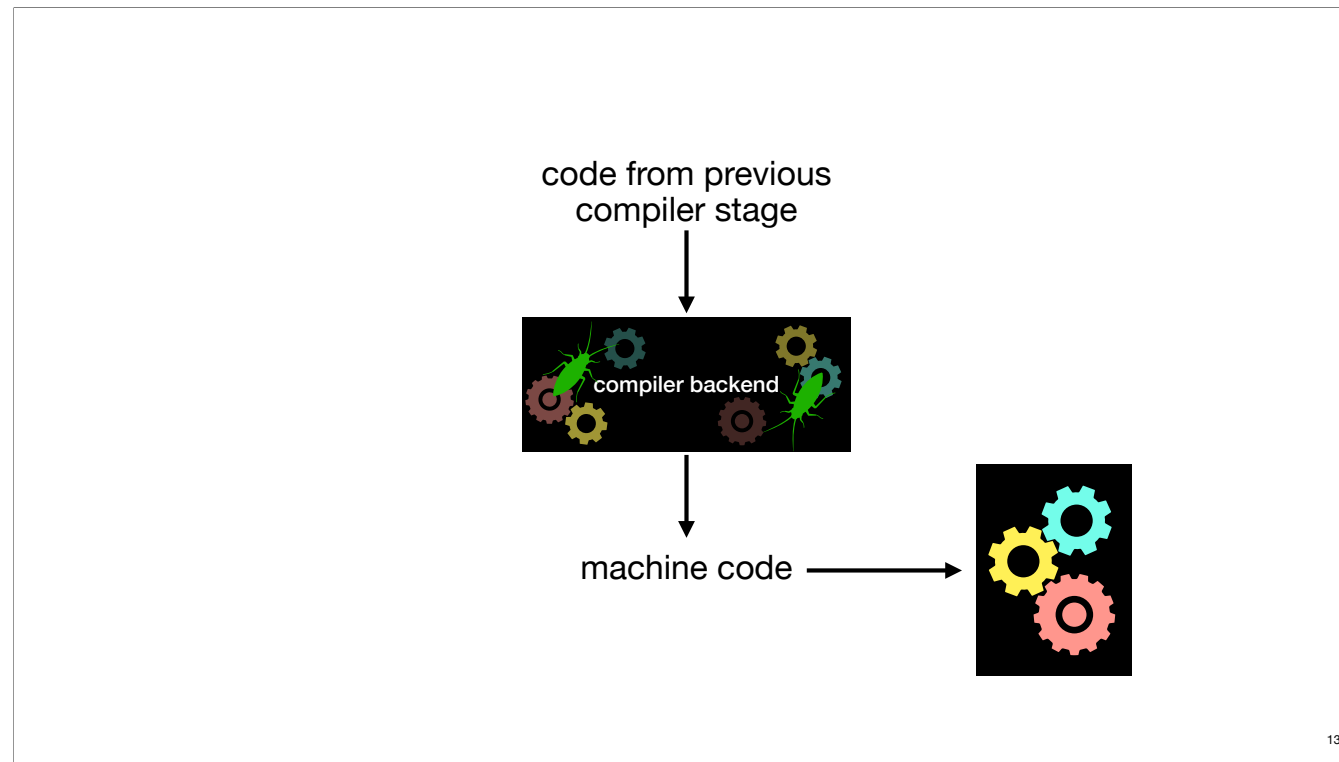


machine code

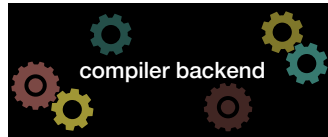




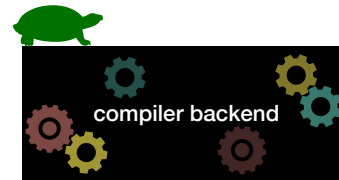
Implicit models also cause many headaches during development. Modifying implicit models often requires the time and attention of experts, as it can be unclear where modifications need to be made, and modifications can have unintended side affects.



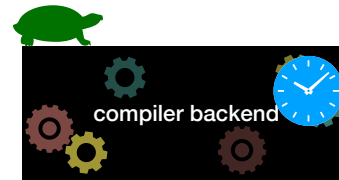
Lastly and most importantly, implicit models can be an insidious source of bugs. If the implicit models are incorrect, then the code they generate will be buggy. Worse still, fixing bugs in these implicit models is made harder by the fact that they are implicit.



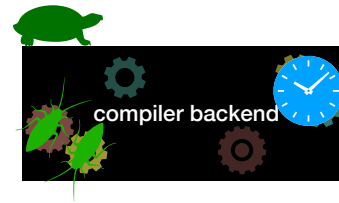
Implicit hardware models in compiler
backends are potential sources of



Implicit hardware models in compiler
backends are potential sources of
imperfect optimization,



Implicit hardware models in compiler backends are potential sources of imperfect optimization, difficulties in development,



Implicit hardware models in compiler backends are potential sources of imperfect optimization, difficulties in development, and hard-to-find bugs!

This leads directly to my thesis!

Automatically generating compiler backends from explicit, formal hardware models

16

I hypothesize that...

...that is, the optimizer is able to automatically compose small facts about the underlying hardware into large and complex optimizations.

**Automatically generating compiler backends
from explicit, formal hardware models**

- **gives rise to emergent optimizations,**

**Automatically generating compiler backends
from explicit, formal hardware models**

- gives rise to emergent optimizations,
- reduces development time, and

**Automatically generating compiler backends
from explicit, formal hardware models**

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

optimizer *discovers* optimizations that
are not explicitly programmed



This thesis is very much so in line with a lot of Norman Ramsey’s work on automatically generated instruction selectors.
(Build)
However, unlike Ramsey, we will be applying these ideas not on CPUs, but on fixed function accelerators, and programmable hardware.

Automatically Generating Instruction Selectors
Using Declarative Machine Descriptions

João Dias
Tufts University
dias@cs.tufts.edu

Norman Ramsey

Machine Descriptions to Build Tools for
Embedded Systems

Norman Ramsey and Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
norman.ramsey@cs.virginia.edu jwd@cs.virginia.edu

Unlike Ramsey, we will focus not on CPUs, but on fixed-function accelerators and programmable hardware!

Abstract

Despite years of work on retargetable compilers, creating a good, reliable back end for an optimizing compiler still entails a lot of hard work. Moreover, a critical component of the back end—the instruction selector—must be written by a person who is expert in both the compiler’s intermediate code and the target machine’s instruction set. By *generating* the instruction selector from declarative machine descriptions we have (a) made it unnecessary for one person to be both a compiler expert and a machine expert, and (b) made creating an optimizing back end easier than ever before.

Our achievement rests on two new results. First, finding a mapping from intermediate code to machine code is an undecidable problem. Second, using heuristic search, we can find mappings for machines of practical interest in at most a few minutes of CPU time.

Our most significant new idea is that heuristic search should be controlled by algebraic laws. Laws are used not only to show when a sequence of instructions implements part of an intermediate code, but also to limit the search: we drop a sequence of instructions not when it gets too long or when it computes too complicated a result, but when *too much reasoning* will be required to show that the result computed might be useful.

Categories and Subject Descriptors D.3.4 [Processors]: Code generation; D.3.4 [Processors]: Retargetable compilers

General Terms Algorithms, Experimentation, Theory

technique, an instruction selector is generated automatically from *declarative machine descriptions*. (A declarative machine description contains no code and no information about any compiler’s data structures; instead, it simply and formally describes properties of a target machine.)

Our contributions are as follows:

- We show that given a description of an arbitrary instruction set, generating an instruction selector is undecidable (Section 8). To find machine instructions that implement intermediate code, it is therefore necessary to search heuristically.
- We present a new heuristic search algorithm, which starts with the expressions computed by the machine’s instruction set and gradually adds to a pool of computable expressions until every intermediate-code expression is computable.

A crucial invariant is that we consider *only* computations that we *know* can be implemented entirely by machine instructions. This invariant makes our algorithm significantly simpler than earlier search algorithms, which start with goal computations whose implementations by machine instructions are not known.

- To increase the pool of computable expressions, we rewrite existing computable expressions using algebraic laws. To match the left-hand side of an algebraic law, we have developed a new algorithm called *establishment*, which uses a novel combination of unification and machine code to make two expressions equal (Section 7, especially Figure 4).

Abstract. Because of poor tools, developing embedded systems can be unnecessarily hard. Machine descriptions based on register-transfer lists (RTLs) have proven useful in building retargetable compilers, but not in building other retargetable tools. Simulators, assemblers, linkers, debuggers, and profilers are built by hand if at all—previous machine descriptions have lacked the detail and precision needed to generate them. This paper presents detailed and precise machine-description techniques that are based on a new formalization of RTLs. Unlike previous notations, these RTLs have a detailed, unambiguous, and machine-independent semantics, which makes them ideal for supporting automatic generation of retargetable tools. The paper also gives examples of λ -RTL, a notation that makes it possible for human beings to read and write RTLs without becoming overwhelmed by machine-dependent detail.

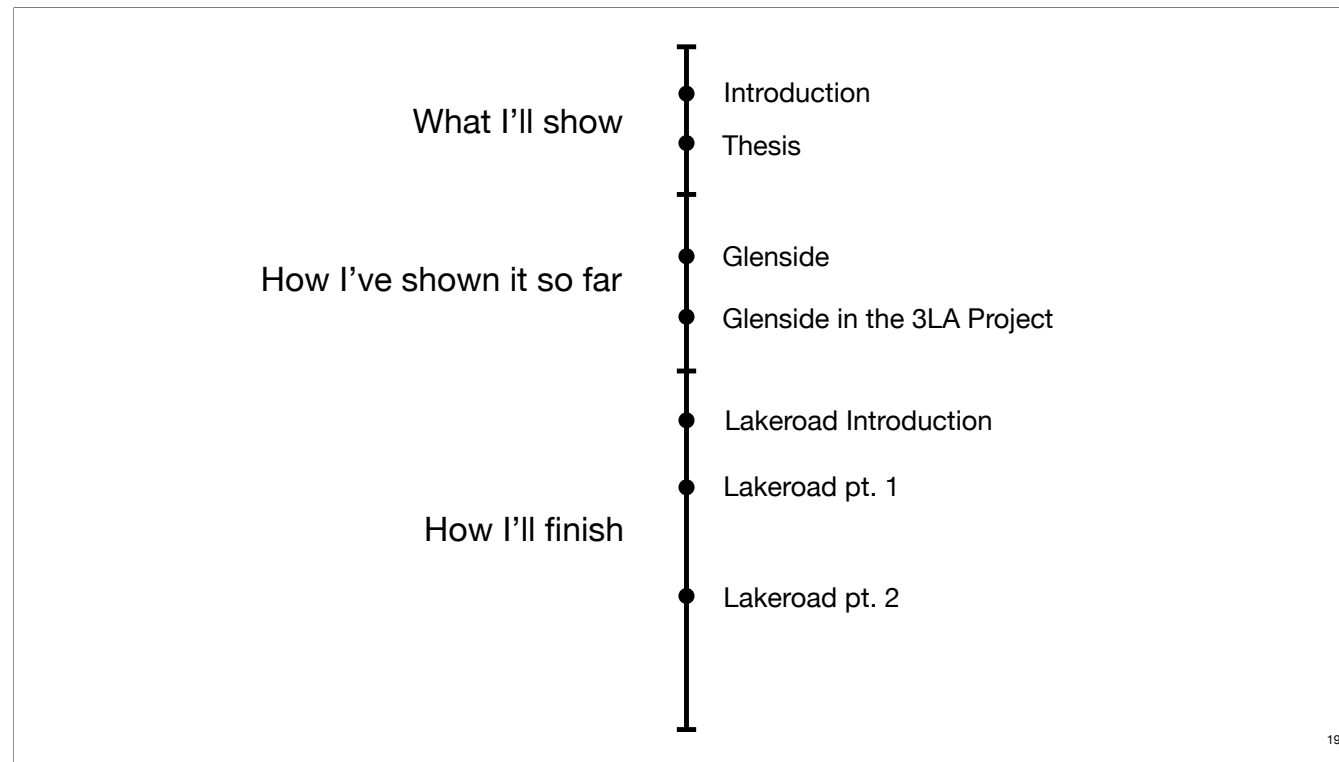
Machine Descriptions for Machine-Level Tools

Tools for embedded systems often work without the benefit of the best available tools. Embedded systems can have unusual architectural features, and processors can be introduced rapidly. Development is typically done on processors, and cross-development can make it hard to get basic compilation, linking, and debugging tools. These tools are often developed by hand, and the development process is often tedious and error-prone. This paper presents a new formalization of RTLs, which makes it possible to generate RTLs automatically from machine descriptions. The new RTLs have a detailed, unambiguous, and machine-independent semantics, which makes them ideal for supporting automatic generation of retargetable tools. The paper also gives examples of λ -RTL, a notation that makes it possible for human beings to read and write RTLs without becoming overwhelmed by machine-dependent detail.

Structure of this talk

18

So let's discuss the structure of this talk.



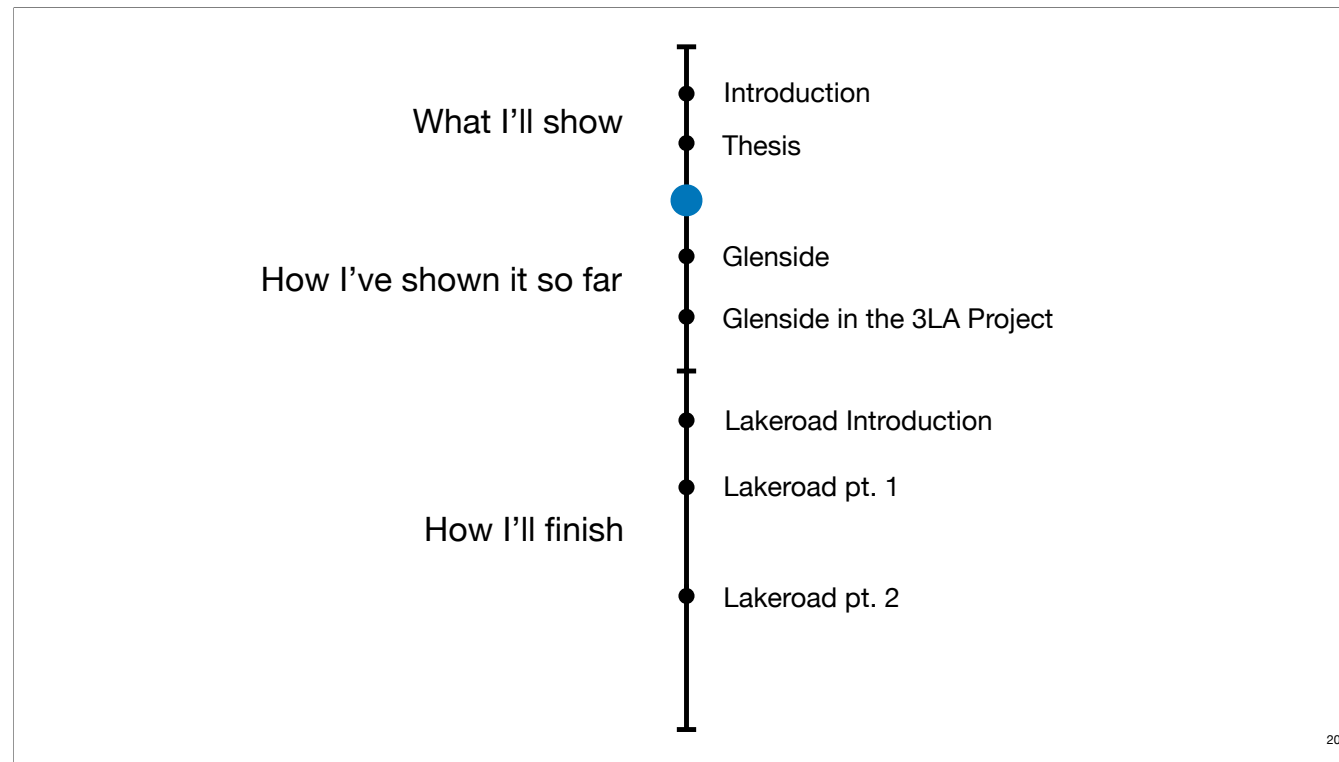
I've broken this talk into three primary sections:

What I'll show: namely, my stated thesis,

How I've shown it so far, which is the work I've already done towards demonstrating this thesis,

And

How I'll finish, which is where I propose my final project.

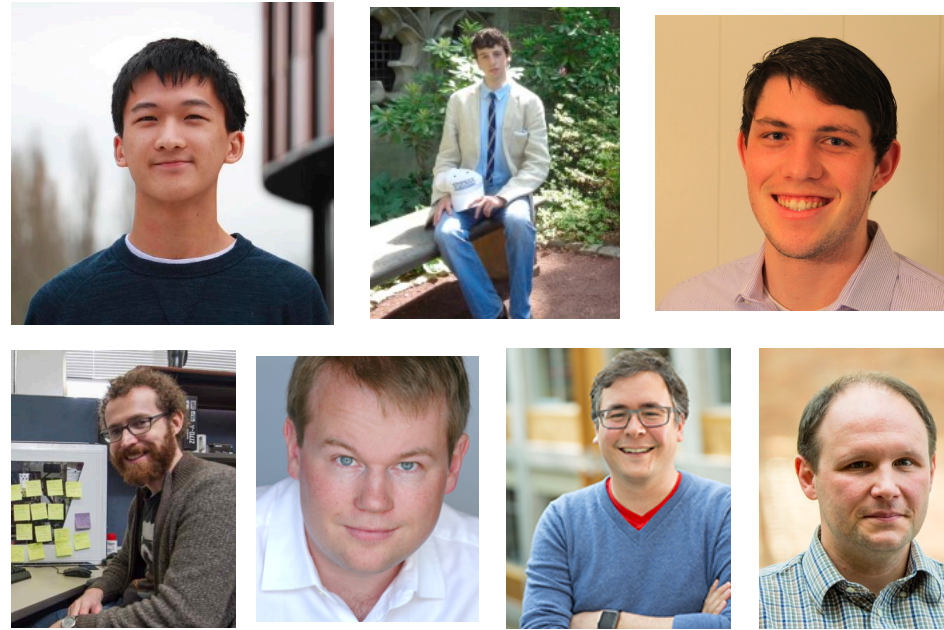


So I just finished explaining what I will show;
Now, let's jump into how I've shown it so far.

Glenside

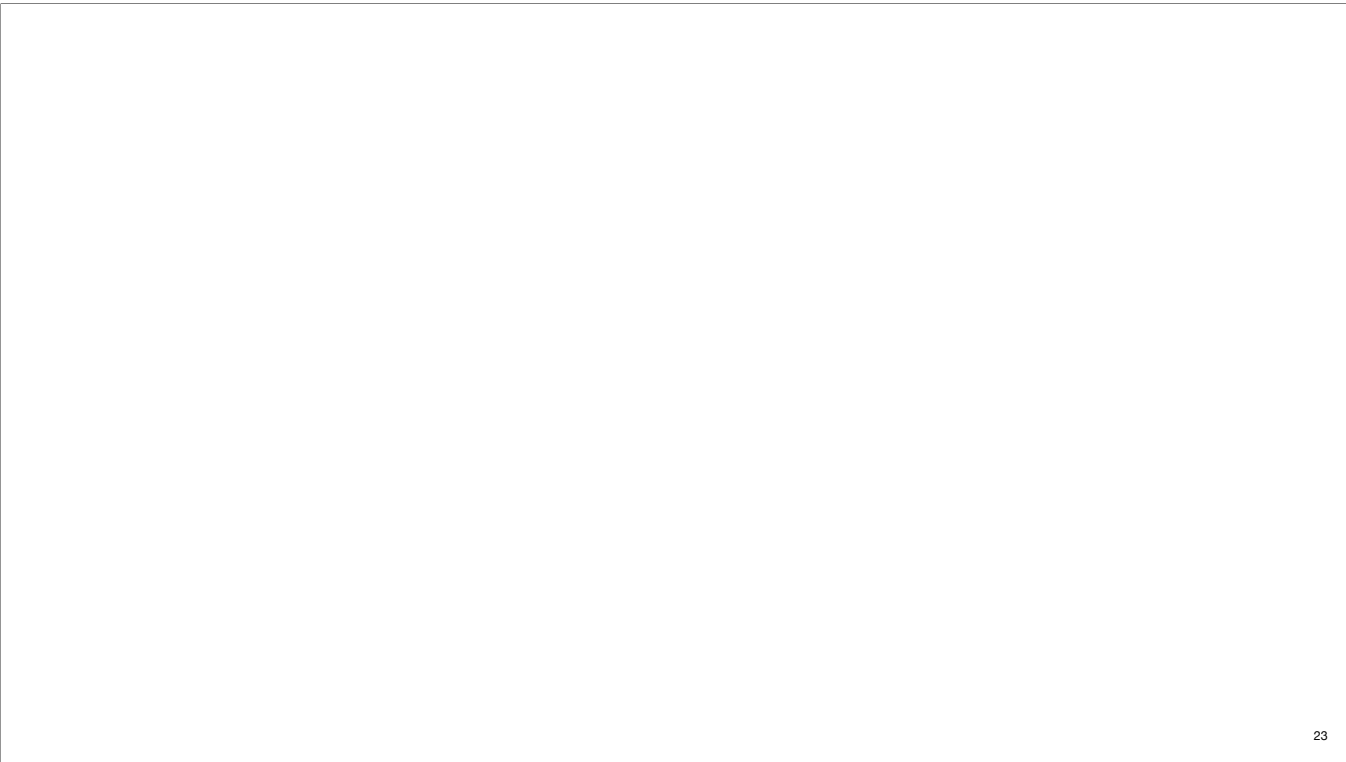
21

With that, let's talk about glenside.



Gus Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock.
"Pure tensor program rewriting via access patterns (representation pearl)." MAPS 2021.

This work was done with my colleagues here at UW and was published at MAPS 2021.



Because we’re about to get a bit into the weeds, I want to first summarize the high-level story of Glenside.

Glenside is a tensor IR* built for equality saturation.

* intermediate representation

Glenside is a tensor IR* built for equality saturation.

Glenside enables users to model hardware accelerators as program rewrites.

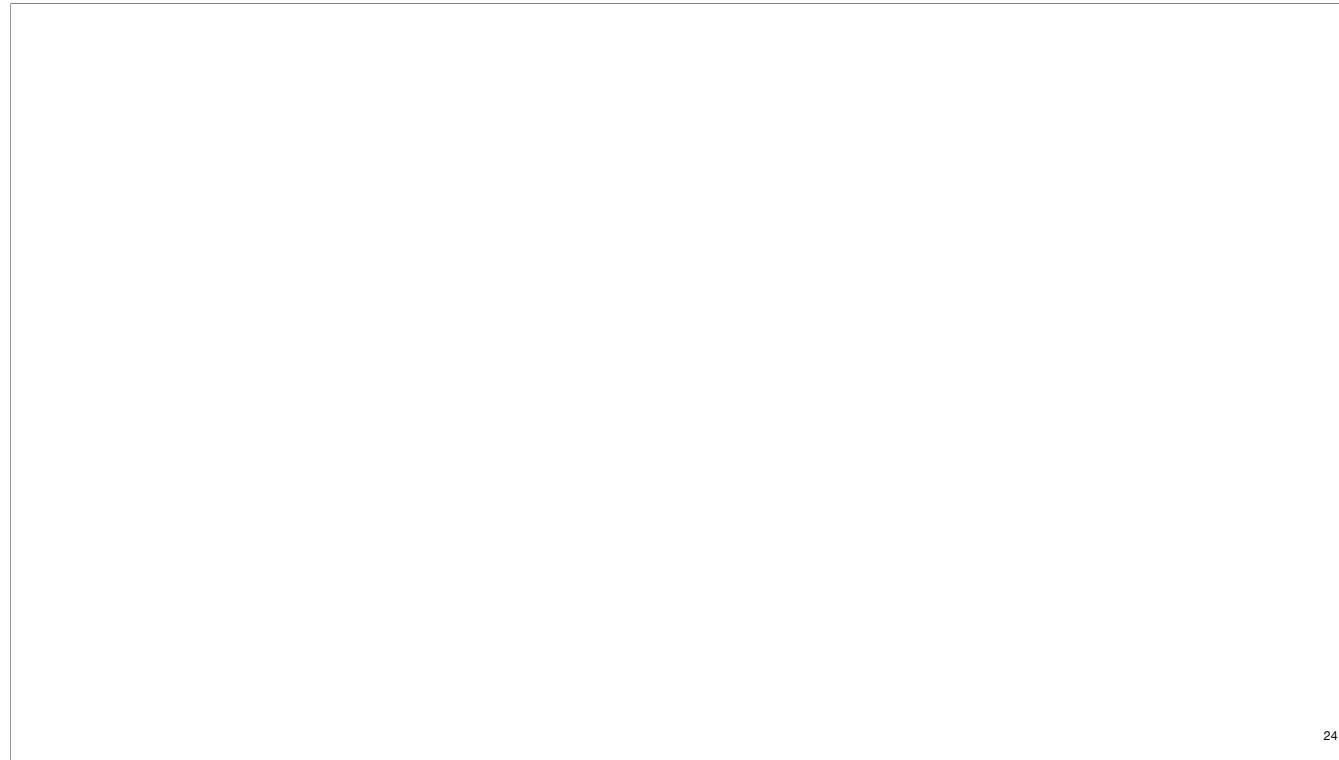
* intermediate representation

Glenside is a tensor IR* built for equality saturation.

Glenside enables users to model hardware accelerators as program rewrites.

These rewrites, in concert with Glenside's built-in rewrites, automatically discover ways to map machine learning workloads to accelerators.

* intermediate representation



When we began designing glenside, we had three primary requirements

First, the language must be pure,

Second, the language must be low-level, so that we can actually reason about hardware.

And lastly, the language should avoid binding, which makes term rewriting much easier.

Three design requirements for Glenside:

Three design requirements for Glenside:

1. The language must be **pure**—a necessary requirement for equality saturation.

Three design requirements for Glenside:

1. The language must be **pure**—a necessary requirement for equality saturation.
2. The language must be **low-level**, letting us reason about hardware.

Three design requirements for Glenside:

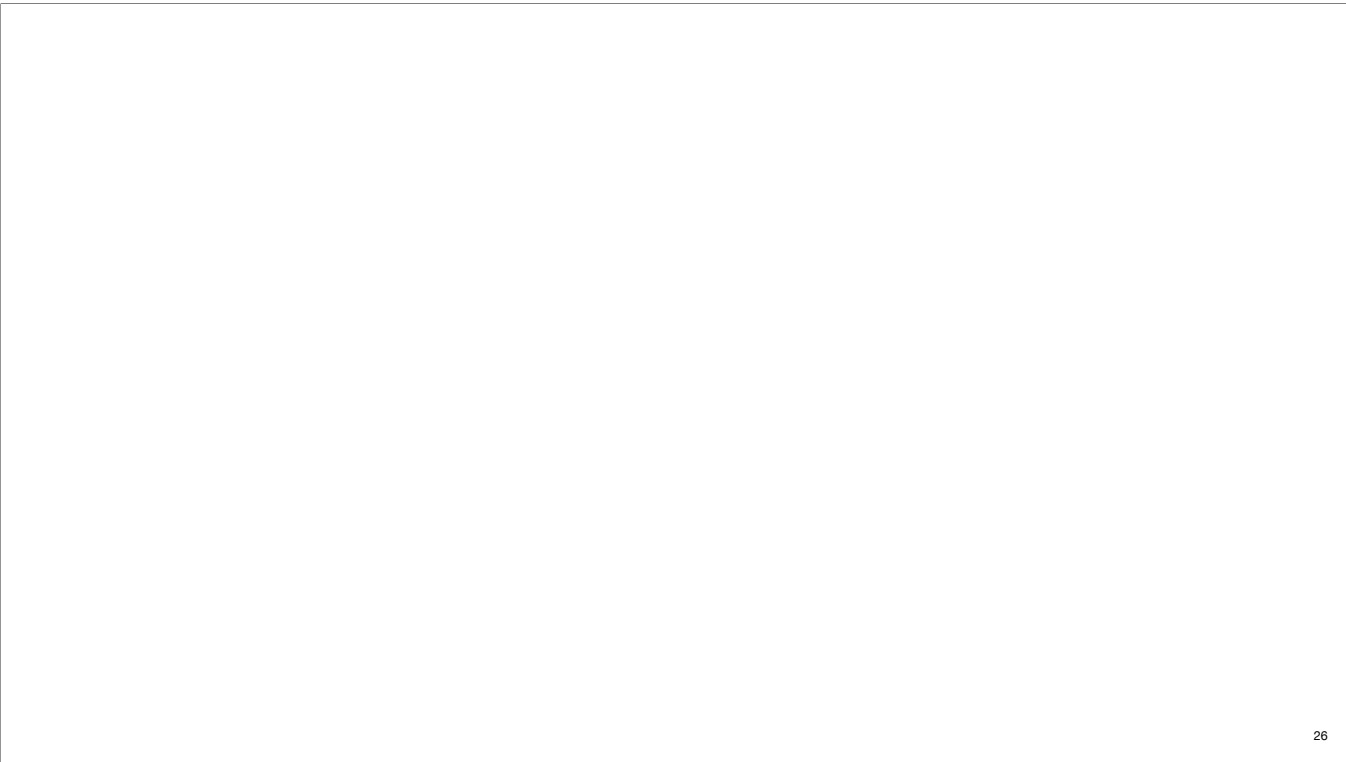
1. The language must be **pure**—a necessary requirement for equality saturation.
2. The language must be **low-level**, letting us reason about hardware.
3. The language must **not use binding**, making term rewriting much easier.

**Let's begin with an example:
matrix multiplication!**

25

So, as always, I think it's easiest to start with an example.

So let's begin with the most common kernel in deep learning — matrix multiplication!



Remember, our goals are to represent multiplication in a way that...

We want to represent matrix multiplication in a way that

We want to represent matrix multiplication in a way that

1. is pure,

We want to represent matrix multiplication in a way that

1. is pure,
2. is low-level, and

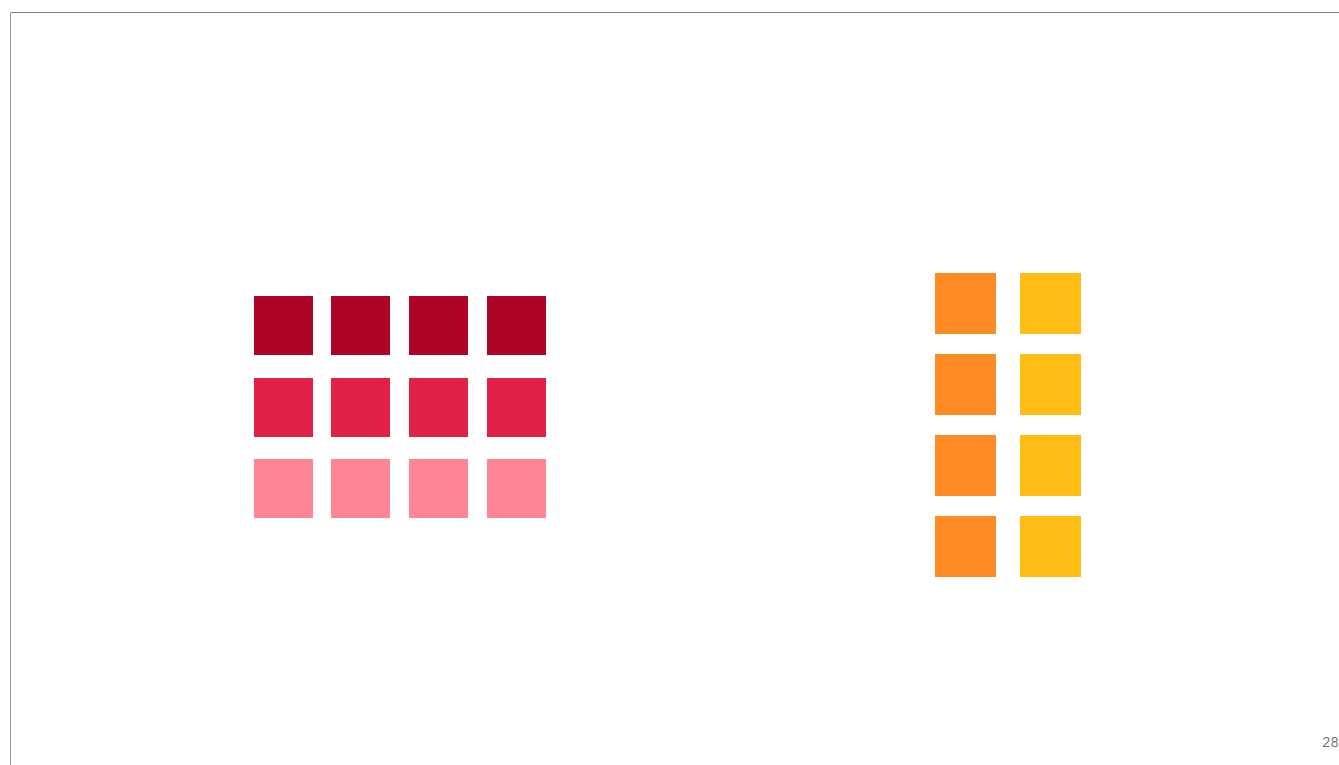
We want to represent matrix multiplication in a way that

1. is pure,
2. is low-level, and
3. avoids binding.

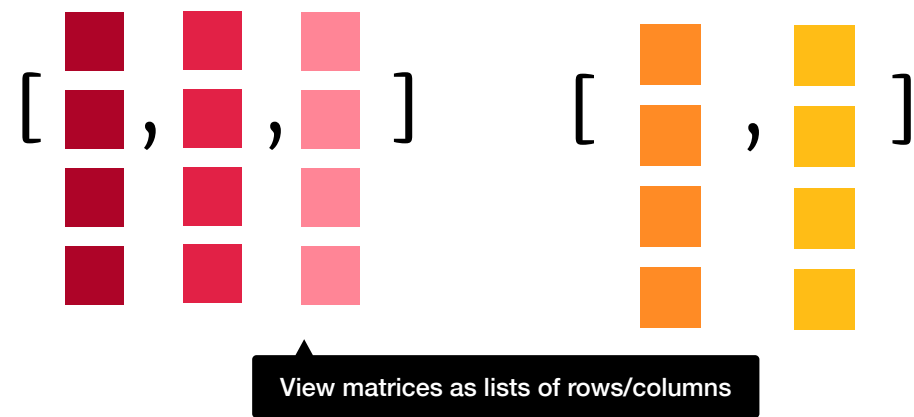
Given matrices A and B, pair each row of A with each column of B, compute their dot products, and arrange the results back into a matrix.

27

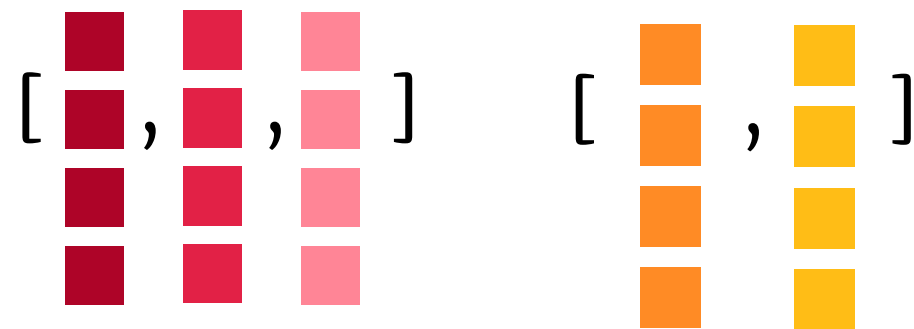
Just to remind ourselves, here is the matrix multiplication algorithm. Given matrices A and B...



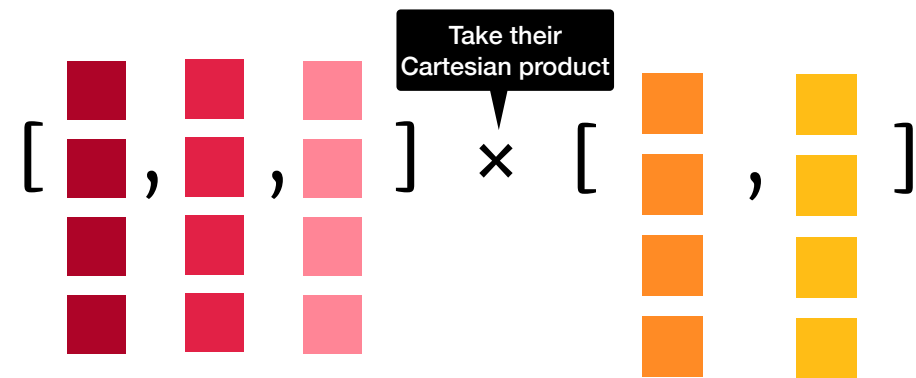
This simple English description of the algorithm immediately suggests a possible pure, low-level, binder-free implementation.



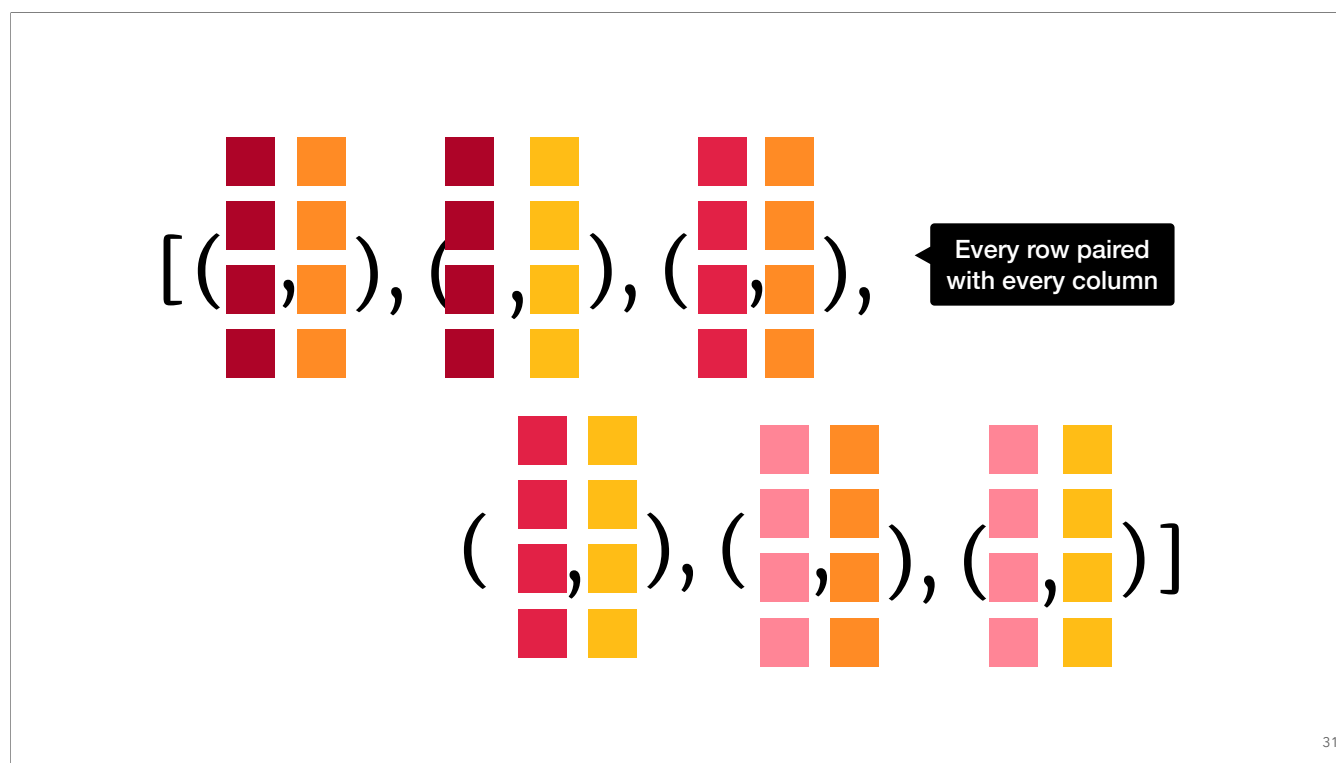
First, we view the matrix A, on the left, as a list of its rows, and the matrix B, on the right, as a list of its columns.



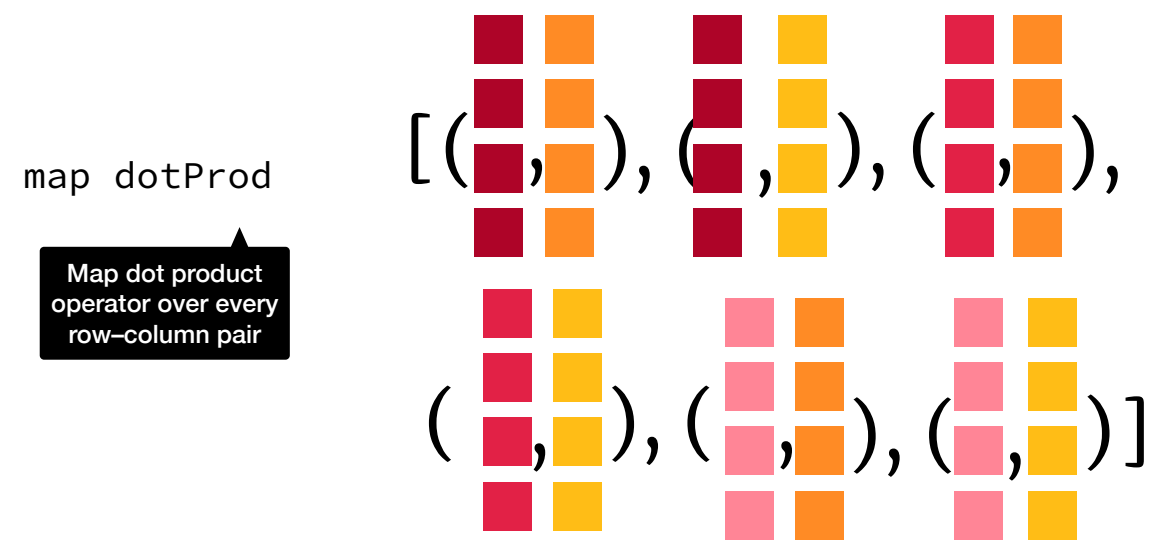
View matrices as lists of rows/columns









We then pair each row from A with each column from B by taking the Cartesian product of the two lists.



The result is a list of row-column pairs.



Finally, we map the dot product operator over this list of row-column pairs.

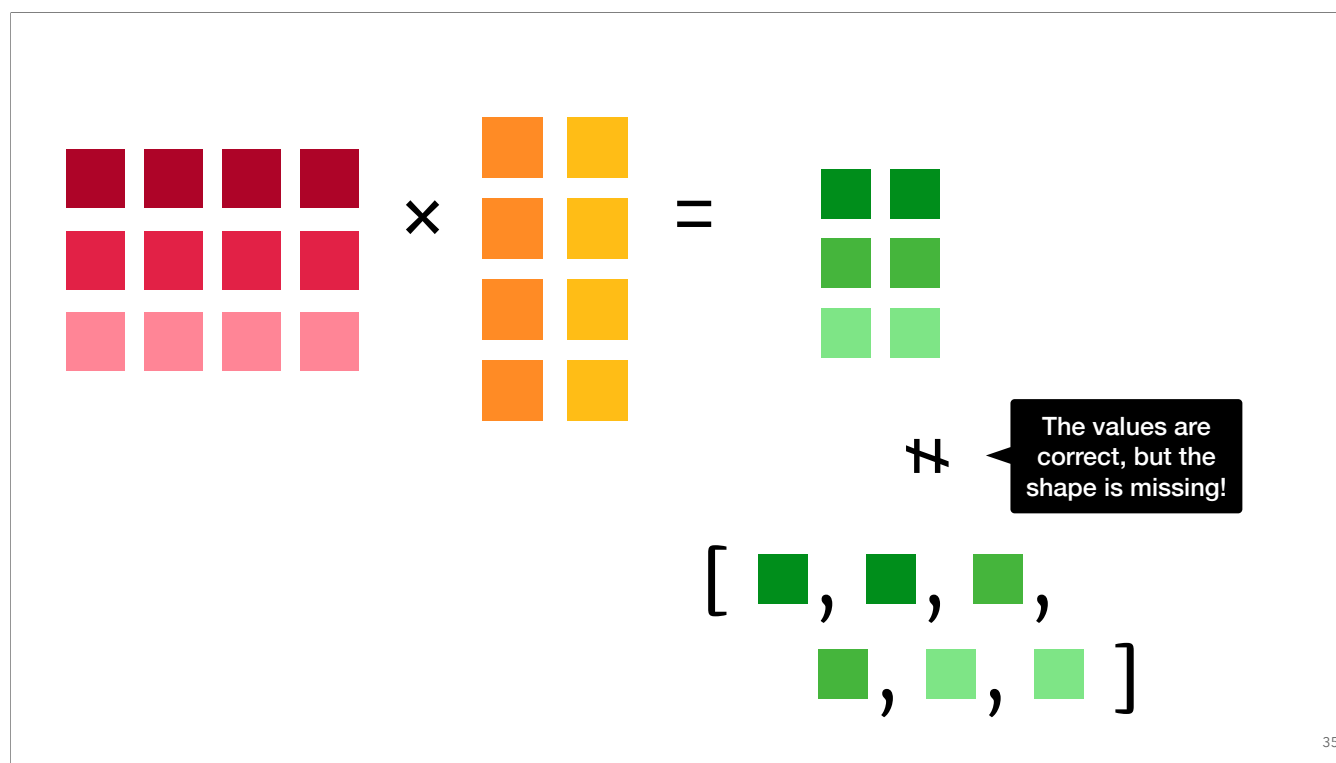
[, , ,
 , , ]

Our result is a list of scalar values!







But there's a problem!

34

But there's a problem



A 2D matrix times a 2D matrix should produce a 2D matrix.
However, our algorithm produces a one-dimensional list of values!
Even if the values are correct, we're missing a core piece of information: the shape!

[, , ,
, , ]

Let's step back in time to understand where the shape information got lost.



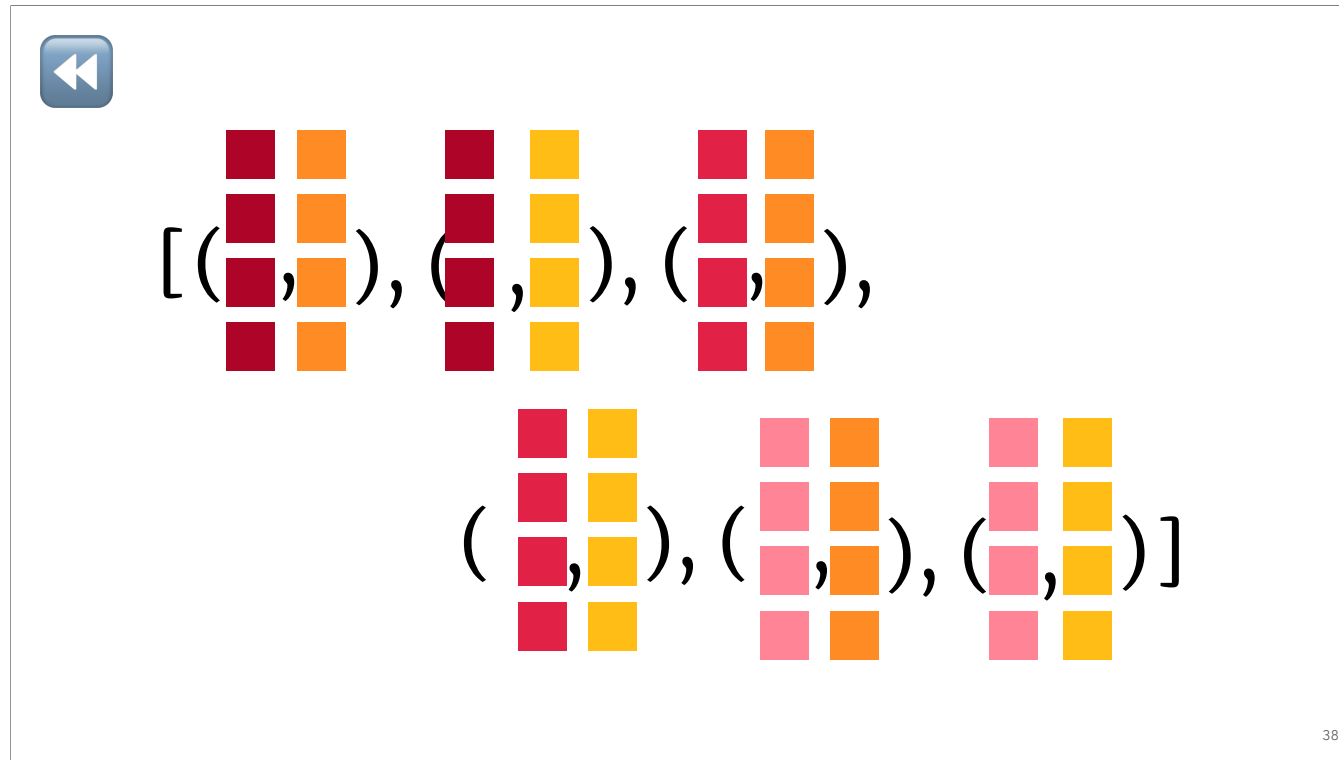
[■, ■, ■,
■, ■, ■]



map dot-product $\left[\left(\begin{array}{cc} \text{red} & \text{orange} \\ \text{red} & \text{orange} \\ \text{red} & \text{orange} \\ \text{red} & \text{orange} \end{array} \right), \left(\begin{array}{cc} \text{red} & \text{yellow} \\ \text{red} & \text{yellow} \\ \text{red} & \text{yellow} \\ \text{red} & \text{yellow} \end{array} \right), \left(\begin{array}{cc} \text{red} & \text{orange} \\ \text{red} & \text{orange} \\ \text{red} & \text{orange} \\ \text{red} & \text{orange} \end{array} \right), \right.$

$\left. \left(\begin{array}{cc} \text{red} & \text{yellow} \\ \text{red} & \text{yellow} \\ \text{red} & \text{yellow} \\ \text{red} & \text{yellow} \end{array} \right), \left(\begin{array}{cc} \text{pink} & \text{orange} \\ \text{pink} & \text{orange} \\ \text{pink} & \text{orange} \\ \text{pink} & \text{orange} \end{array} \right), \left(\begin{array}{cc} \text{pink} & \text{yellow} \\ \text{pink} & \text{yellow} \\ \text{pink} & \text{yellow} \\ \text{pink} & \text{yellow} \end{array} \right) \right]$

By the time we're mapping the dot product operator over our list, the list is already a flat list of pairs.



As is the case in the step before.



$$\begin{bmatrix} \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \end{bmatrix} \times \begin{bmatrix} \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \end{bmatrix}$$

But at this step, just before we take the cartesian product, all of the original shape information seems to be present. Specifically, we still have A and B in their two dimensional forms, as a list of rows and columns, respectively. So shape information is present here, but when we step forward again...



$$\begin{bmatrix} \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \end{bmatrix} \times \begin{bmatrix} \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \end{bmatrix}$$



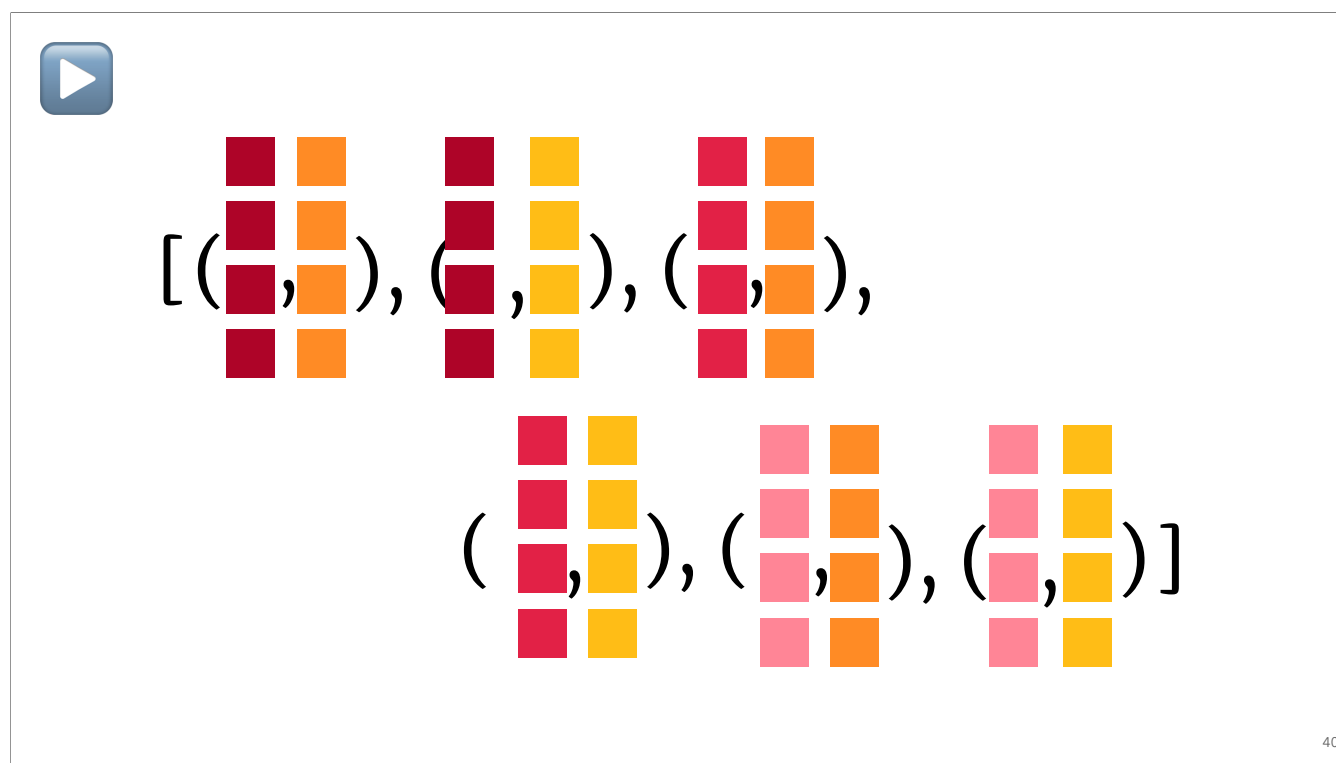
Shape information
is present here...

$$\begin{bmatrix} \text{dark red} & \text{red} & \text{light pink} \\ \text{dark red} & \text{red} & \text{light pink} \\ \text{dark red} & \text{red} & \text{light pink} \\ \text{dark red} & \text{red} & \text{light pink} \end{bmatrix} \times \begin{bmatrix} \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \end{bmatrix}$$



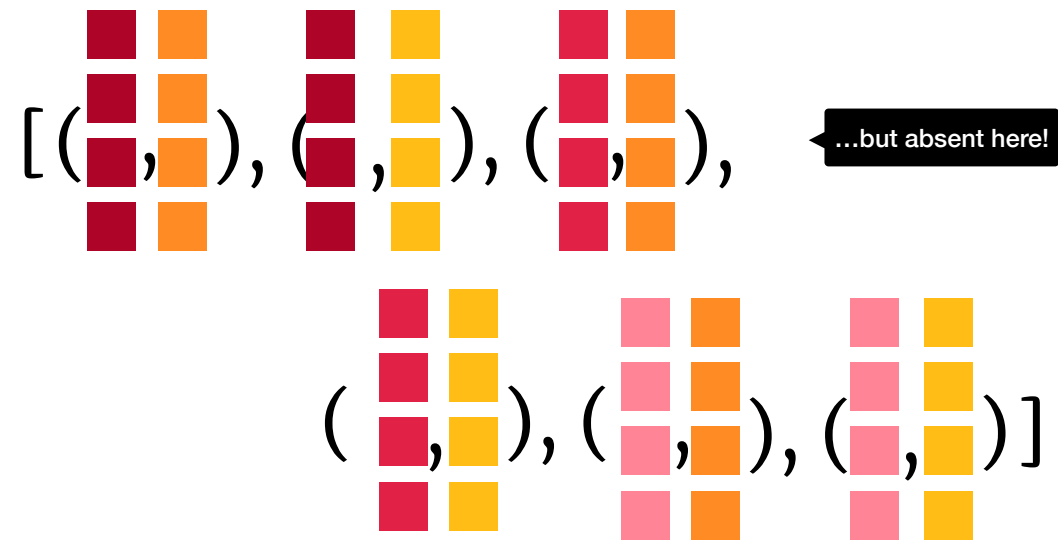
Shape information
is present here...

$$\begin{bmatrix} \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \\ \text{dark red} & \text{red} & \text{pink} \end{bmatrix} \times \begin{bmatrix} \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \\ \text{orange} & \text{yellow} \end{bmatrix}$$



...the shape information is lost.

Our 2D matrices have been flattened into a one-dimensional list, with no way to recover the shapes of the original matrices.



**Cartesian product destroys our
shape information!**

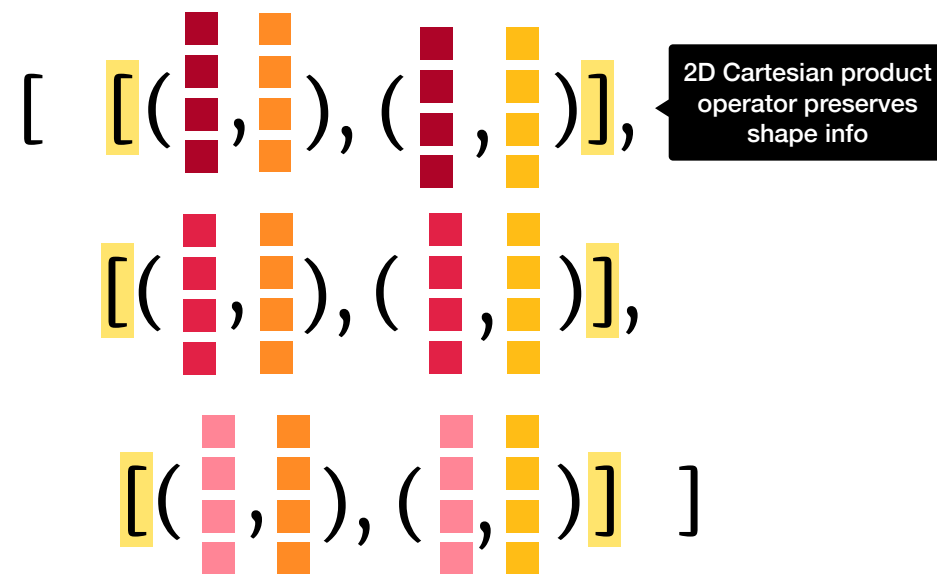
41

So it seems that the Cartesian product operator is destroying our shape information!

We introduce a new Cartesian product operator

$$\left[\begin{array}{c} \text{dark red} \\ \text{dark red} \\ \text{dark red} \\ \text{dark red} \end{array}, \begin{array}{c} \text{red} \\ \text{red} \\ \text{red} \\ \text{red} \end{array}, \begin{array}{c} \text{pink} \\ \text{pink} \\ \text{pink} \\ \text{pink} \end{array} \right] \times_{2D} \left[\begin{array}{c} \text{orange} \\ \text{orange} \\ \text{orange} \\ \text{orange} \end{array}, \begin{array}{c} \text{yellow} \\ \text{yellow} \\ \text{yellow} \\ \text{yellow} \end{array} \right]$$

Thus, we introduce a new, two dimensional Cartesian product operator.



43

This new cartesian product operator preserves shape information.

Now, the cartesian product of the three rows of A and the two columns of B are a three-by-two matrix of pairs.

map dotProd





$$[[(\begin{bmatrix} \text{dark red} \\ \text{dark red} \\ \text{dark red} \\ \text{dark red} \end{bmatrix}, \begin{bmatrix} \text{orange} \\ \text{orange} \\ \text{orange} \\ \text{orange} \end{bmatrix}), (\begin{bmatrix} \text{dark red} \\ \text{dark red} \\ \text{dark red} \\ \text{dark red} \end{bmatrix}, \begin{bmatrix} \text{yellow} \\ \text{yellow} \\ \text{yellow} \\ \text{yellow} \end{bmatrix})],$$

$$[(\begin{bmatrix} \text{pink} \\ \text{pink} \\ \text{pink} \\ \text{pink} \end{bmatrix}, \begin{bmatrix} \text{orange} \\ \text{orange} \\ \text{orange} \\ \text{orange} \end{bmatrix}), (\begin{bmatrix} \text{pink} \\ \text{pink} \\ \text{pink} \\ \text{pink} \end{bmatrix}, \begin{bmatrix} \text{yellow} \\ \text{yellow} \\ \text{yellow} \\ \text{yellow} \end{bmatrix})],$$





$$[(\begin{bmatrix} \text{pink} \\ \text{pink} \\ \text{pink} \\ \text{pink} \end{bmatrix}, \begin{bmatrix} \text{orange} \\ \text{orange} \\ \text{orange} \\ \text{orange} \end{bmatrix}), (\begin{bmatrix} \text{pink} \\ \text{pink} \\ \text{pink} \\ \text{pink} \end{bmatrix}, \begin{bmatrix} \text{yellow} \\ \text{yellow} \\ \text{yellow} \\ \text{yellow} \end{bmatrix})]]$$





But now, when we try to map our dot product operator over our new list, we run into another problem.



[dotProd [(, )], (, )],

But now, map
operator maps over
wrong dimension!

dotProd [(, )], (, )],

dotProd [(, )], (, )]]

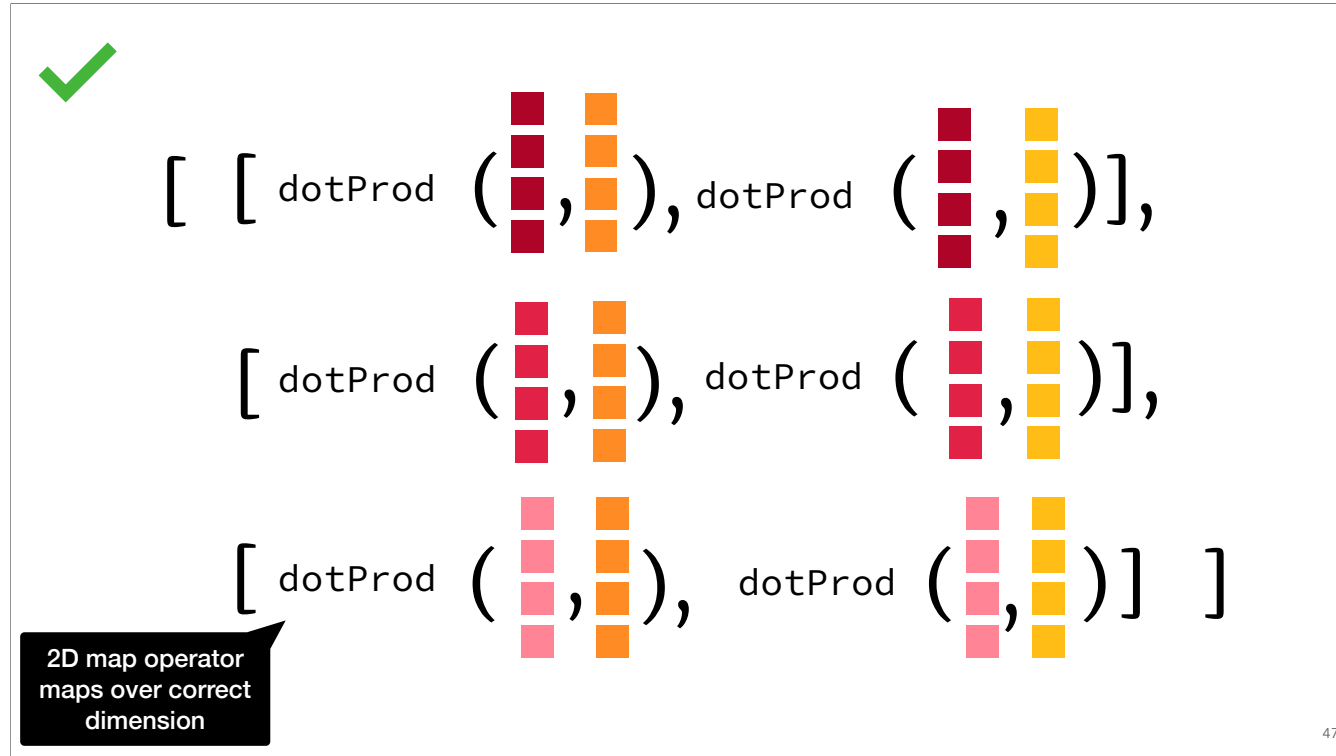
Our map operator maps the dot product over the wrong dimension!

map2D dotProd

We also need a
new map operator

$$\begin{aligned} & [[(\begin{smallmatrix} \color{darkred} \square \\ \color{darkred} \square \\ \color{darkred} \square \\ \color{darkred} \square \end{smallmatrix}, \begin{smallmatrix} \color{orange} \square \\ \color{orange} \square \\ \color{orange} \square \\ \color{orange} \square \end{smallmatrix}), (\begin{smallmatrix} \color{darkred} \square \\ \color{darkred} \square \\ \color{darkred} \square \\ \color{darkred} \square \end{smallmatrix}, \begin{smallmatrix} \color{yellow} \square \\ \color{yellow} \square \\ \color{yellow} \square \\ \color{yellow} \square \end{smallmatrix})], \\ & [(\begin{smallmatrix} \color{red} \square \\ \color{red} \square \\ \color{red} \square \\ \color{red} \square \end{smallmatrix}, \begin{smallmatrix} \color{orange} \square \\ \color{orange} \square \\ \color{orange} \square \\ \color{orange} \square \end{smallmatrix}), (\begin{smallmatrix} \color{red} \square \\ \color{red} \square \\ \color{red} \square \\ \color{red} \square \end{smallmatrix}, \begin{smallmatrix} \color{yellow} \square \\ \color{yellow} \square \\ \color{yellow} \square \\ \color{yellow} \square \end{smallmatrix})], \\ & [(\begin{smallmatrix} \color{pink} \square \\ \color{pink} \square \\ \color{pink} \square \\ \color{pink} \square \end{smallmatrix}, \begin{smallmatrix} \color{orange} \square \\ \color{orange} \square \\ \color{orange} \square \\ \color{orange} \square \end{smallmatrix}), (\begin{smallmatrix} \color{pink} \square \\ \color{pink} \square \\ \color{pink} \square \\ \color{pink} \square \end{smallmatrix}, \begin{smallmatrix} \color{yellow} \square \\ \color{yellow} \square \\ \color{yellow} \square \\ \color{yellow} \square \end{smallmatrix})]] \end{aligned}$$

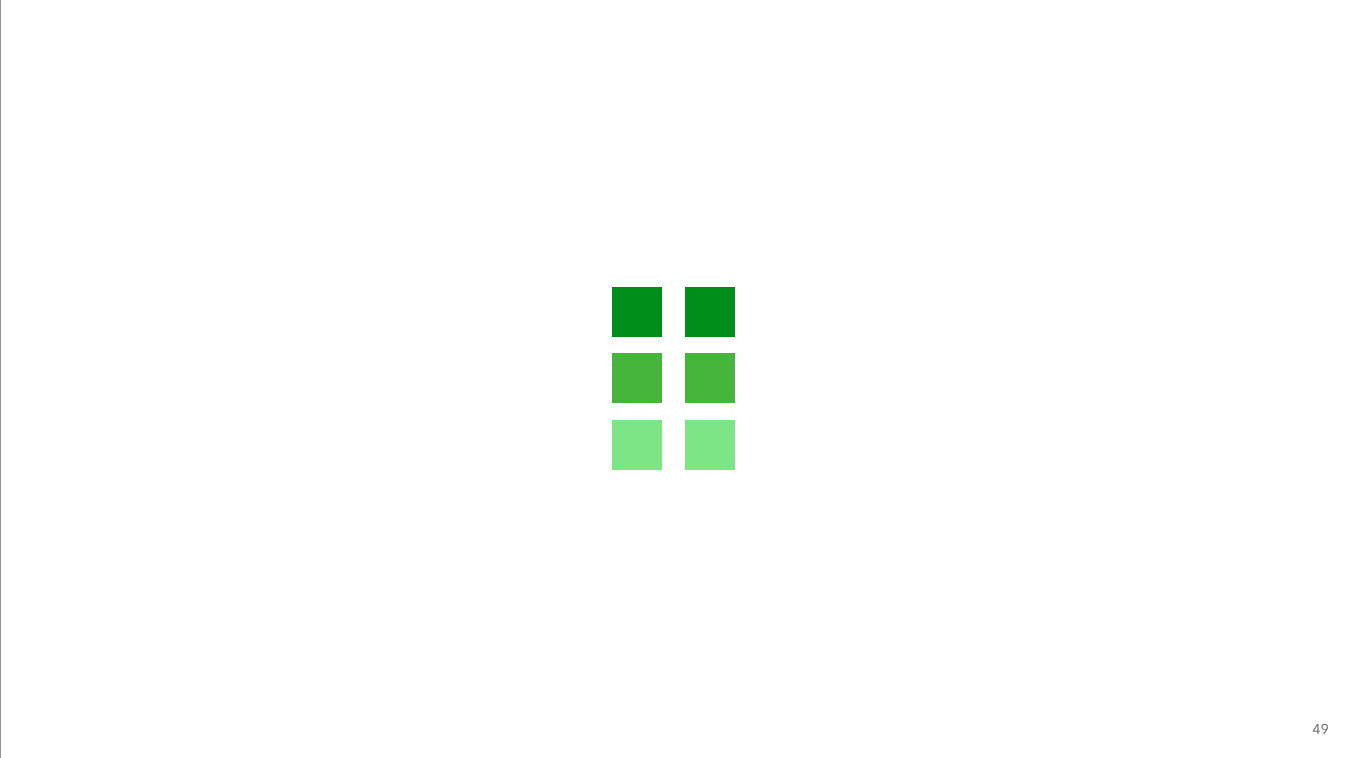
Thus, we also introduce a new map operator.



This map operator knows to map the dot product two dimensions deep, correctly mapping the dot product onto the row-column pairs.

[[■ , ■],
[■ , ■],
[■ , ■]]

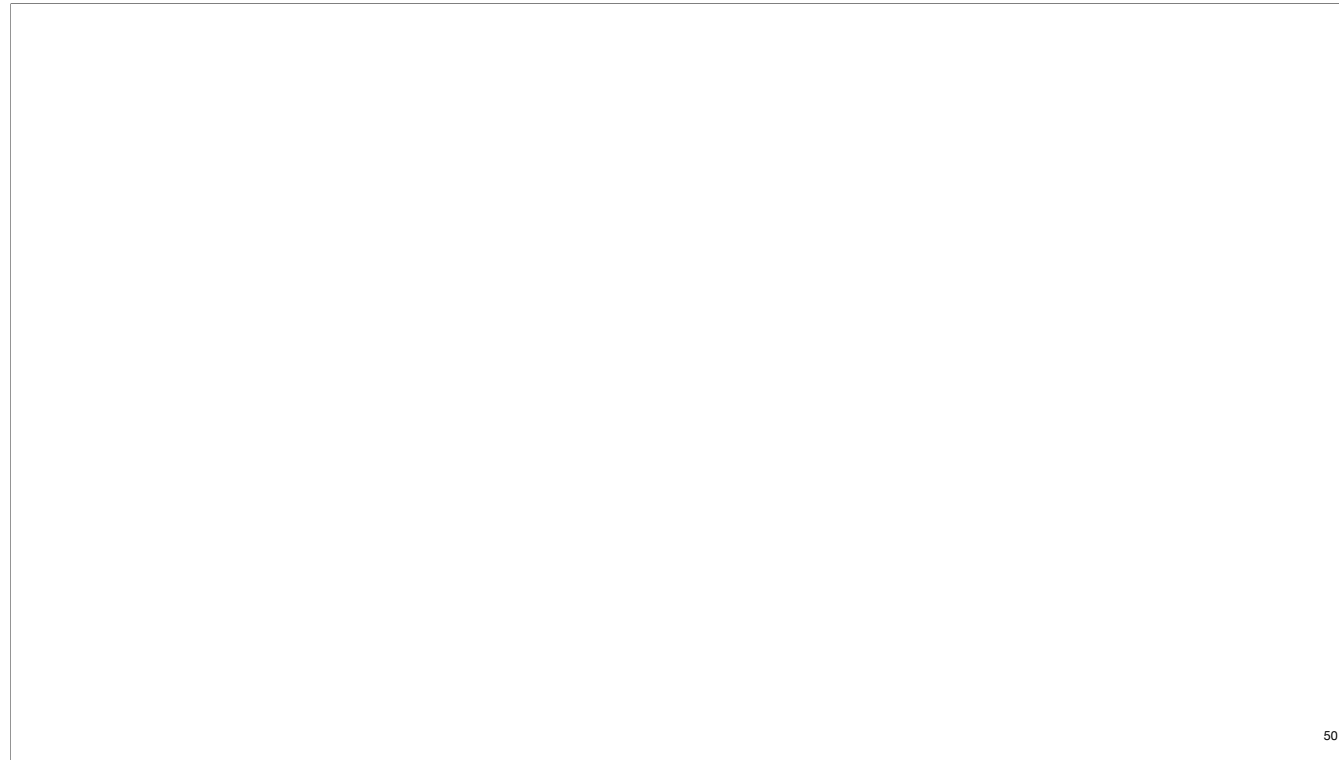
Finally, we get out what we expected:



A two dimensional matrix!



Shape information
is preserved!



50

So what happened here?

Cartesian product 2D and map 2D hard-code which dimensions are iterated over and which dimensions are computed on.

But if the shapes of A and B change, we'll need entirely new operators, for example, map3D.

So can we encode this information—that is, which dimensions are iterated over and which dimensions are computed on—directly in the tensor itself?

Unsurprisingly, the answer is yes!

This is exactly what access patterns do.

\times_{2D} and `map2D` hard-code which dimensions are iterated over and which dimensions are computed on...

\times_{2D} and map2D hard-code which dimensions are **iterated over** and which dimensions are **computed on...**

...but if tensor shapes change, we'll need entirely new operators!

\times_{2D} and `map2D` hard-code which dimensions are **iterated over** and which dimensions are **computed on...**

...but if tensor shapes change, we'll need entirely new operators!

Can we encode this in the tensor itself?

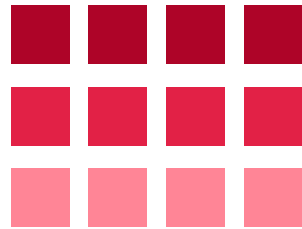
\times_{2D} and `map2D` hard-code which dimensions are **iterated over** and which dimensions are **computed on...**

...but if tensor shapes change, we'll need entirely new operators!

Can we encode this in the tensor itself?

(Yes! This is what Glenside's access patterns do!)

A tensor looks like...



(3, 4)

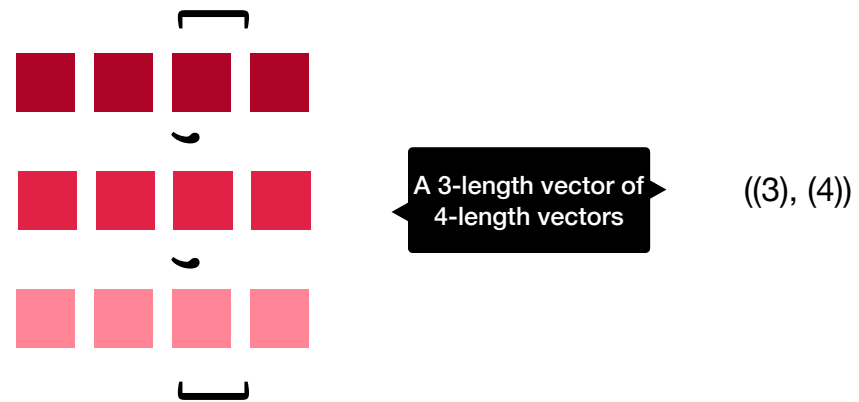
51

Access patterns are the core innovation of Glenside.

Access patterns simply add a bit more information on top of a traditional tensor.

Whereas a tensor is defined by a tuple of integers called a shape—in this case, three comma four...

An access pattern looks like...



52

An access pattern is defined by a pair of tuples, where the first tuple holds the access dimensions, or the dimensions that are iterated over, and the second tuple holds the compute dimensions, or the dimensions which are computed on.

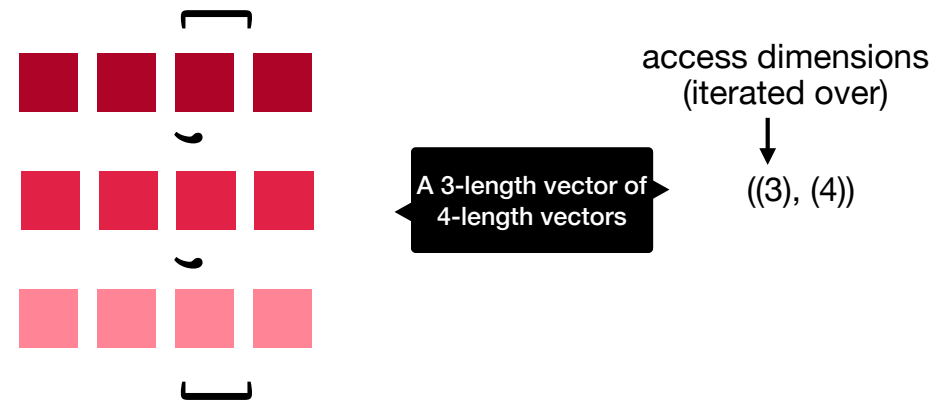
Concatenating the tuple gives the shape of the underlying tensor.

An access pattern simply represents a view over a tensor, conveying how an algorithm computes over the tensor.

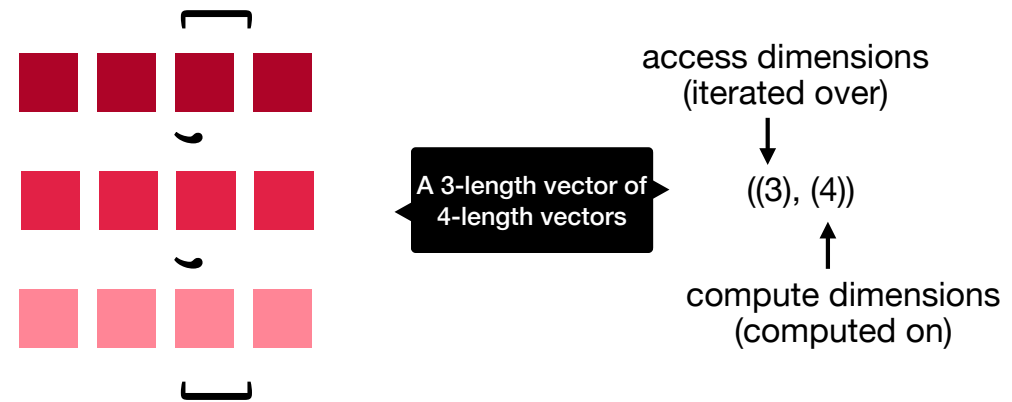
For example, this access pattern is viewing our three comma four shaped tensor as a three-length vector of four length vectors.

But if we shift the dimensions of the access pattern over,

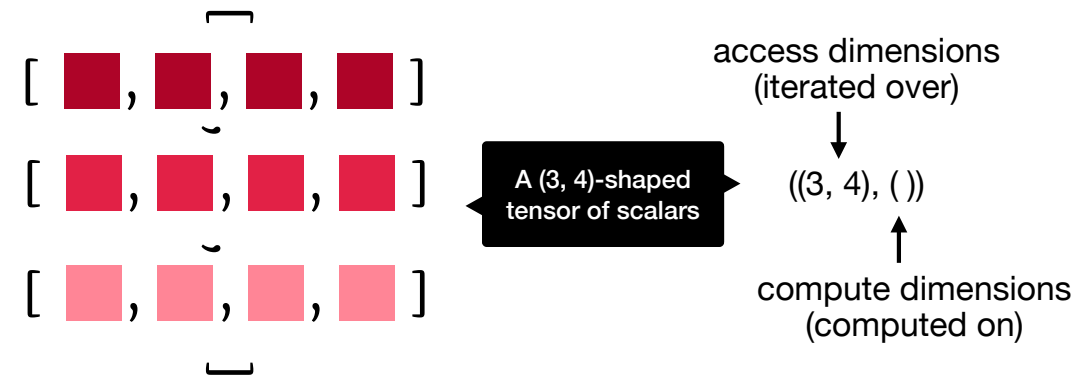
An access pattern looks like...



An access pattern looks like...

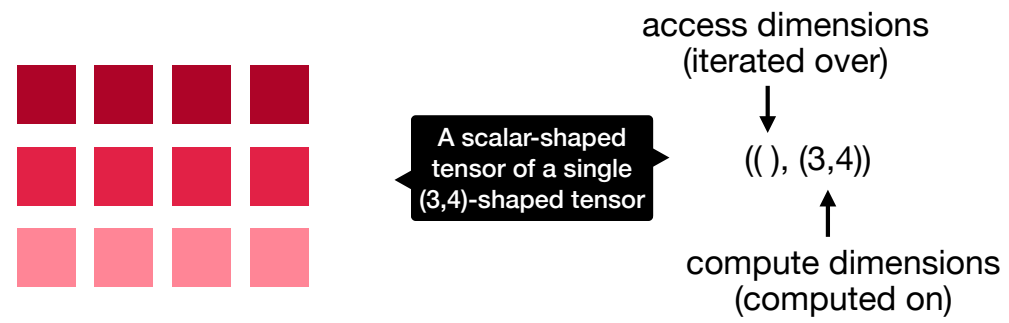


An access pattern looks like...



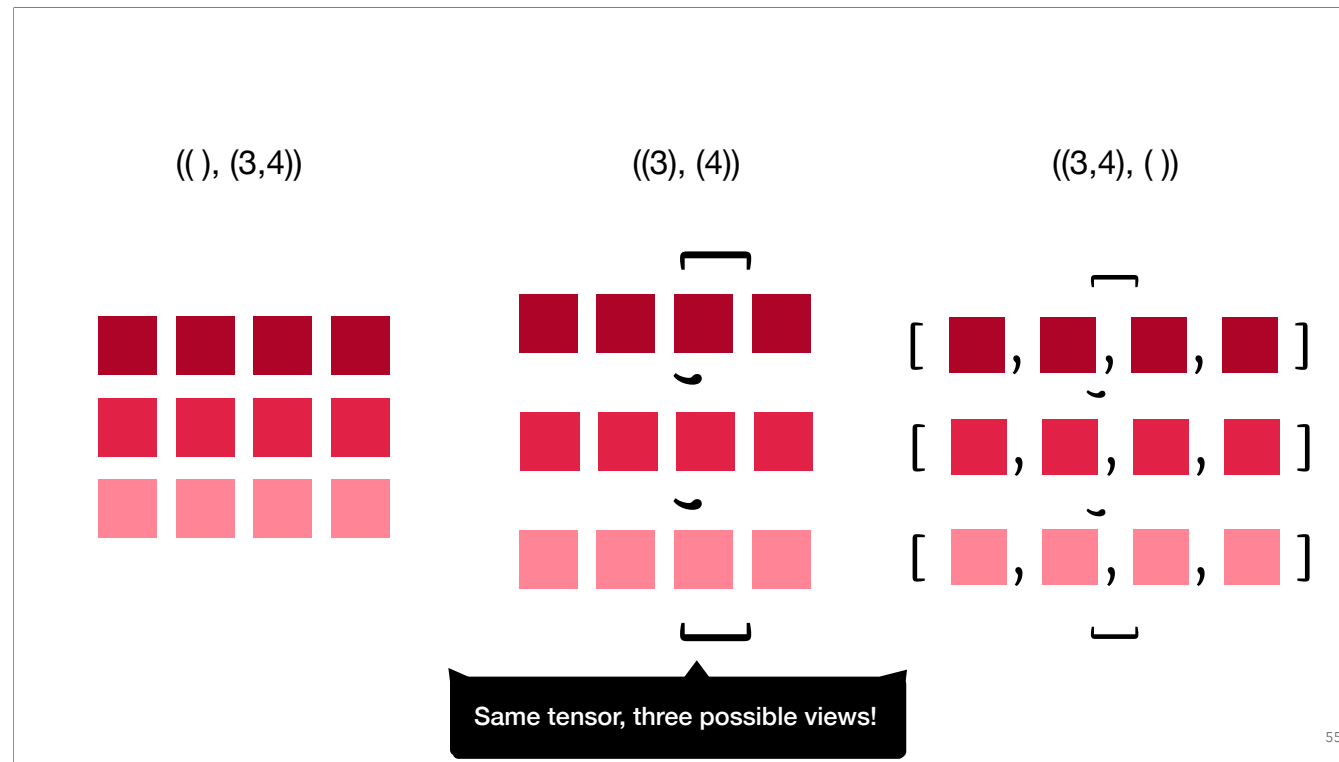
We can view the same tensor as a three-comma-four shaped tensor of scalars.
And if we shift the dimensions in the other direction,

An access pattern looks like...



54

We can view the tensor as a scalar-shaped tensor, in which that quote unquote scalar, i.e. a single element, is a single three-comma-four shaped tensor.



Thus, each of these different access patterns represents a different view of the same underlying tensor, each view being potentially useful depending on the algorithm processing the tensor.

Transformer	Input(s)	Output Shape
access	$((a_0, \dots), (\dots, a_n))$ and non-negative integer i	$((a_0, \dots, a_{i-1}), (a_i, \dots, a_n))$
cartProd	$((a_0, \dots, a_n), (c_0, \dots, c_p))$ and $((b_0, \dots, b_m), (c_0, \dots, c_p))$	$((a_0, \dots, a_n, b_0, \dots, b_m), (2, c_0, \dots, c_p))$
windows	$((a_0, \dots, a_m), (b_0, \dots, b_n)),$ window shape literal $((c_0, \dots, c_p), (d_0, \dots, d_q))$	$((a_0, \dots, a_m, b'_0, \dots, b'_n), (w_0, \dots, w_n)),$ where $b'_i = \lfloor (b_i - c_i) / d_i \rfloor + 1$
slice	$((a_0, \dots, a_m), (b_0, \dots, b_n))$ dimension index d	$((a_0, \dots, a_m, b_0, \dots, b_n))$ with a_d removed
squeeze	$((a_0, \dots), (\dots, a_n))$, dimension index d ; we assume $a_d = 1$	$((a_0, \dots), (\dots, a_n))$ with a_d removed
flatten	$((a_0, \dots, a_m), (b_0, \dots, b_n))$	$((a_0 \cdots a_m), (b_0 \cdots b_n))$
reshape	$((a_0, \dots, a_m), (b_0, \dots, b_n)),$ access pattern shape literal $((c_0, \dots, c_p), (d_0, \dots, d_q))$	$((c_0, \dots, c_p), (d_0, \dots, d_q)),$ if $a_0 \cdots a_m = c_0 \cdots c_p$ and $b_0 \cdots b_n = d_0 \cdots d_q$

Table 1. Glenside’s access pattern transformers.

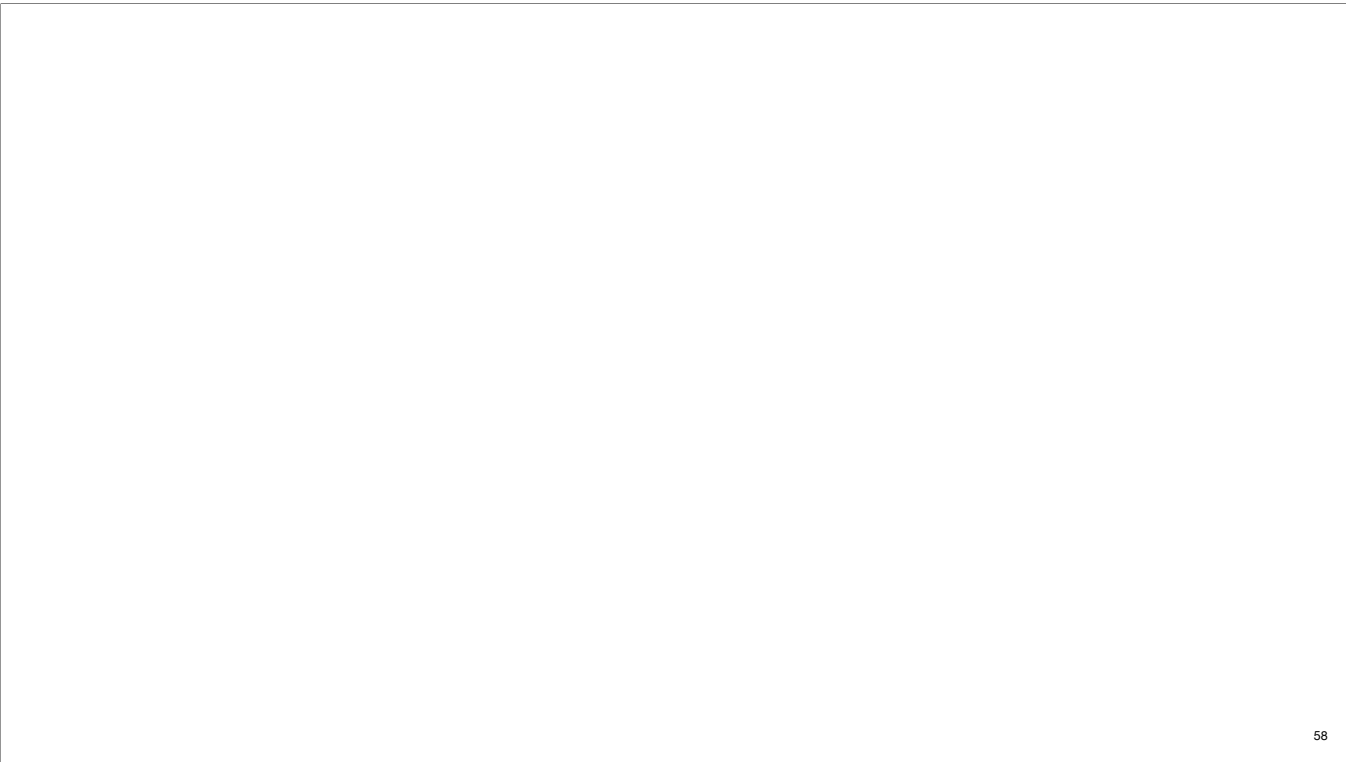
Now, once we’ve defined access patterns, we can redefine a bunch of common tensor and list operators using access pattern semantics—all of these operators collectively form the Glenside IR. There’s a bunch of detail about the operators in the paper, and we’ll see a few of them in our case studies.

<pre> (transpose ; ((N, O, H', W'), ()) (squeeze ; ((N, H', W', O), ())) (compute dotProd ; ((N, 1, H', W', O), ())) (cartProd ; ((N, 1, H', W', O), (2, C, K_h, K_w)) (windows ; ((N, 1, H', W'), (C, K_h, K_w)) (access activations 1) ; ((N), (C, H, W)) (shape C Kh Kw) (shape 1 Sh Sw)) (access weights 1))) ; ((O), (C, K_h, K_w)) 1) (list 0 3 1 2)) </pre> <p style="text-align: center;">(a) 2D convolution.</p>	<pre> (compute dotProd ; ((M, O), ())) (cartProd ; ((M, O), (2, N)) (access activations 1) ; ((M), (N)) (transpose ; ((O), (N)) (access weights 1) ; ((N), (O)) (list 1 0))) </pre> <p style="text-align: center;">(b) Matrix multiplication.</p> <pre> (compute reduceMax ; ((N, C, H', W'), ())) (windows ; ((N, C, H', W'), (K_h, K_w)) (access activations 2) ; ((N, C), (H, W)) (shape Kh Kw) (shape Sh Sw))) </pre> <p style="text-align: center;">(c) Max pooling.</p>
---	---

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.

Glenside can represent common kernels in machine learning.

And then once we define Glenside, we can use it to represent common kernels in machine learning. Not just matrix multiplication, but more complex kernels like 1, 2, and 3d convolution.



58

So we've defined Glenside, and we see that it can be used to represent kernels in deep learning.

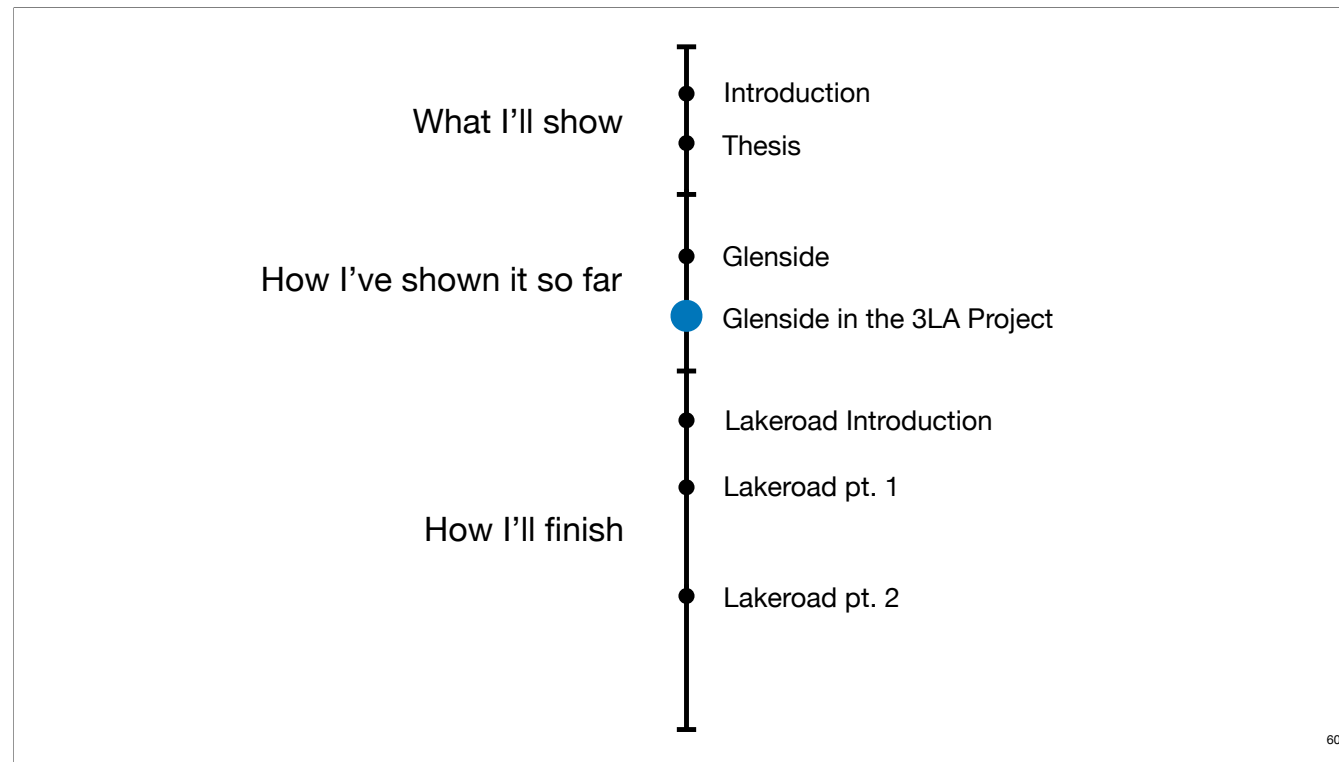
But how is Glenside useful?

But how is Glenside useful?
More importantly, how does it demonstrate my thesis?

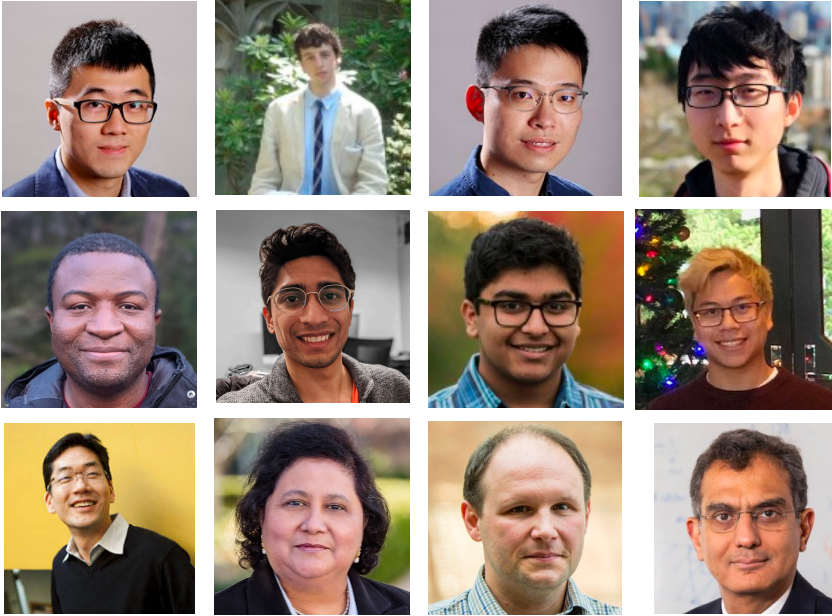
Glenside in the 3LA project


59




To answer that, we're going to move to the next section of my talk, where I talk about Glenside's in a project called 3LA



So just to keep the map in mind, we're moving on to the last part of how I've shown my thesis so far.





Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Thierry Tambe, **Gus Smith**, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, Sharad Malik

"Specialized Accelerators and Compiler Flows: Replacing Accelerator APIs with a Formal Software/Hardware Interface." *arXiv* 2022.

3LA is a collaborative project between us here at UW and our colleagues at Princeton and Harvard. We are currently resubmitting the work to ASPLOS, but for now, we have a paper on Arxiv.



To save time, I'm not going to go into too much detail about the internals of 3LA; instead, I'm mostly going to focus on Glenside's role. However, I wanted to begin by giving the high-level motivation for 3LA.

Simulating, verifying, and compiling workloads on custom accelerators is hard.

Simulating, verifying, and compiling workloads on custom accelerators is hard.

3LA is a toolkit which makes it easier, by compiling workloads to the ILA simulation and verification framework.

Simulating, verifying, and compiling workloads on custom accelerators is hard.

3LA is a toolkit which makes it easier, by compiling workloads to the ILA simulation and verification framework.

Glenside is a key component of 3LA, where it is used to discover mappings of workloads to accelerators.



Huang, B. Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., & Malik, S. (2019). Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification. *ACM Transactions on Design Automation of Electronic Systems*, 24(1), [10]. <https://doi.org/10.1145/3282444>

63

The Instruction Level Abstraction, or ILA, is a specification system developed by our collaborators at Princeton. ILA allows hardware developers to formally specify the behavior of their hardware by defining an ISA-like interface.

This interface is portable across platforms, easy to target from compilers, and provides both verification and simulation abilities out of the box.

Allows hardware developers to specify
ISA-like interface for their design



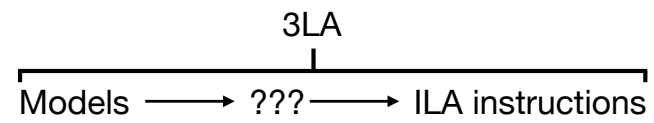
Huang, B. Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., & Malik, S. (2019). Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification. *ACM Transactions on Design Automation of Electronic Systems*, 24(1), [10]. <https://doi.org/10.1145/3282444>

Allows hardware developers to specify
ISA-like interface for their design



Portable, compiler-friendly, and provides verification and simulators out of the box!

Huang, B. Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., & Malik, S. (2019). Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification. *ACM Transactions on Design Automation of Electronic Systems*, 24(1), [10]. <https://doi.org/10.1145/3282444>



64

The core goal of the 3LA project is to compile deep learning workloads down to ILA instructions, so that we can do simulation, verification, and compilation to our custom hardware.

But the question of how to actually map the computations within deep learning models down to ILA instructions

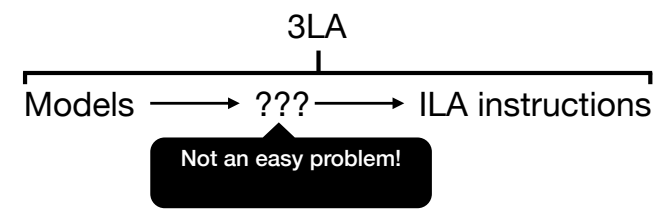
(Build)

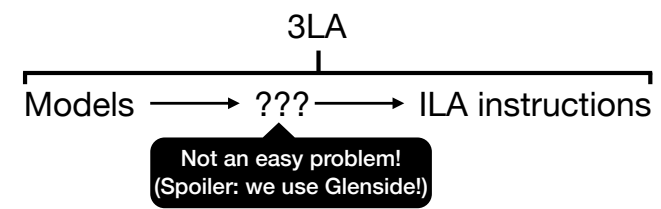
Is not an easy problem.

(Build)

And a spoiler alert here: we're going to use Glenside to solve this problem.

But first, let's talk about what we tried before Glenside.





Can we use TVM's Bring Your Own Codegen?

Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang.
"Bring Your Own Codegen to Deep Learning Compiler." *arXiv* 2021.

65

Before using glenside to map models down to accelerators, we attempted to use TVM's bring your own codegen, or BYOC.

BYOC allows you to match patterns

(Build)

Such as this, within machine learning models.

This pattern matches what's called linear layer: a dense followed by a bias_add.

Can we use TVM's Bring Your Own Codegen?

```
bias_add(dense(*, *), *)
```

Can we use TVM's Bring Your Own Codegen?

```
bias_add(dense(*, *), *)
```

Matches a linear layer: a dense followed by a bias addition.

	EfficientNet	MobileNet V2	ResMLP	ResNet-20	Transformer
VTA	0	1	38	2	66
FlexASR	0	0	0	2	0

Moreau, Thierry, et al. "VTA: an open hardware-software stack for deep learning." *arXiv preprint arXiv:1807.04188* (2018).
T. Tambe *et al.*, "9.8 A 25mm² SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET," *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 158-160, doi: 10.1109/ISSCC42613.2021.9366062.

Using BYOC, we attempt to map five workloads to two different accelerators: VTA and FlexASR, which both accelerate linear layers. As you can see, with the exception of ResMLP and Transformer on VTA, BYOC finds very few mapping opportunities. And, as we’ll see, it’s not because the opportunities aren’t there!


```
%242 = dense(%240, %241, units=10);  
add(%242, %linear_bias)
```

67

Why aren't we finding many matches? In many cases, it is because of simple mismatches. For example, this snippet of code from Mobilenet represents a linear layer, yet it will not match our pattern

(Build)

as the add should be a bias_add.

Many small mismatches like this exist within the workloads. One solution would be to write additional BYOC patterns, to match the subtle variations in the workload.

(Build)

But a more sustainable solution would be to somehow make these rewrites more flexible.

```
%242 = dense(%240, %241, units=10);  
add(%242, %linear_bias)
```

Won't match—this should be a `bias_add`!

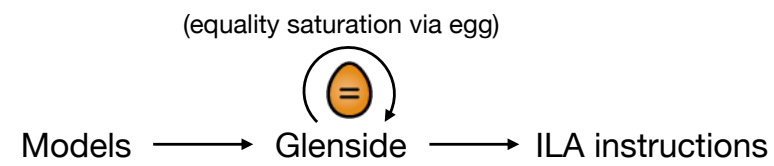
```
%242 = dense(%240, %241, units=10);  
add(%242, %linear_bias)
```

Won't match—this should be a `bias_add`!
If only these rewrites were more flexible...

**Let's use Glenside and equality
saturation!**

68

This is where we will use Glenside.



Flexible matching: using small exploratory rewrites,
we expose many more possible mappings!

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.

"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

Gus Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock.

"Pure tensor program rewriting via access patterns (representation pearl)." MAPS 2021.

69

Using Glenside and equality saturation, we implement what we call flexible matching, where small exploratory rewrites provided by Glenside expose many more possible hardware mappings.

What is equality saturation?

Basic idea:

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization."
In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 264-276. 2009.

Basic idea:
**instead of *destructively* rewriting a program
with a predetermined list of program rewrites,**

Basic idea:

instead of *destructively* rewriting a program
with a predetermined list of program rewrites,
run *all* rewrites simultaneously and repeatedly,

Basic idea:

instead of *destructively* rewriting a program
with a predetermined list of program rewrites,
run *all* rewrites simultaneously and repeatedly,
and keep *all* of the discovered versions of the
program!

Enabled by the equality graph, or egraph, data structure!

Basic idea:

instead of *destructively* rewriting a program
with a predetermined list of program rewrites,
run *all* rewrites simultaneously and repeatedly,
and keep *all* of the discovered versions of the
program!

So let's look at a simple example. Imagine we have a small math language with the following rewrites.

We know that $x \text{ times } 1$ equals 1 .

We know that $x \text{ divided by } x$ also equals 1 .

We know that $x \text{ times two}$ can be implemented as $x \text{ left shifted by } 1$.

And finally, we know that we can re-associate multiplication and division.

$$x * 1 ==> 1$$

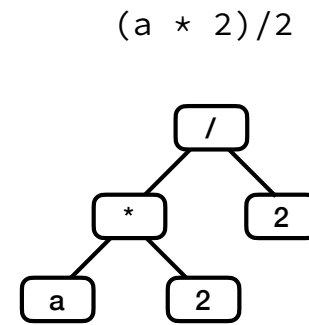
$$x * 1 ==> 1$$

$$x / x ==> 1$$

$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$

$$\begin{aligned}x * 1 &==> 1 \\x / x &==> 1 \\x * 2 &==> x << 1 \\(x * y) / z &==> x * (y / z)\end{aligned}$$

$x * 1 \implies 1$
 $x / x \implies 1$
 $x * 2 \implies x \ll 1$
 $(x * y) / z \implies x * (y / z)$

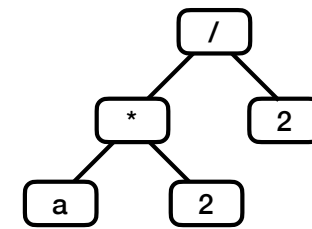


So let's use these rewrites to simplify a times two divided by two down to

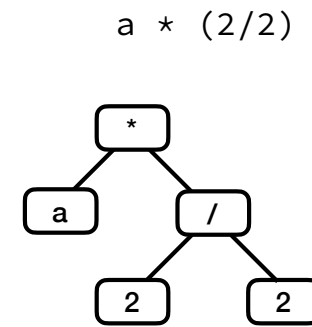
(Build)

a.

$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$

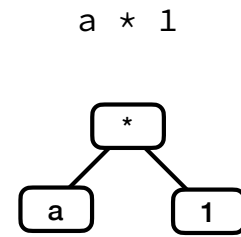


$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



First, we reassociate the multiplication and the division, so that we have two-divided by two.

$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



Then, we simplify two divided by two.

a

a

x * 1 ==> 1

x / x ==> 1

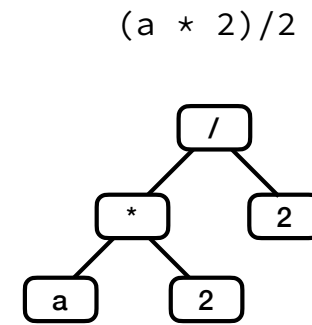
x * 2 ==> x << 1

(x * y) / z ==> x * (y / z)

Finally, we simplify a times 1 to just a.

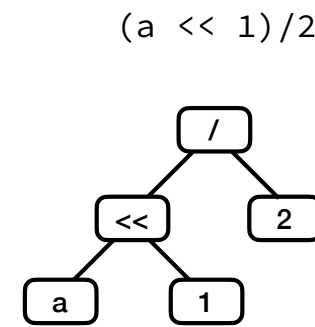
**But what if rewrites ran in a
different order?**

$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



Imagine for example...

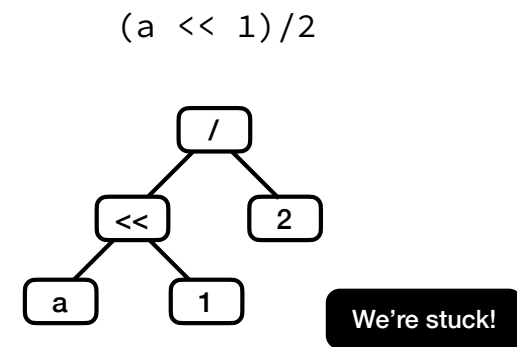
$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



That we ran this rewrite first, and converted a times two to a left shifted by one.

Well at this point, we're stuck! We can't do any more rewrites, and we won't be able to simplify this expression.

$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



**Ordering matters because
rewrites are destructive.**

Ordering matters because rewrites are destructive.

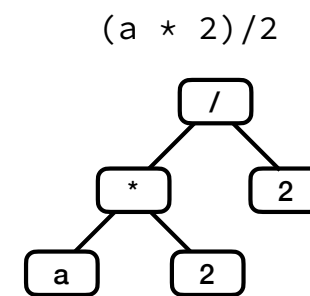
This is called the phase ordering problem!

**So why not keep around all
discovered versions of the program?**

**So why not keep around all
discovered versions of the program?**

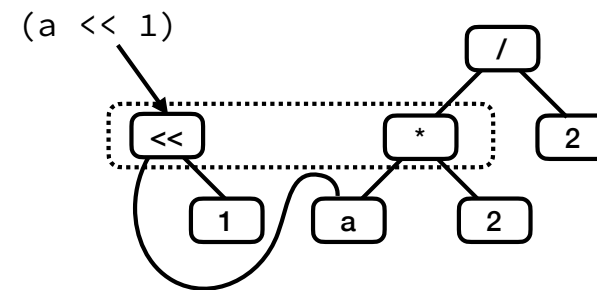
This is what egraphs do!

$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



Using an egraph, when we apply our rewrite to a times 2,

$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



We still get our a left shifted by 1 expression, but

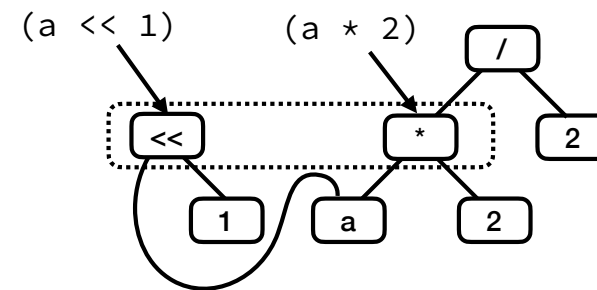
(Build)

We keep around the a times two expression!

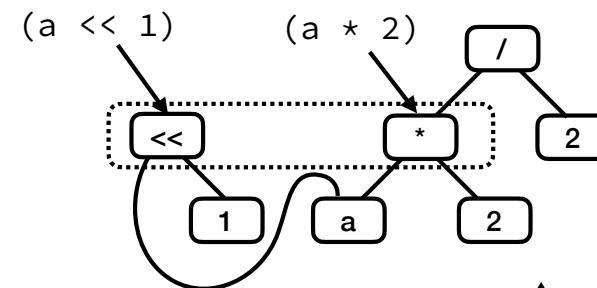
As a result

(Build)

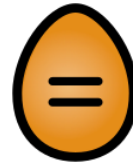
$x * 1 ==> 1$
 $x / x ==> 1$
 $\mathbf{x * 2 ==> x << 1}$
 $(x * y) / z ==> x * (y / z)$



$x * 1 ==> 1$
 $x / x ==> 1$
 $x * 2 ==> x << 1$
 $(x * y) / z ==> x * (y / z)$



We can fire the rewrites in any order —
all discovered programs will be kept!

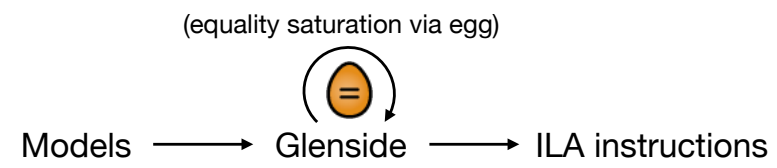


<https://egraphs-good.github.io/>

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

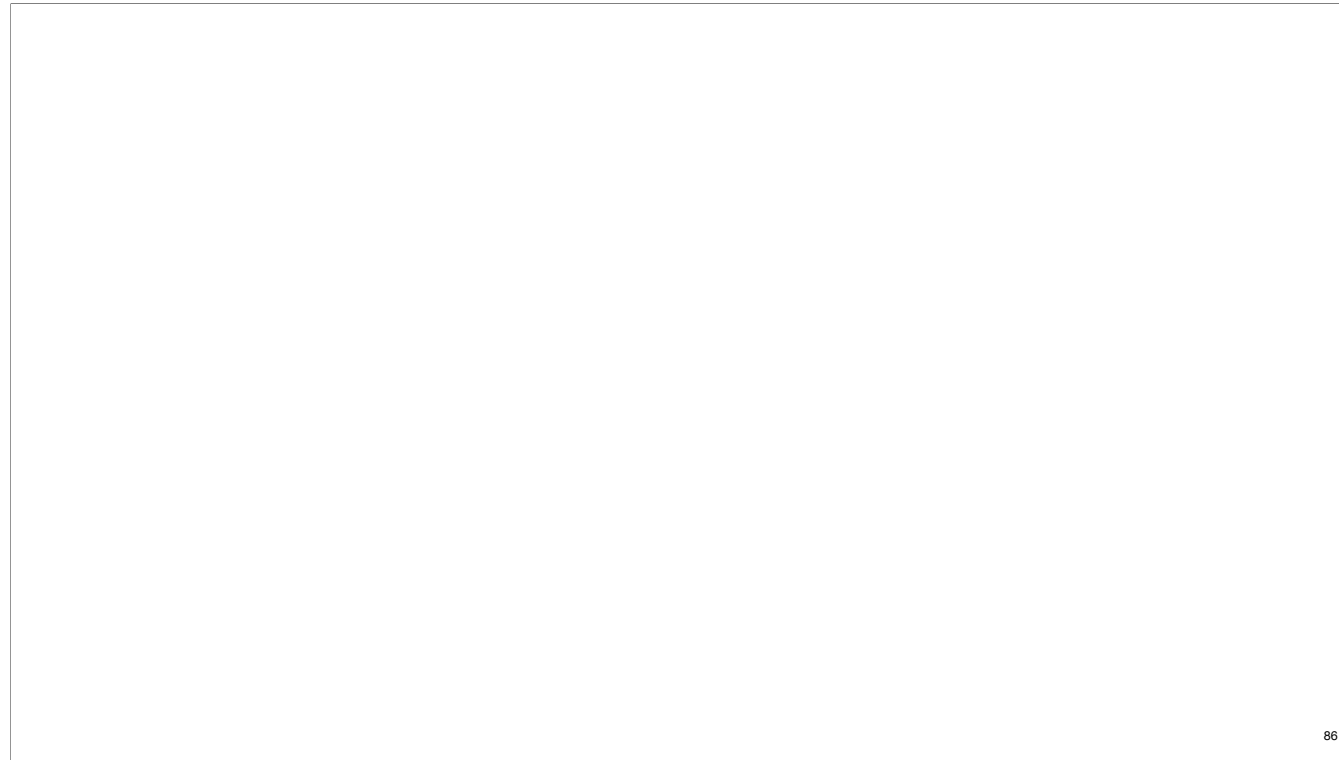
84

This is implemented within a library called egg, developed here at UW by Max Willsey.



Flexible matching: using small exploratory rewrites, we expose many more possible mappings!

So coming back to 3LA. Equality saturation allows us to use small exploratory rewrites to discover many equivalencies within the egraph, which expose more possible mappings down to hardware.



86

So what do these rewrites look like?

Well, we have two types of rewrites.

(Build)

The first type of rewrite capture accelerator semantics as rewrites which rewrite glenside expressions to accelerator calls, and

(Build)

The second type of rewrite, our exploratory rewrites, are just general purpose rewrites over Glenside, provided by Glenside itself!

We capture accelerator semantics as program rewrites...

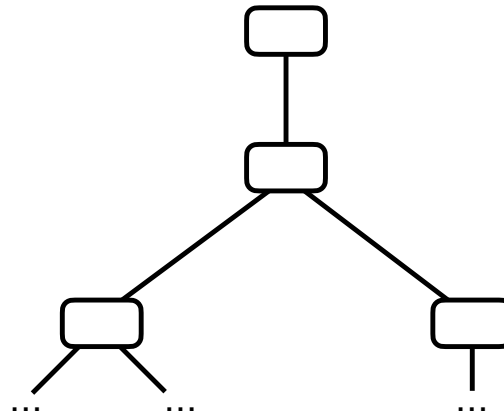
```
(compute dot-product (access-cartesian-product ?x ?w))  
  => (accelerator-call vta-dense ?x ?w)  
(compute reduce-max (access-windows ?a (shape 2) (shape 2)))  
  => (accelerator-call flex-maxpool ...)  
(bias-add (dense ?x ?w) ?bias ?axis)  
  => (accelerator-call flex-linear ?x ?w ?bias)
```

We capture accelerator semantics as program rewrites...

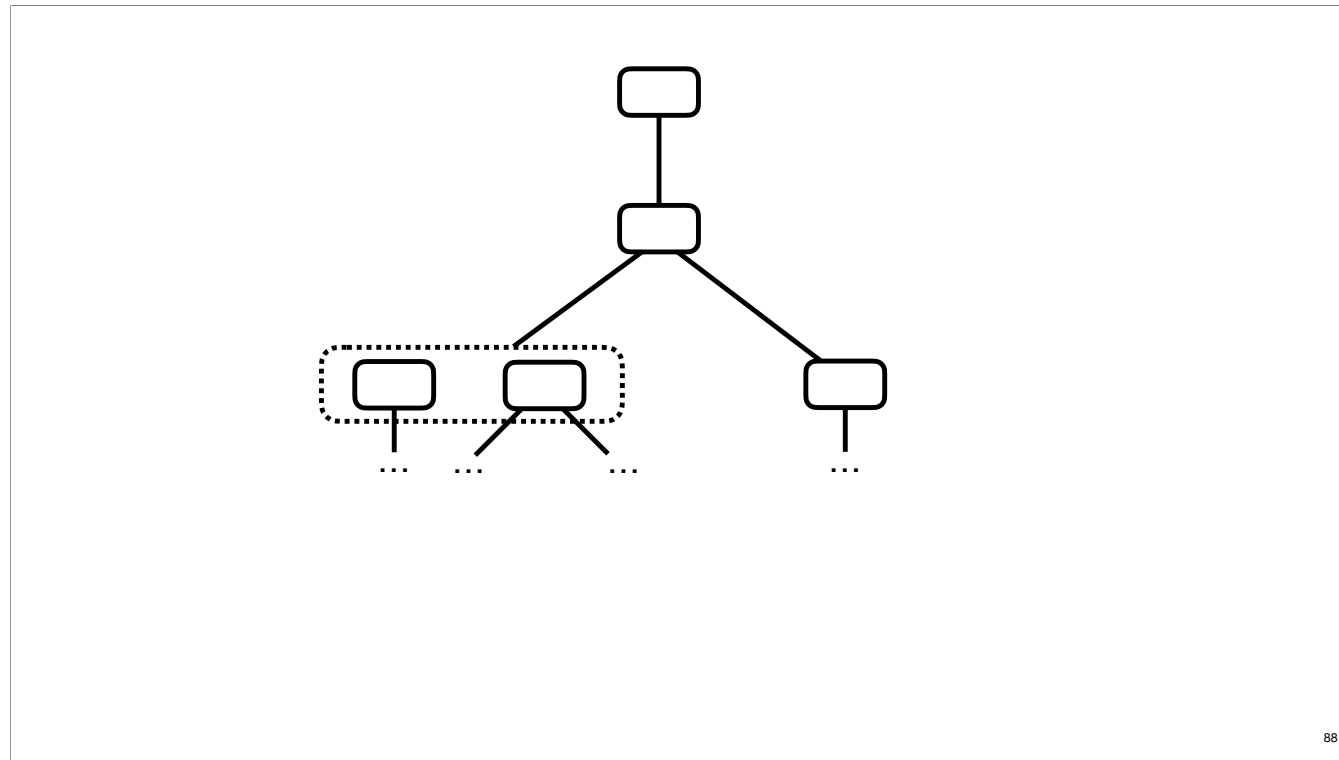
```
(compute dot-product (access-cartesian-product ?x ?w))  
=> (accelerator-call vta-dense ?x ?w)  
  
(compute reduce-max (access-windows ?a (shape 2) (shape 2)))  
=> (accelerator-call flex-maxpool ...)  
  
(bias-add (dense ?x ?w) ?bias ?axis)  
=> (accelerator-call flex-linear ?x ?w ?bias)
```

...and our exploratory rewrites are general-purpose rewrites over Glenside!

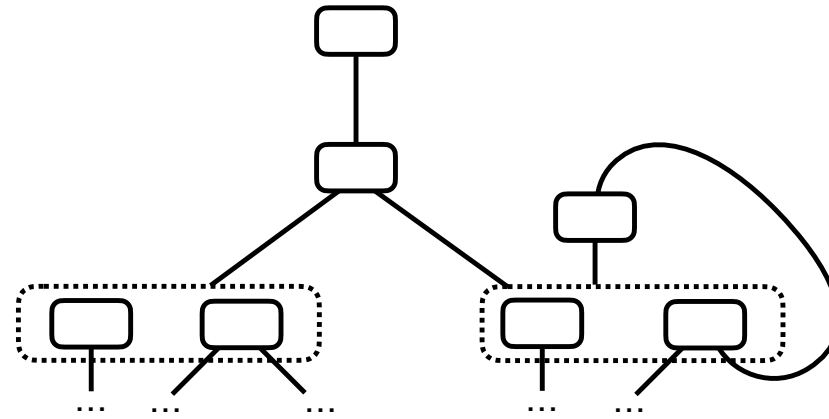
```
?x => (relay-operator-call bias-add ?x  
      (relay-operator-call zeros (shape ...)) 1)  
?a => (reshape (flatten ?a) ?shape)  
(cartProd (reshape ?a0 ?shape0) (reshape ?a1 ?shape1))  
=> (reshape (cartProd ?a0 ?a1) ?newShape)  
(compute dotProd (reshape ?a ?shape))  
=> (reshape (compute dotProd ?a) ?newShape)  
:  
:
```

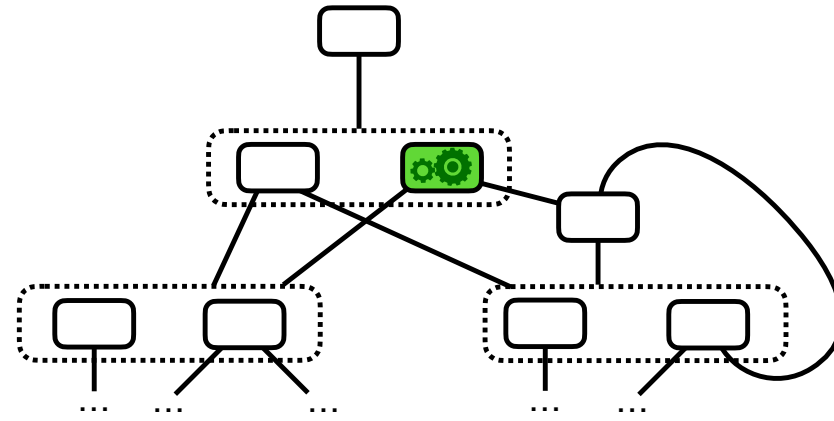


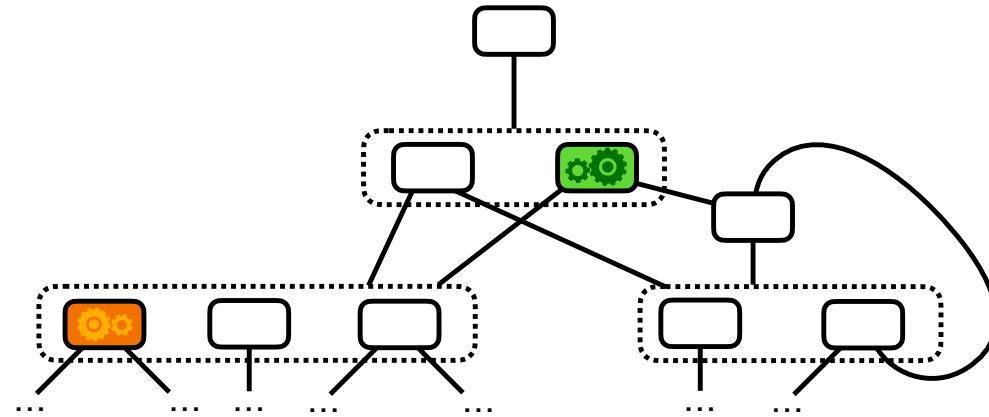
So what does it look like when we actually run these rewrites?



The exploratory rewrites will learn equivalencies about our deep learning model, and eventually,







	EfficientNet	MobileNet V2	ResMLP	ResNet-20	Transformer
VTA	0 → 35	1 → 41	38	2 → 22	66
FlexASR	0 → 35	0 → 41	0 → 38	2 → 22	0 → 66

With flexible matching, we can drastically increase the number of matches for each workload on each accelerator over BYOC.

```
?a → (reshape (flatten ?a) ?shape)
```

```
(cartProd (reshape ?a0 ?shape0) (reshape ?a1 ?shape1))  
→ (reshape (cartProd ?a0 ?a1) ?newShape)
```

```
(compute dotProd (reshape ?a ?shape))  
→ (reshape (compute dotProd ?a) ?newShape)
```

These rewrites *rediscover* the im2col transformation, without explicitly encoding it!

93

One example of a mapping that Glenside finds is the im2col transformation, which allows complex kernels like 2d convolution to be run on matrix multiplication accelerators.

This mapping is uncovered by these three exploratory rewrites, which are general purpose rewrites over Glenside itself. These rewrites do not explicitly encode the im2col transformation, but when used all together, they emergently rediscover the transformation.




So that is the 3LA project.
(Build)
But what does it show about Glenside?

**What does it show about
Glenside?**


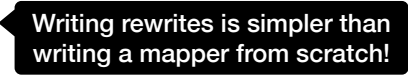
**Automatically generating compiler backends
from explicit, formal hardware models**

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.


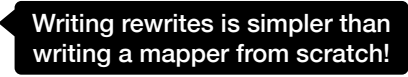
**Automatically generating compiler backends
from explicit, formal hardware models**


- gives rise to emergent optimizations, 
- reduces development time, and
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations, 
- reduces development time, and 
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations, 
- reduces development time, and 
- enables verification.



What I'll show

Introduction

Thesis

How I've shown it so far

Glenside

Glenside in the 3LA Project

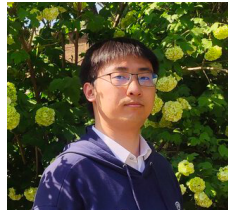
Lakeroad Introduction

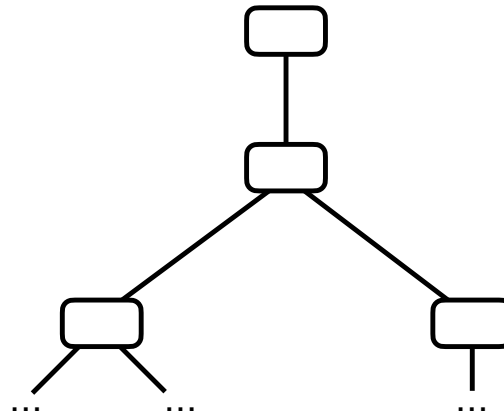
Lakeroad pt. 1

How I'll finish

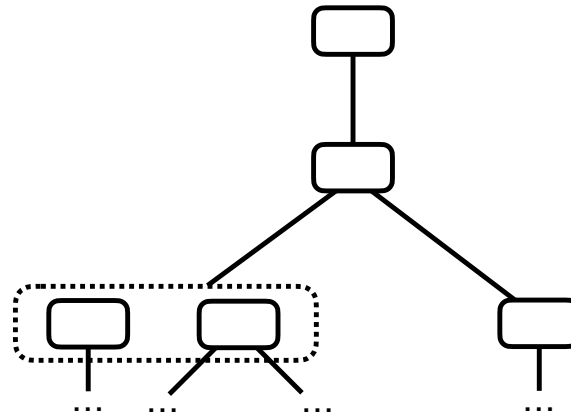
Lakeroad pt. 2

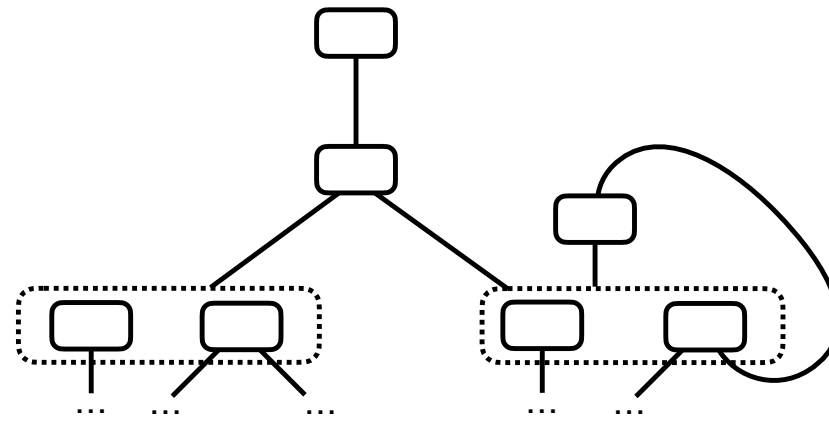
Lakeroad

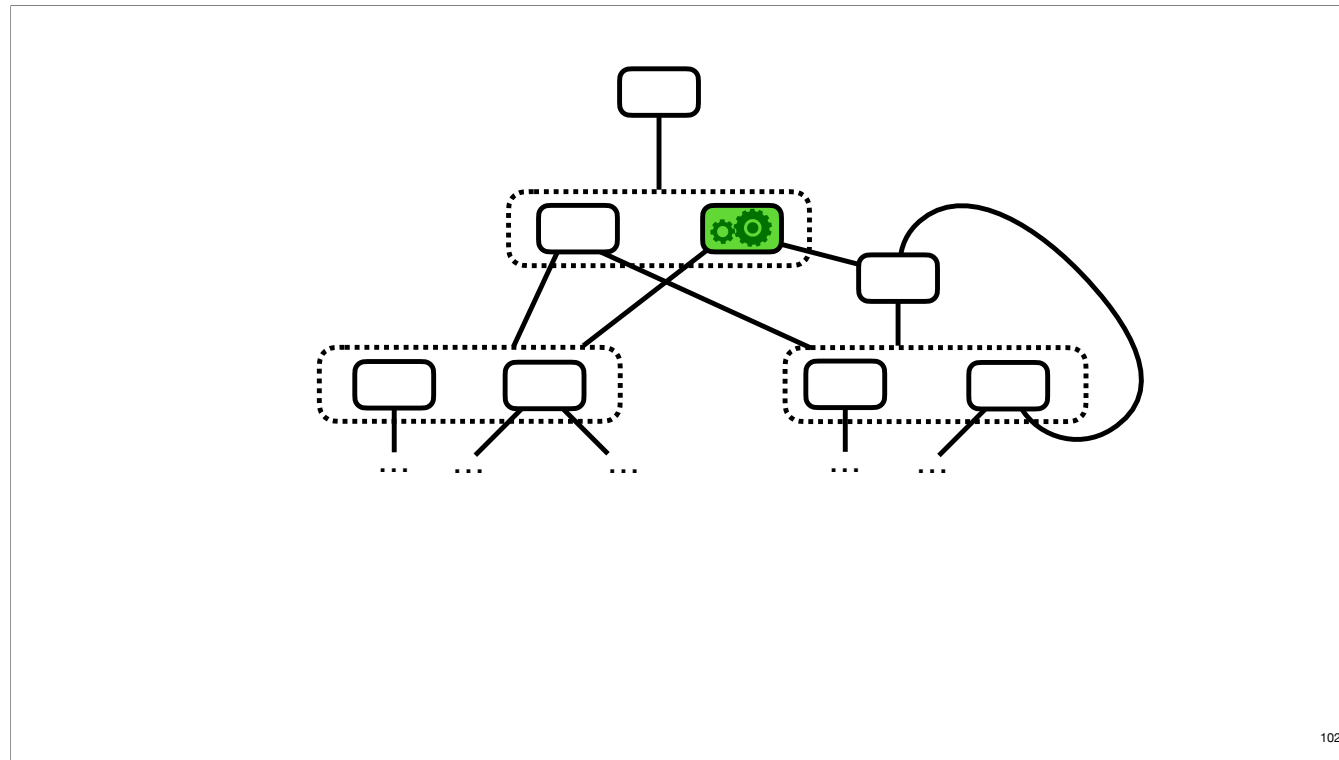




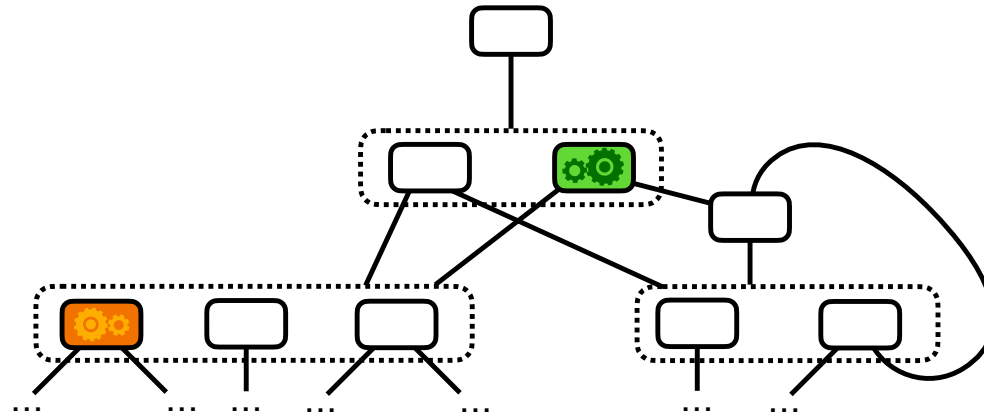
Let's recall what is happening within Glenside as we're running rewrites and searching for hardware mappings. As we run rewrites, we are discovering equivalencies within the egraph.

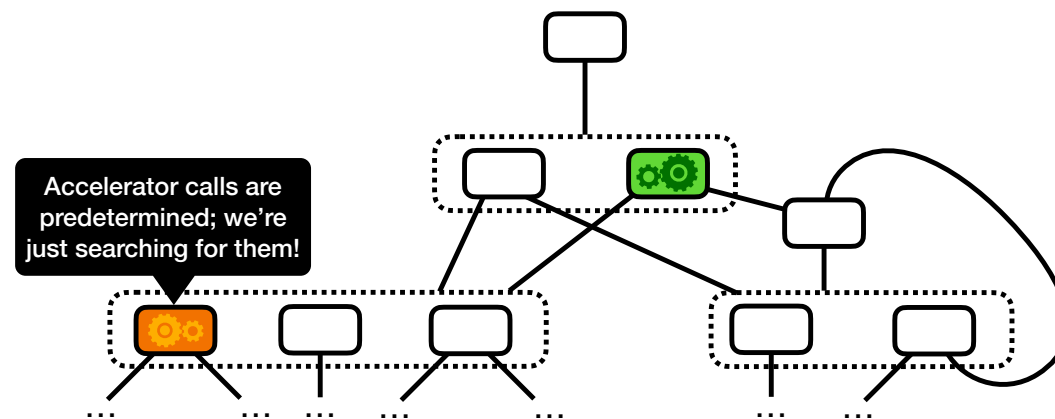


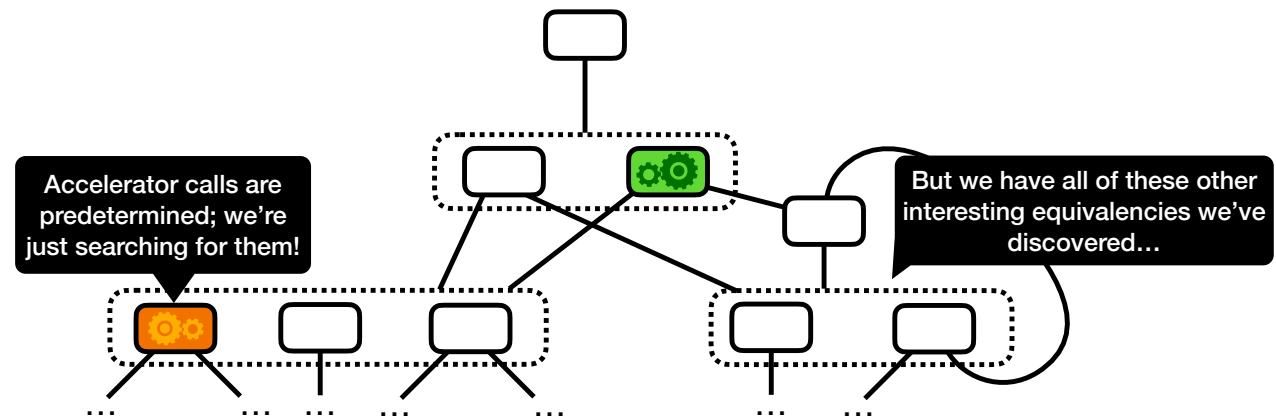


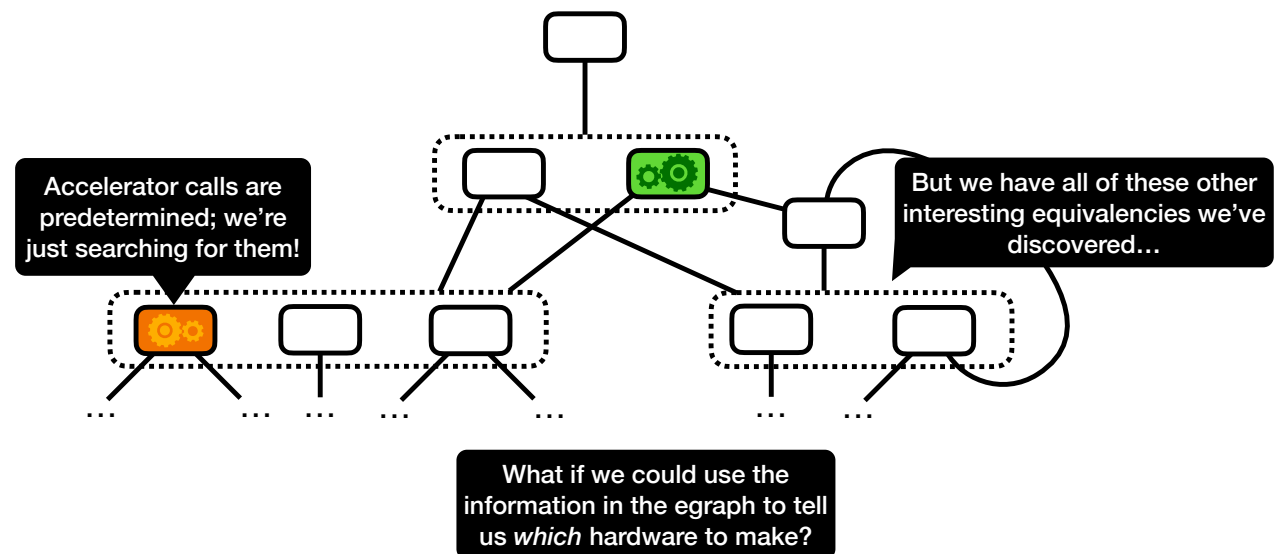


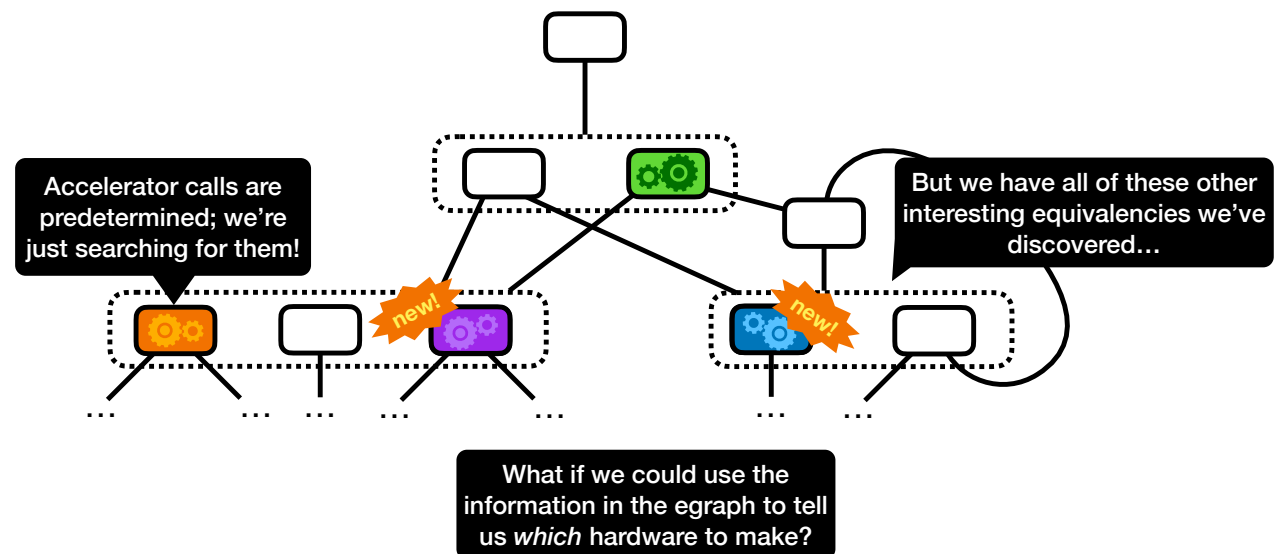
And eventually, we find places to map in calls to our hardware













This is the core idea behind lakeroad.
(Build)

Lakeroad uses similar techniques to Glenside (i.e. equality saturation) to map computation to custom hardware—in this case, FPGAs.

Lakeroad uses similar techniques to Glenside (i.e. equality saturation) to map computation to custom hardware—in this case, FPGAs.

However, Lakeroad additionally uses what it discovers to propose entirely *new* hardware primitives!

What are FPGAs?

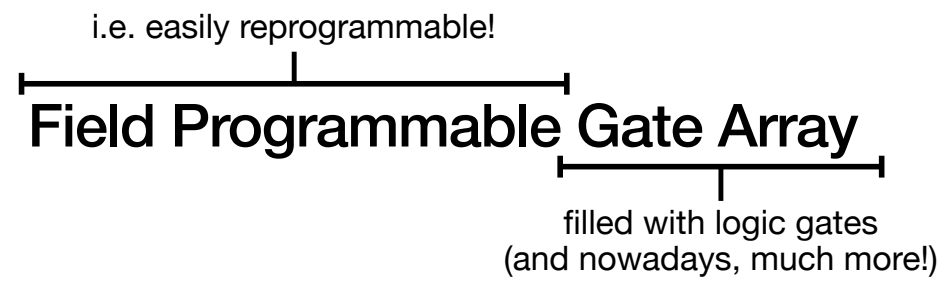
Field Programmable Gate Array

106

Fpga stands for field programmable gate array

i.e. easily reprogrammable!

Field Programmable Gate Array





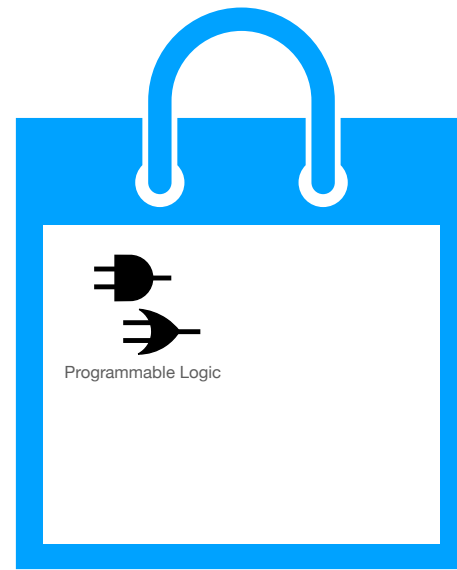


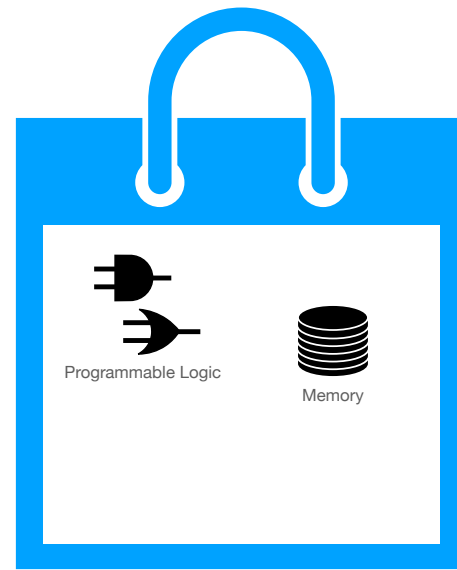
FPGAs are composed of three primary types of devices:

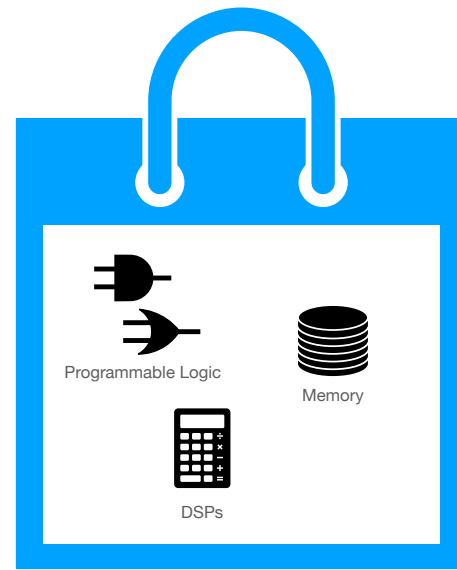
First and most importantly, logic gates, from which FPGAs get their names.

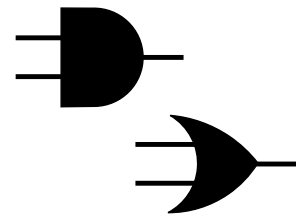
Second, memory. There are memory units scattered throughout the device.

Lastly, digital signal processors or DSPs. These are more complex computational units; often they are small programmable processors! They can implement complex datatypes like floating point or bfloat.









109

For the purposes of this proposal, we will be focusing only on the programmable logic. Don't get me wrong: DSPs and memory are crucial in implementing good hardware designs, but for this proposal, we will test our ideas on programmable logic.

So what is programmable logic? In most cases, it is not and and or gates like I have shown here, but instead, hardware blocks called look-up tables.

(Build)

A look up table is just that—a table.

It takes some number of inputs and outputs some number of outputs, in this case, four inputs and one output.

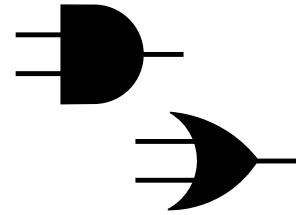
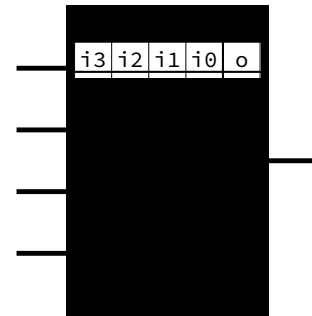
Based on the inputs, it just looks up the output in its internal table.

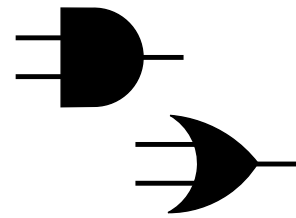
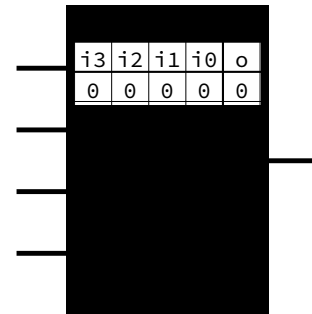
(Build)

In this case, we've programmed this lookup table to implement an and on i0 and i1.

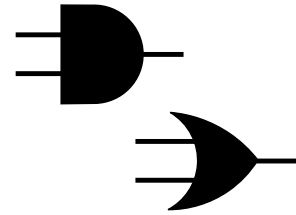
(Build)

Lookup tables come in all shapes and sizes: for example, the lookup tables on the Xilinx Ultrascale+ FPGA architecture have six inputs and two outputs.

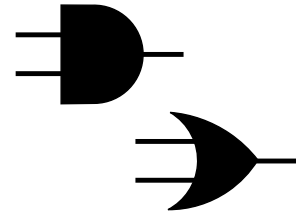




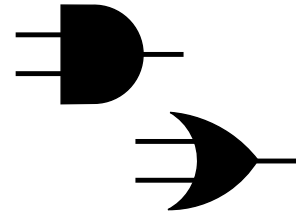
i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0



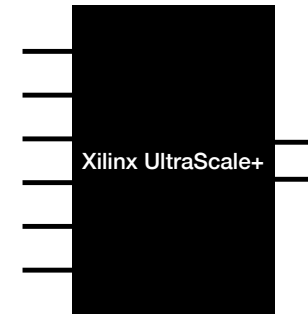
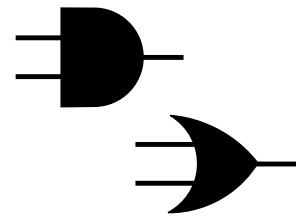
i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0



i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	X
0	1	0	1	X
...



i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	X
0	1	0	1	X
...





Now, let's get into how designs are compiled for FPGAs.

**Current FPGA compilers are slow
and unpredictable.**

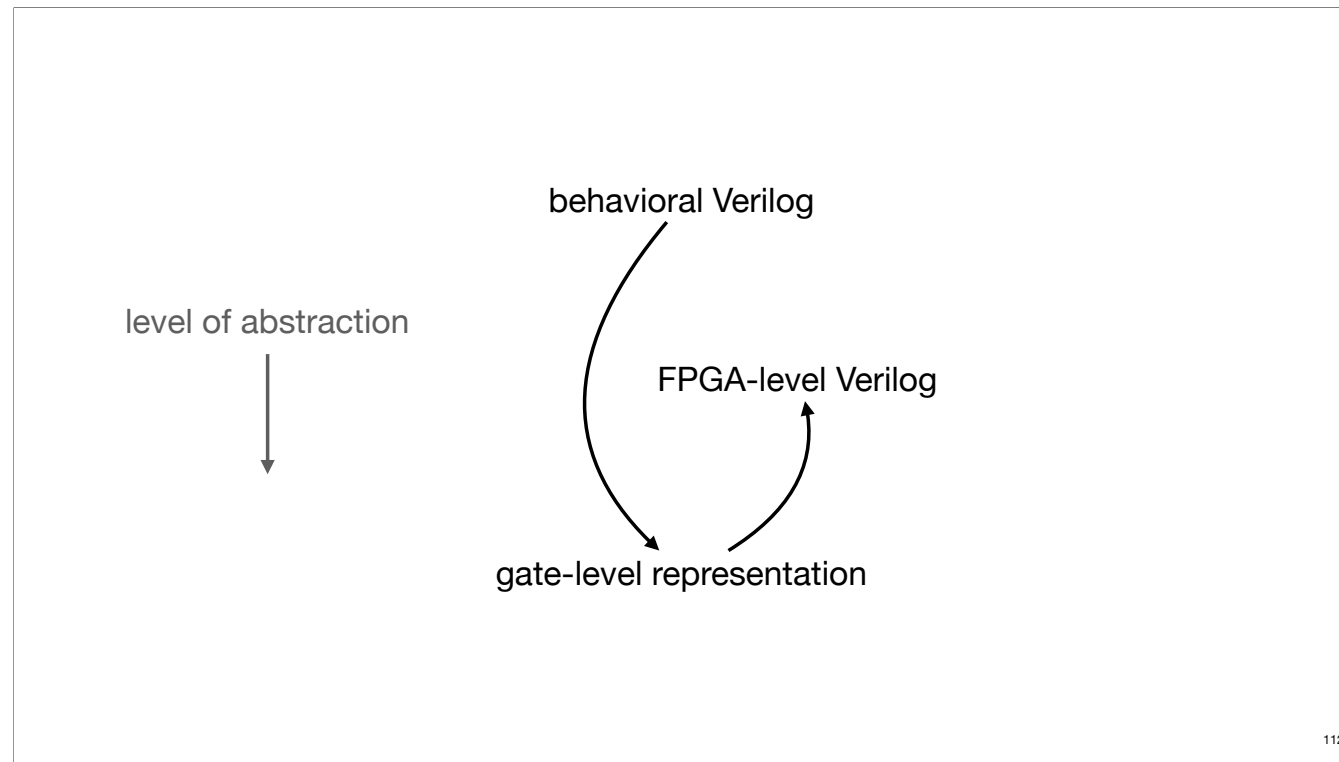
“UltraScale+ devices employ DSP blocks that are rated at 891MHz for the fastest speed grade. Nonetheless, large designs implemented on FPGAs typically achieve system frequencies lower than 400MHz.”

Lavin, Chris, and Alireza Kaviani. "RapidWright: Enabling custom crafted implementations for FPGAs." *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018.

111

In a paper written by Xilinx engineers, the authors state.

So things must be pretty bad if Xilinx engineers themselves are bemoaning the fact that FPGA compilers fail to compile effectively for their hardware!




The reason for the poor performance of FPGA compilers is that, among other reasons, FPGA compilers perform a complex dance, in which they compile the input hardware design from high-level behavioral verilog all the way down to a low level gate representation, and then attempt to raise it back up to the level of abstraction of FPGAs.

Recent works (Reticle!) have attempted a more direct, software-compiler-like approach.

behavioral Verilog

level of abstraction

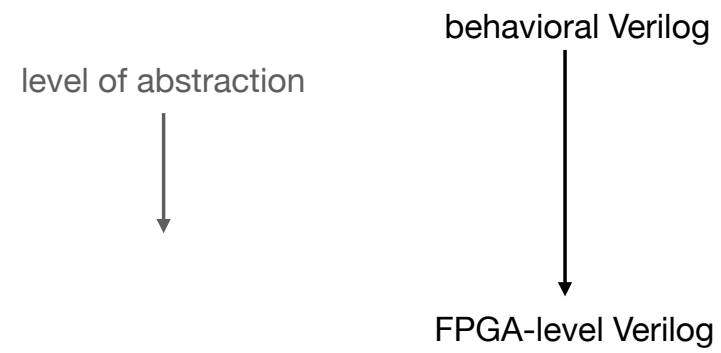


Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. "Reticle: a virtual machine for programming modern FPGAs." PLDI 2021.

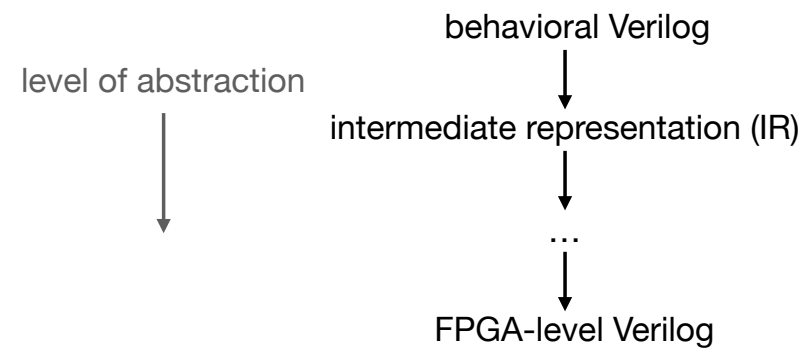
Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth et al. "MLIR as Hardware Compiler Infrastructure." WOSET 2021.

Callahan, Timothy J., Philip Chong, Andre DeHon, and John Wawrzynek. "Fast module mapping and placement for datapaths in FPGAs." FPGA 1998.

114



Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. "Reticle: a virtual machine for programming modern FPGAs." PLDI 2021.
Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth et al.
"MLIR as Hardware Compiler Infrastructure." WOSSET 2021.
Callahan, Timothy J., Philip Chong, Andre DeHon, and John Wawrzynek. "Fast module mapping and placement for datapaths in FPGAs." FPGA 1998.



Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. "Reticle: a virtual machine for programming modern FPGAs." PLDI 2021.
Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth et al.
"MLIR as Hardware Compiler Infrastructure." WOSET 2021.
Callahan, Timothy J., Philip Chong, Andre DeHon, and John Wawrzynek. "Fast module mapping and placement for datapaths in FPGAs." FPGA 1998.

**To implement this, we need an FPGA
“ISA”: the lowest-level IR which gets
converted to FPGA-ready Verilog.**



**New FPGA compiler toolchains
specify their ISAs explicitly!**

'comb' Dialect


Types and operations for comb dialect This dialect defines the `comb` dialect, which is intended to be a generic representation of combinational logic outside of a particular use-case.

- [Operation definition](#)
 - [comb.add \(::circt::comb::AddOp\)](#)
 - [comb.and \(::circt::comb::AndOp\)](#)
 - [comb.concat \(::circt::comb::ConcatOp\)](#)
 - [comb.divs \(::circt::comb::DivSQOp\)](#)
 - [comb.divu \(::circt::comb::DivUOp\)](#)
 - [comb.extract \(::circt::comb::ExtractOp\)](#)
 - [comb.icmp \(::circt::comb::ICmpOp\)](#)
 - [comb.mods \(::circt::comb::ModSQOp\)](#)
 - [comb.modu \(::circt::comb::ModUOp\)](#)
 - [comb.mul \(::circt::comb::MulOp\)](#)
 - [comb.mux \(::circt::comb::MuxOp\)](#)
 - [comb.or \(::circt::comb::OrOp\)](#)
 - [comb.parity \(::circt::comb::ParityOp\)](#)
 - [comb.replicate \(::circt::comb::ReplicateOp\)](#)
 - [comb.shl \(::circt::comb::ShlOp\)](#)
 - [comb.shrs \(::circt::comb::ShrSQOp\)](#)
 - [comb.shru \(::circt::comb::ShrUOp\)](#)
 - [comb.sub \(::circt::comb::SubOp\)](#)
 - [comb.xor \(::circt::comb::XorOp\)](#)

5cbc6be6d4 calyx / primitives / core.futil

Go to file





...

 rachitnigam

@reset interface port (#579) ... ✓





Latest commit 154becf on Jul 1, 2021 History

4 contributors



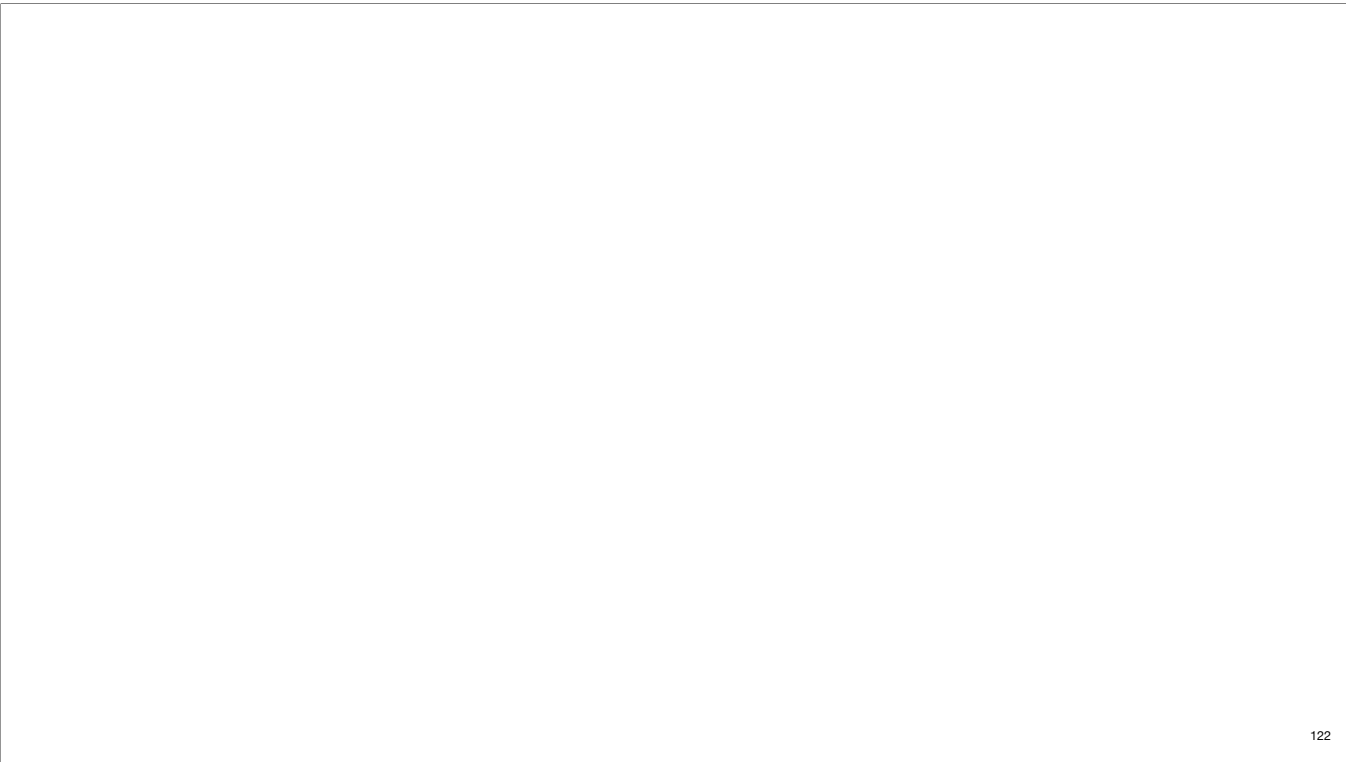
100 lines (93 sloc) 2.87 KB

Raw Blame



```
1 extern "core.sv" {
2   /// Primitives
3   primitive std_const<"share"=1>[WIDTH, VALUE]() -> (out: WIDTH);
4   primitive std_slice<"share"=1>[IN_WIDTH, OUT_WIDTH](in: IN_WIDTH) -> (out: OUT_WIDTH);
5   primitive std_pad<"share"=1>[IN_WIDTH, OUT_WIDTH](in: IN_WIDTH) -> (out: OUT_WIDTH);
6
7   /// Logical operators
8   primitive std_not<"share"=1>[WIDTH](in: WIDTH) -> (out: WIDTH);
9   primitive std_and<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
10  primitive std_or<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
11  primitive std_xor<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
12
13  /// Numerical Operators
14  primitive std_add<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
15  primitive std_sub<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
16  primitive std_gt<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
17  primitive std_lt<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
18  primitive std_eq<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
19  primitive std_neq<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
20  primitive std_ge<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
```

Reticle compiles designs to these ISA instructions, and then those instructions get converted to FPGA-specific Verilog.



So we know ISAs are necessary for these new FPGA compilation toolchains.

But how do we choose the ISA?

But how do we choose the ISA?
And how do we implement it?

But how do we choose the ISA?

And how do we implement it?

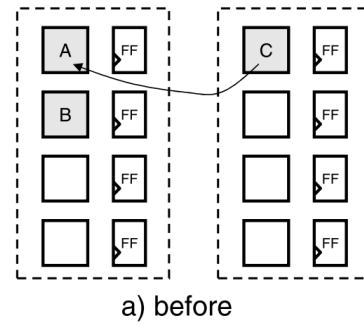
Currently: by hand!



Why is choosing an ISA by hand bad?

**Choosing ISAs by hand may
leave gaps in the ISA.**

A key optimization for FPGAs: *packing* or *fusing* LUTs!



Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. "Packing techniques for virtex-5 FPGAs."
ACM Transactions on Reconfigurable Technology and Systems (TRETs) 2.3 (2009): 1-24.

124

A key optimization for FPGAs is packing or fusing lookup tables.

This diagram shows three lookup tables being used to implement the instructions a, b, and c.

(Build)

If the optimizer discovers that a and c can fit into a single lookup table,

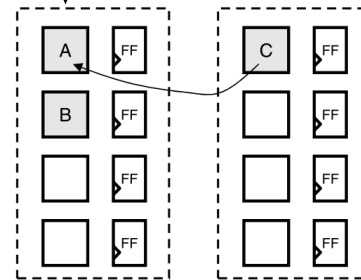
(Build)

It will combine them! This is called packing or fusion.

This is a key optimization in improving FPGA performance and packing larger designs onto the FPGA.

A key optimization for FPGAs: *packing* or *fusing* LUTs!

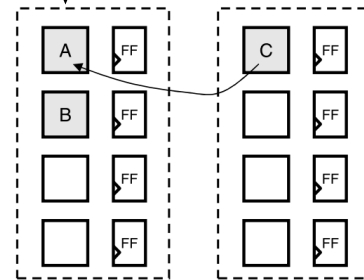
If A and C can fit in a single LUT...



a) before

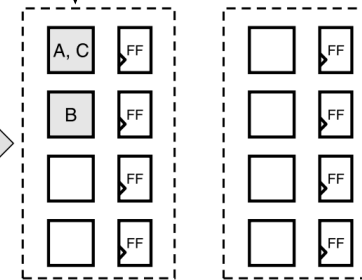
A key optimization for FPGAs: *packing* or *fusing* LUTs!

If A and C can fit in a single LUT...



a) before

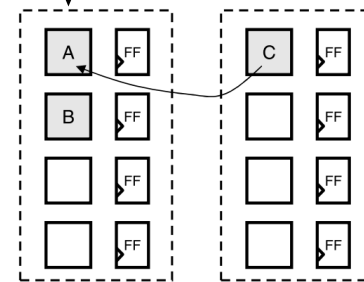
...combine them!



b) after

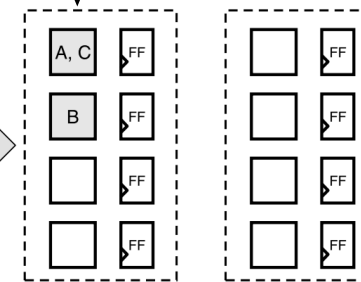
A key optimization for FPGAs: *packing* or *fusing* LUTs!

If A and C can fit in a single LUT...



a) before

...combine them!

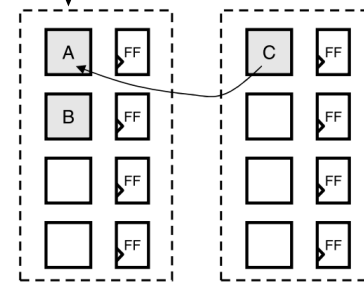


b) after

This requires us to have A,C in our ISA.

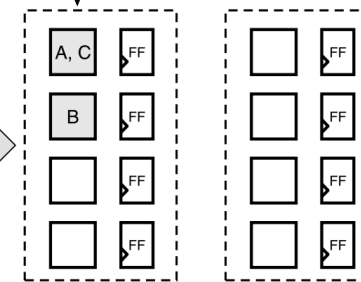
A key optimization for FPGAs: *packing* or *fusing* LUTs!

If A and C can fit in a single LUT...



a) before

...combine them!

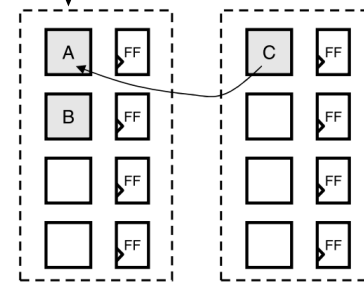


b) after

This requires us to have A,C in our ISA.
Do we also need A,B? Or B,C?

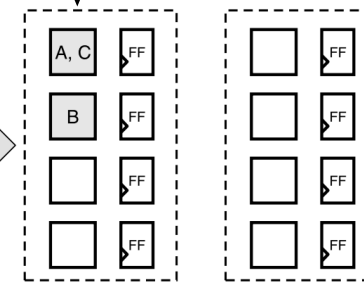
A key optimization for FPGAs: *packing* or *fusing* LUTs!

If A and C can fit in a single LUT...



a) before

...combine them!



b) after

This requires us to have A,C in our ISA.
Do we also need A,B? Or B,C?
Think of all the possible combinations we will have to consider!

**Choosing ISA by hand will miss
many fused instructions.**



Okay, so choosing an ISA by hand is bad.

What about *implementing* ISAs?

**Implementing ISAs by hand is
infeasible for large ISAs—and a
great source of bugs!**


```
8-bit add ISA instruction:  
pat add_i8(a:i8, b:i8) -> (y:i8) {  
    y:i8 = add(a, b) @lut;  
}
```

128

This is an example from Reticle.

In Reticle, Luis has written rewrites which map high level Reticle constructs, such as this add,

(Build)

To low level, FPGA specific implementations of those constructs. This is that same add, implemented using eight lookup tables on a Xilinx FPGA.

(Build)

Luis wrote all of these rewrites by hand for Reticle!

8-bit add ISA instruction:

```
pat add_i8(a:i8, b:i8) -> (y:i8) {  
    y:i8 = add(a, b) @lut;  
}
```



Xilinx 7-series implementation of 8-bit add:

```
imp add_i8[1, 2](a:i8, b:i8) -> (y:i8) {  
    t0:bool = ext[0](a);  
    t1:bool = ext[1](a);  
    t2:bool = ext[2](a);  
    t3:bool = ext[3](a);  
    t4:bool = ext[4](a);  
    t5:bool = ext[5](a);  
    t6:bool = ext[6](a);  
    t7:bool = ext[7](a);  
    t8:bool = ext[0](b);  
    t9:bool = ext[1](b);  
    t10:bool = ext[2](b);  
    t11:bool = ext[3](b);  
    t12:bool = ext[4](b);  
    t13:bool = ext[5](b);  
    t14:bool = ext[6](b);  
    t15:bool = ext[7](b);  
    t16:bool = lut2[6](t0, t8) @a6(??, ??);  
    t17:bool = lut2[6](t1, t9) @b6(??, ??);  
    t18:bool = lut2[6](t2, t10) @c6(??, ??);  
    t19:bool = lut2[6](t3, t11) @d6(??, ??);  
    t20:bool = lut2[6](t4, t12) @e6(??, ??);  
    t21:bool = lut2[6](t5, t13) @f6(??, ??);  
    t22:bool = lut2[6](t6, t14) @g6(??, ??);  
    t23:bool = lut2[6](t7, t15) @h6(??, ??);  
    t24:i8 = cat(t16, t17, t18, t19, t20, t21, t22, t23);  
    y:i8 = carryadd(a, t24) @c8(??, ??);  
}
```

8-bit add ISA instruction:

```
pat add_i8(a:i8, b:i8) -> (y:i8) {  
    y:i8 = add(a, b) @lut;  
}
```

Xilinx 7-series implementation of 8-bit add:

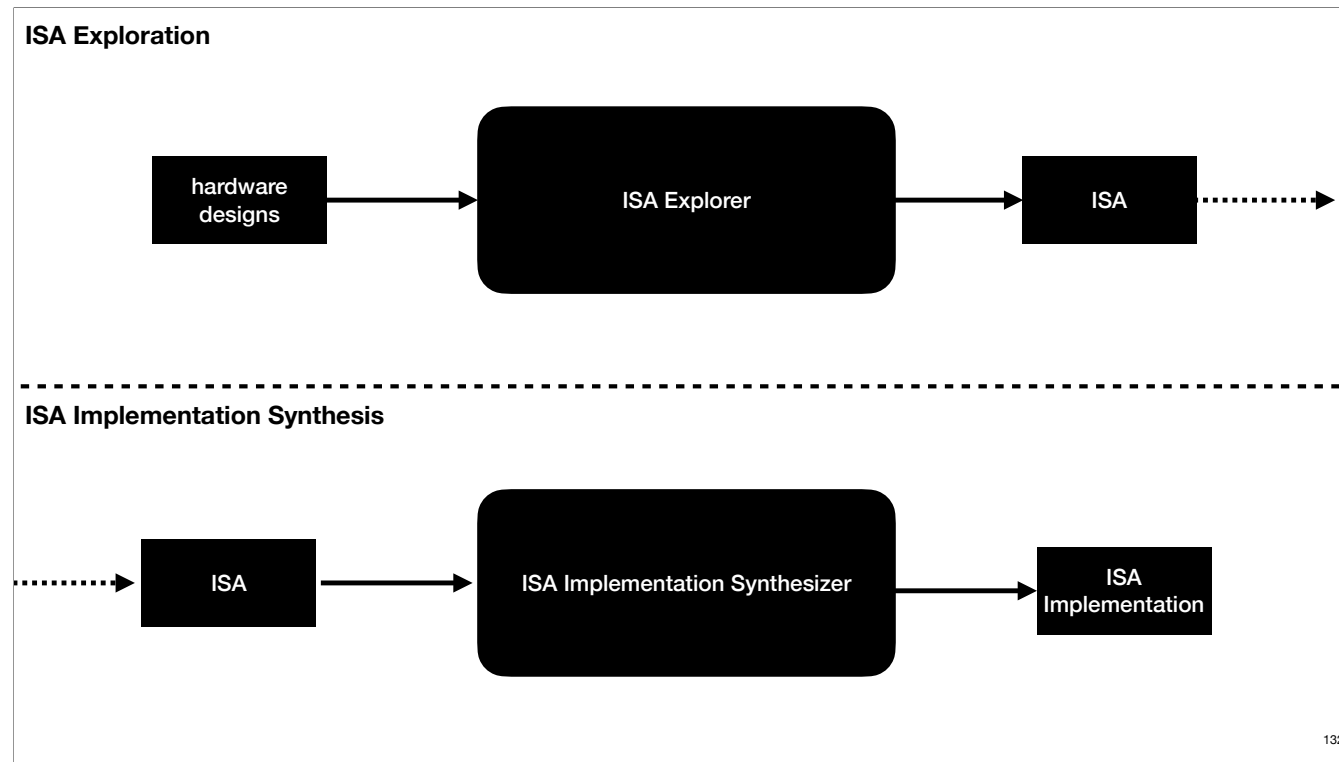
```
imp add_i8[1, 2](a:i8, b:i8) -> (y:i8) {  
    t0:bool = ext[0](a);  
    t1:bool = ext[1](a);  
    t2:bool = ext[2](a);  
    t3:bool = ext[3](a);  
    t4:bool = ext[4](a);  
    t5:bool = ext[5](a);  
    t6:bool = ext[6](a);  
    t7:bool = ext[7](a);  
    t8:bool = ext[0](b);  
    t9:bool = ext[1](b);  
    t10:bool = ext[2](b);  
    t11:bool = ext[3](b);  
    t12:bool = ext[4](b);  
    t13:bool = ext[5](b);  
    t14:bool = ext[6](b);  
    t15:bool = ext[7](b);  
    t16:bool = lut2[6](t0, t8) @a6(??, ??);  
    t17:bool = lut2[6](t1, t9) @b6(??, ??);  
    t18:bool = lut2[6](t2, t10) @c6(??, ??);  
    t19:bool = lut2[6](t3, t11) @d6(??, ??);  
    t20:bool = lut2[6](t4, t12) @e6(??, ??);  
    t21:bool = lut2[6](t5, t13) @f6(??, ??);  
    t22:bool = lut2[6](t6, t14) @g6(??, ??);  
    t23:bool = lut2[6](t7, t15) @h6(??, ??);  
    t24:i8 = cat(t16, t17, t18, t19, t20, t21, t22, t23);  
    y:i8 = carryadd(a, t24) @c8(??, ??);  
}
```

Luis wrote all of these
implementations by hand for Reticle!

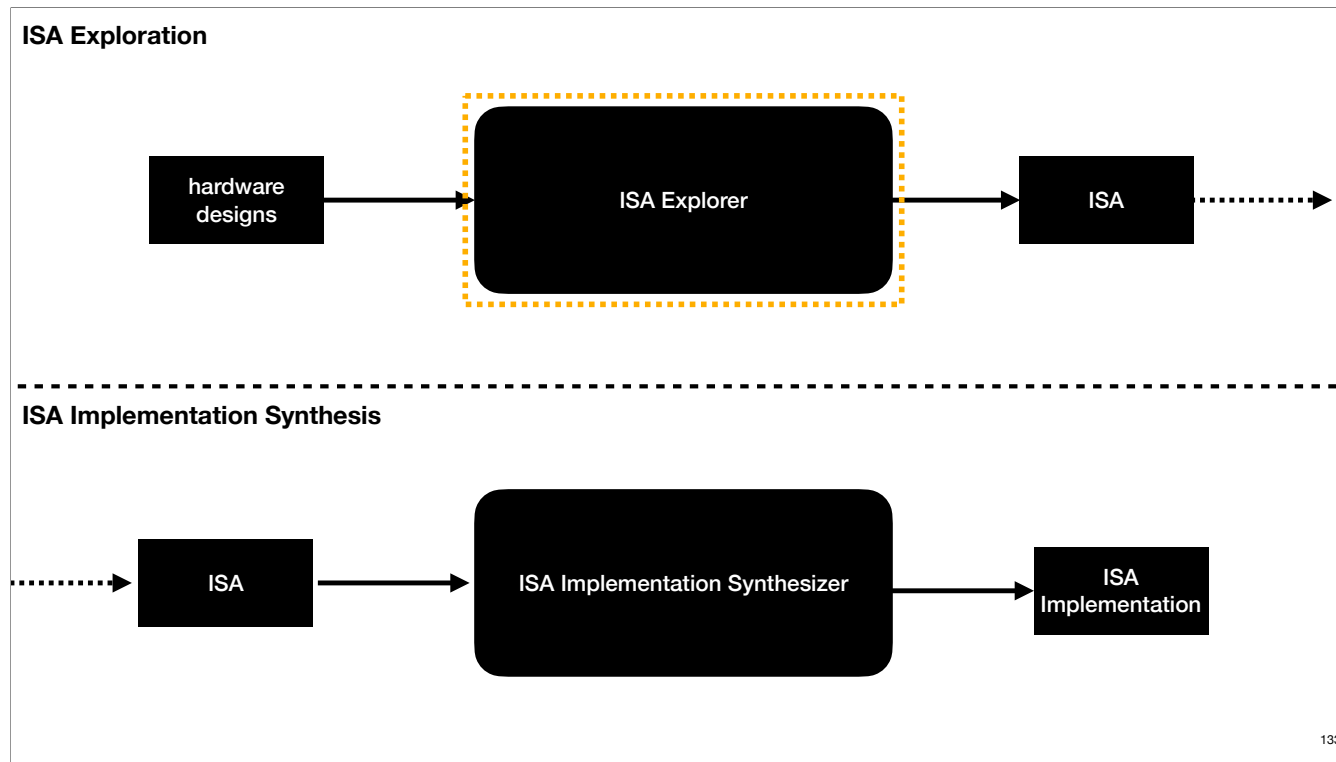
This is slow, especially if we have multiple backends and many fused instructions!

So, can we do it automatically?

**We introduce Lakeroad, a tool for
automatically *defining* and *implementing*
ISAs for FPGAs.**



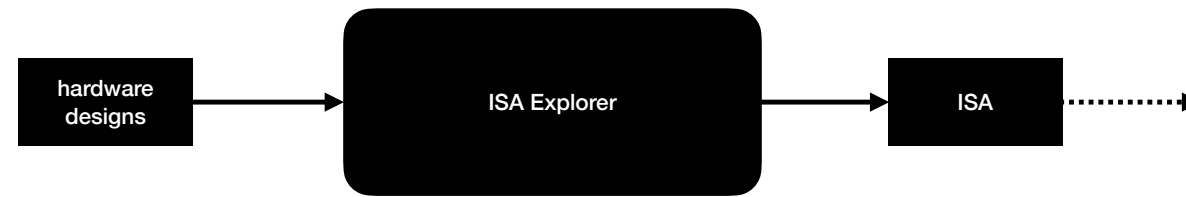
Lakeroad is split into two halves: ISA exploration, which chooses an ISA, and ISA implementation synthesis, which, given an ISA, produces an FPGA-specific implementation.



So let's begin with ISA exploration.

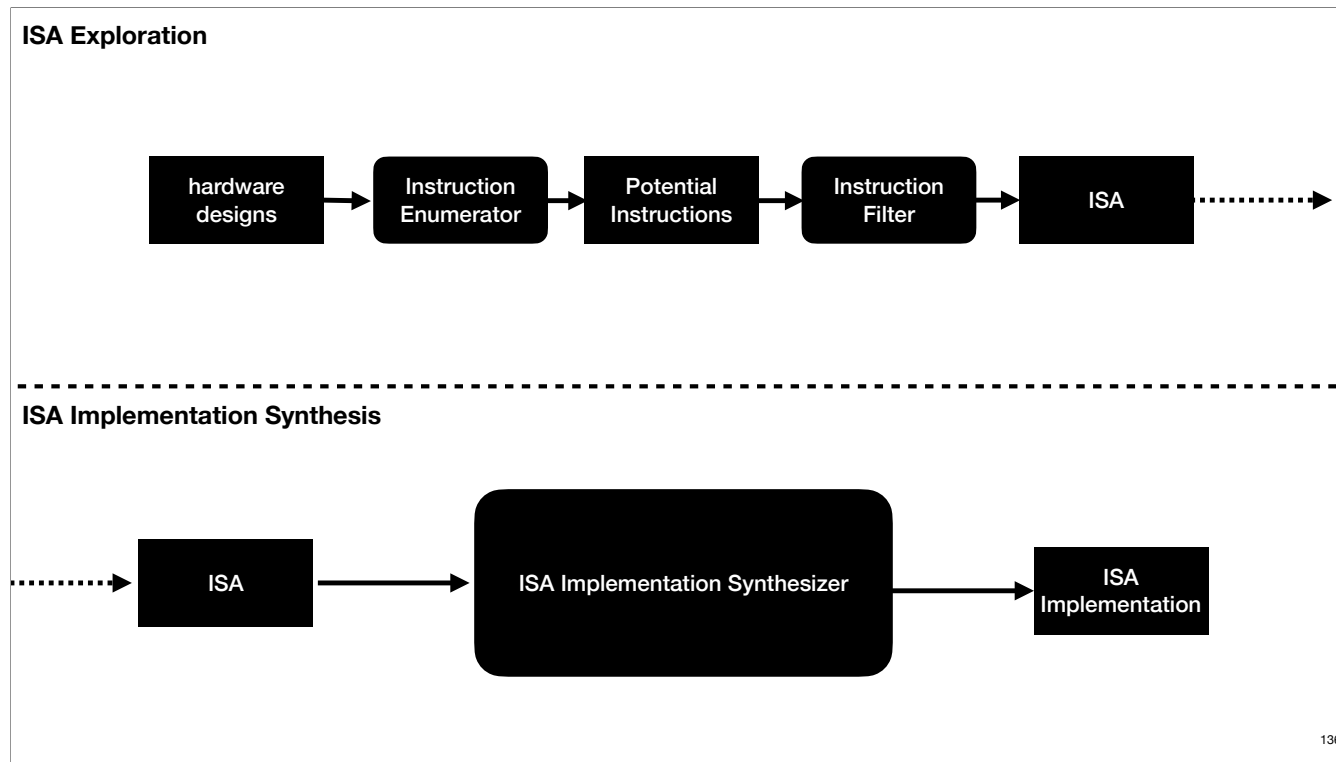
**Core idea of ISA exploration:
define the ISA from instructions
found in real designs.**

ISA Exploration



ISA Implementation Synthesis

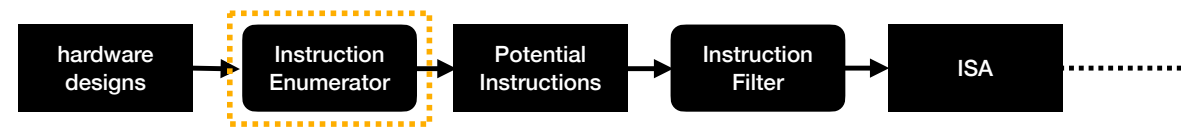




136

Internally, ISA exploration has two stages: instruction enumeration, in which we enumerate all instruction seen in all hardware designs, and instruction filtering, in which we pare down the list of instructions to our final ISA.

ISA Exploration

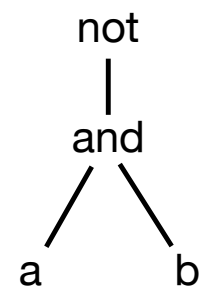


ISA Implementation Synthesis



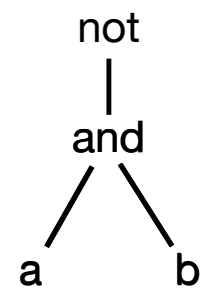
**A simple example: enumerate the
instructions present in
not (a and b)**

not (a and b)

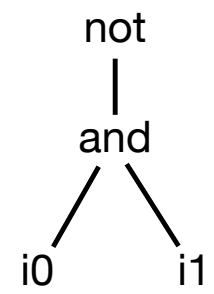


**Naive approach: instructions are
just the subexpressions!**

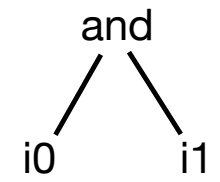
not (a and b)



not (i0 and i1)



i0 and i1



Then the subexpressions would be,

First, the expression itself, not (i0 and i1), where a and b are replaced with generic placeholders i0 and i1,

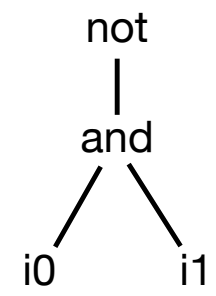
And second, and of i0 and i1.

But we missed one!

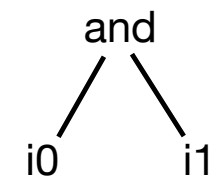
143

But this naive approach misses an instruction!

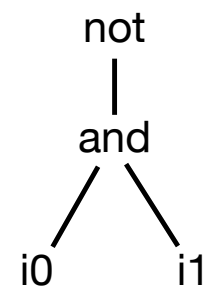
not (i0 and i1)



i0 and i1



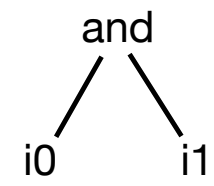
not (i0 and i1)



not i0



i0 and i1



Specifically, the not instruction.

So how do we capture *all* instructions?

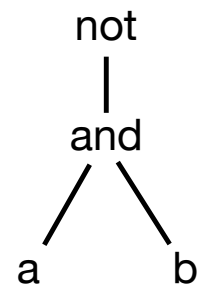
146

So this naive approach doesn't work. So how do we capture all instructions?

Unsurprisingly, I'm going to do it with rewrites, and use egg!

So how do we capture *all* instructions?
With rewrites! 🤖

To convert a node into an instruction,
decide which of its children to convert to arguments.

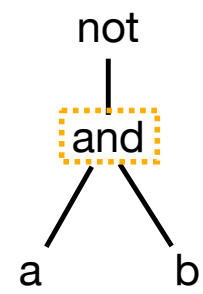


147

First I'll describe what the rewrites do at a high level, and then we'll look at the rewrites running in an egraph.

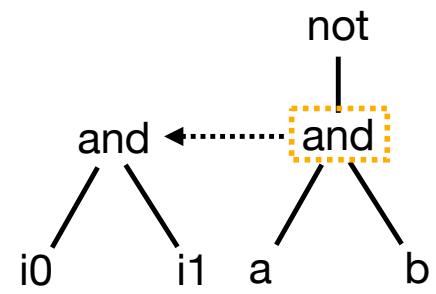
In general, the rewrites say that, to convert a node into an instruction, we have to decide which of its children we will convert to placeholder arguments like `i0` and `i1`, and which children we will keep as their full expressions.

To convert a node into an instruction,
decide which of its children to convert to arguments.



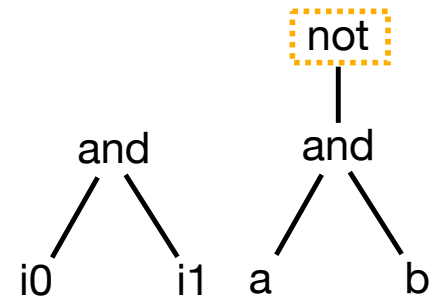
So if we look at the and expression,

To convert a node into an instruction,
decide which of its children to convert to arguments.



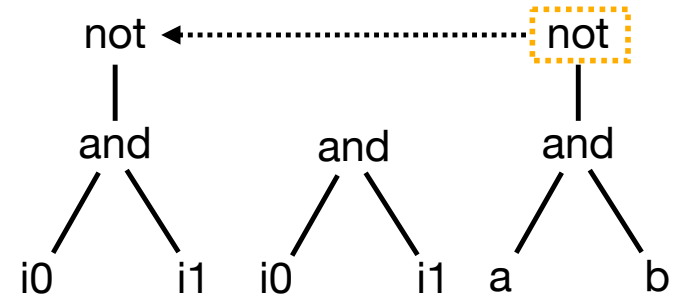
The only option we have is to convert its children to placeholders.

To convert a node into an instruction,
decide which of its children to convert to arguments.

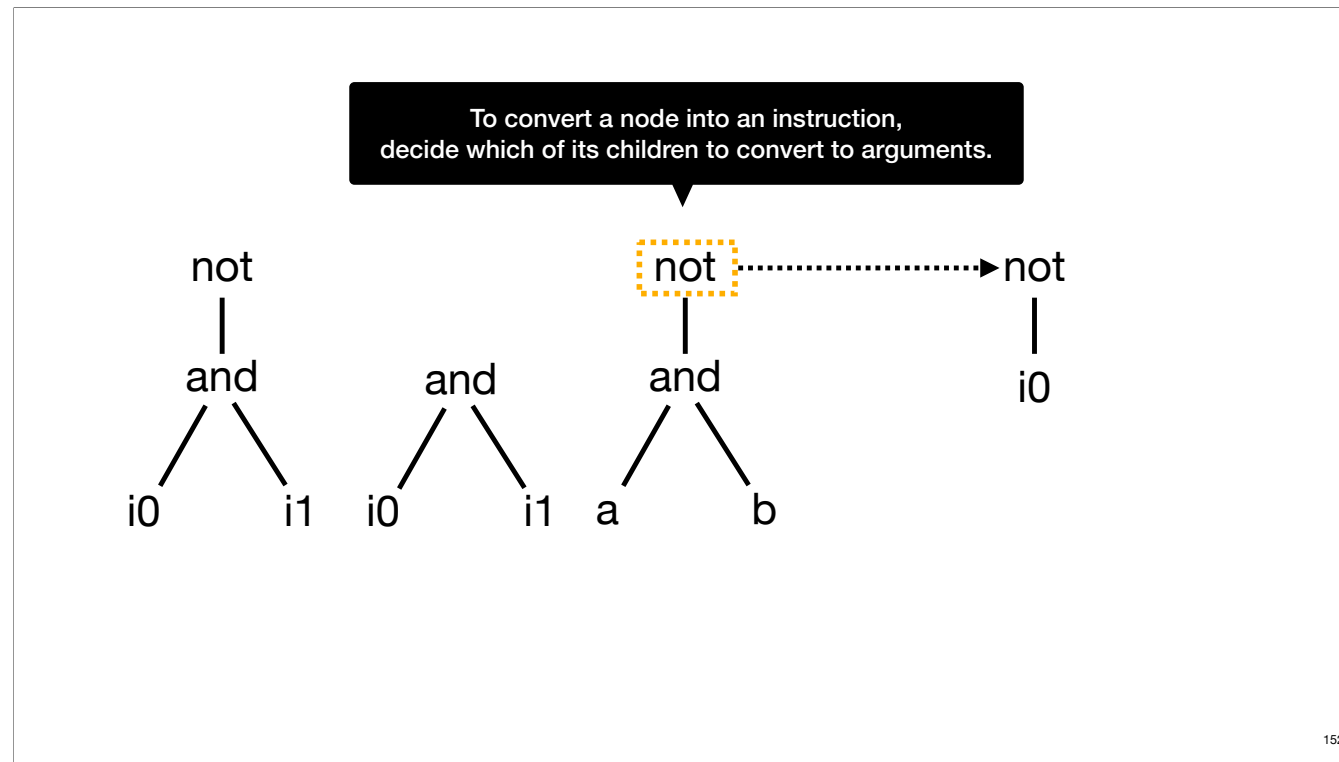


But when we look at the not expression, now we have a choice!

To convert a node into an instruction,
decide which of its children to convert to arguments.



We can either choose to keep the and expression, which produce this instruction, or



We can replace the and with a placeholder.

```

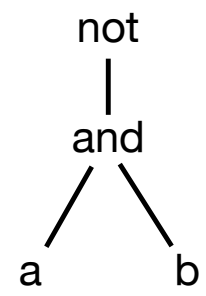
(binop ?op ?bw (apply (instr ?ast0 ?canonical-args0) ?args0) (apply (instr ?ast1 ?canonical-args1) ?args1))
=> (apply (instr (binop-ast ?op ?bw ?ast0 ?ast1) (canonicalize (concat ?args0 ?args1)))
  (concat ?args0 ?args1))

(binop ?op ?bw ?left (apply (instr ?ast1 ?canonical-args1) ?args1))
=> (apply (instr (binop-ast ?op ?bw (hole ?bw) ?ast1) (canonicalize (concat (list ?left) ?args1)))
  (concat (list
    (binop ?op ?bw (apply (instr ?ast0 ?canonical-args0) ?args0) (apply (instr ?ast1 ?canonical-args1) ?args1))
    (concat ?args0 (list ?right))))))

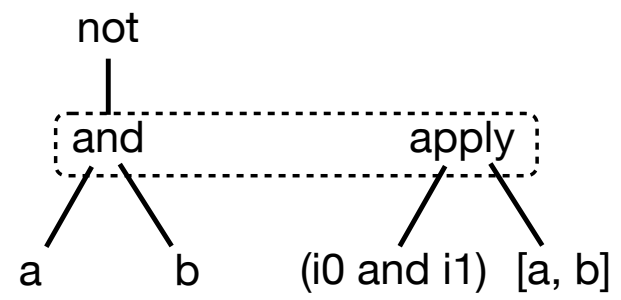
(binop ?op ?bw ?a ?b)
=> (apply (instr (binop-ast ?op ?bw (hole ?bw) (hole ?bw)) (canonicalize (list ?a ?b)))
  (list ?a ?b))

```

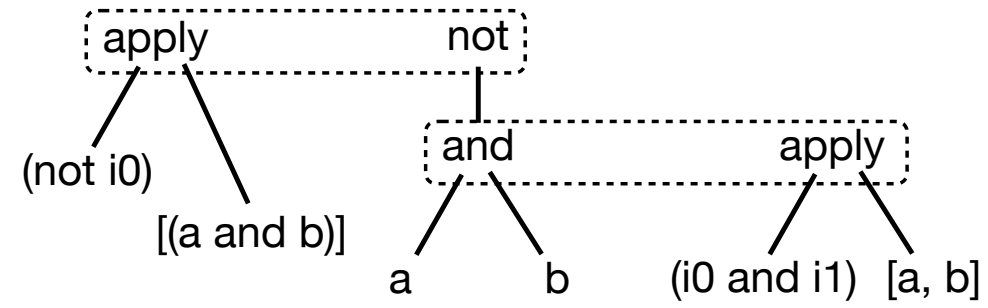
This can be encoded as a small set of rewrites in egg!



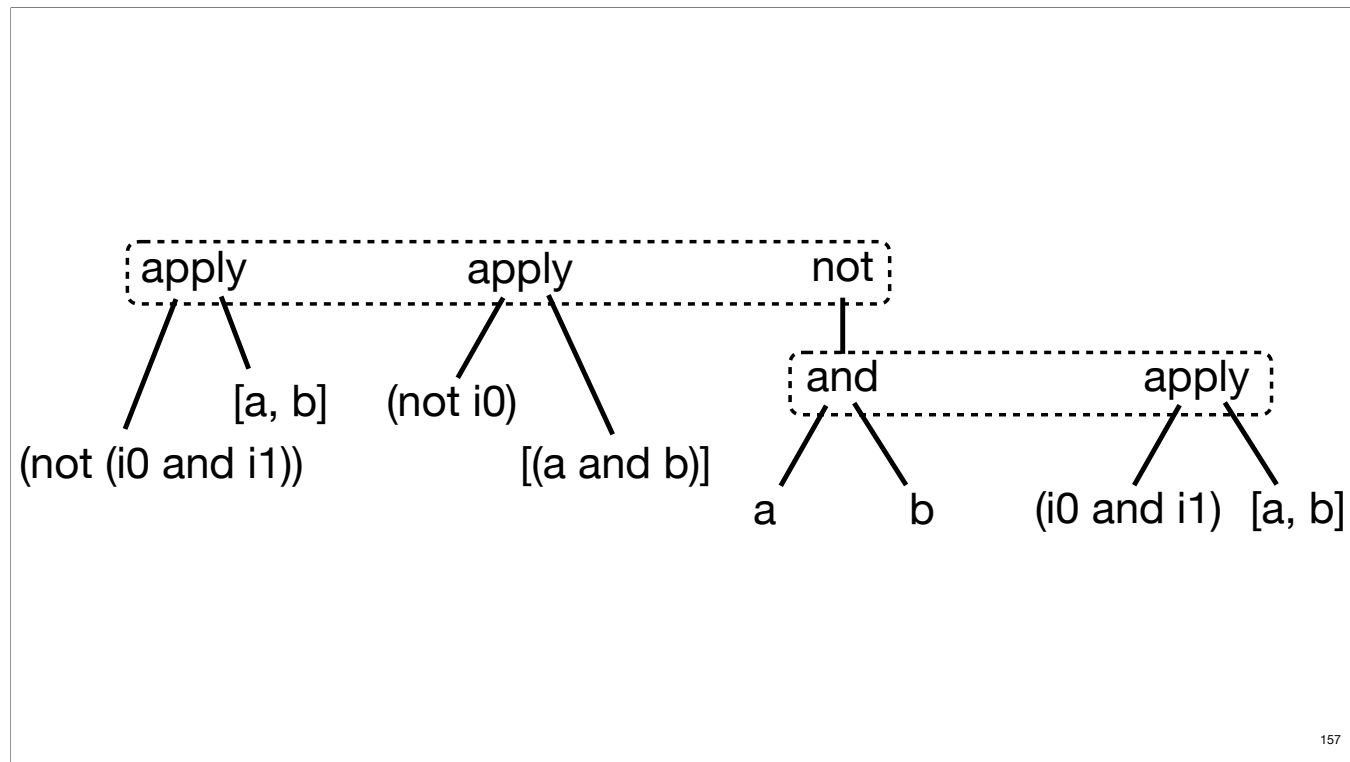
Running the rewrites over an egraph looks something like this.



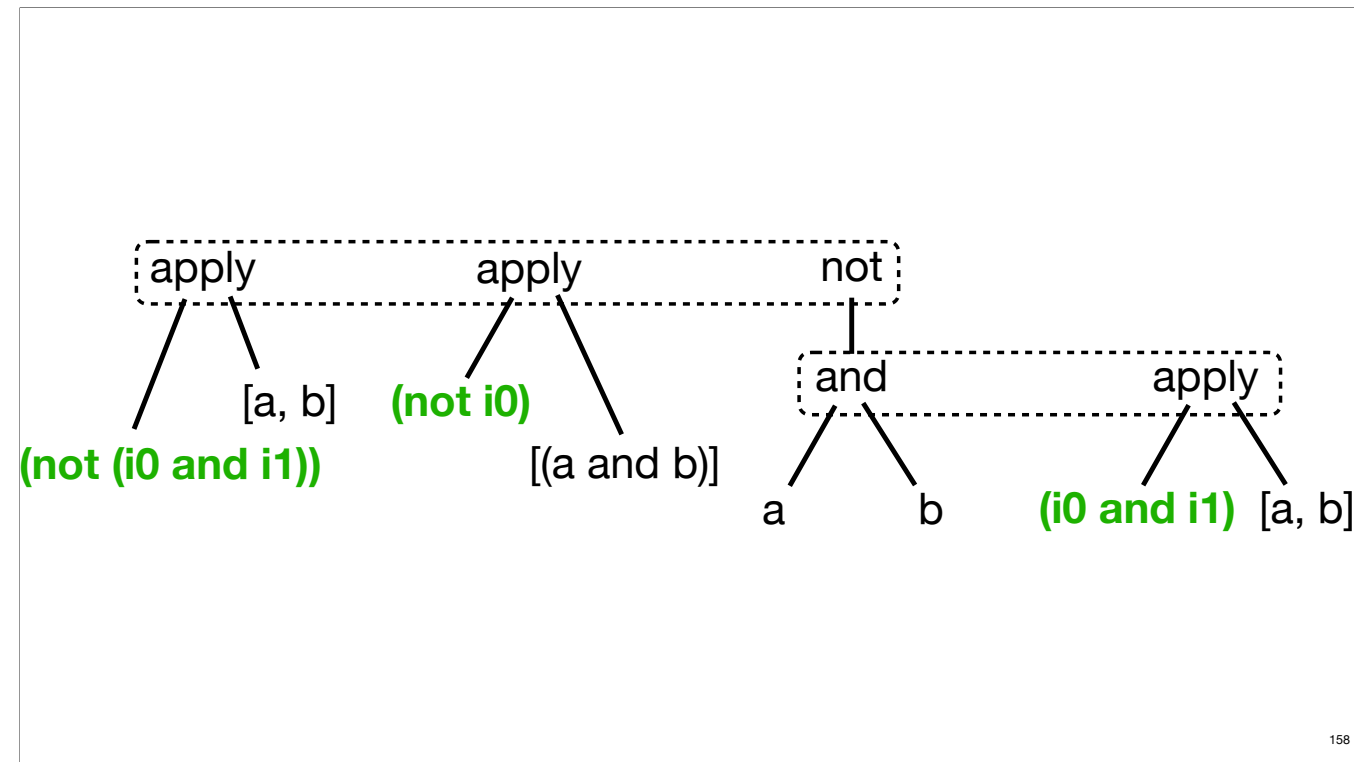
First, we'll rewrite the and expression into the application of the i0 and i1 instruction on arguments a and b.



Then we'll rewrite the not expression similarly, to the application of the not i0 instruction onto a and b.
Finally, when we see that we have two instructions in sequence, that is a not instruction being applied to the result of an and instruction,



We have a rewrite that fuses them into a single fused instruction, not i0 and i1.



From here, it's easy to pull out the enumerated instructions from the egraph.

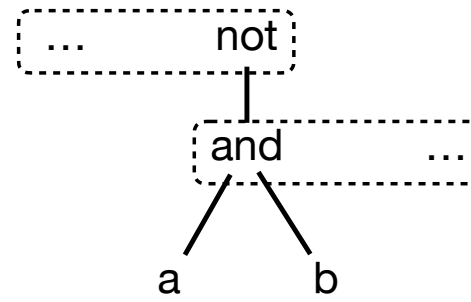
**We can even apply other rewrites
simultaneously!**

De Morgan's law

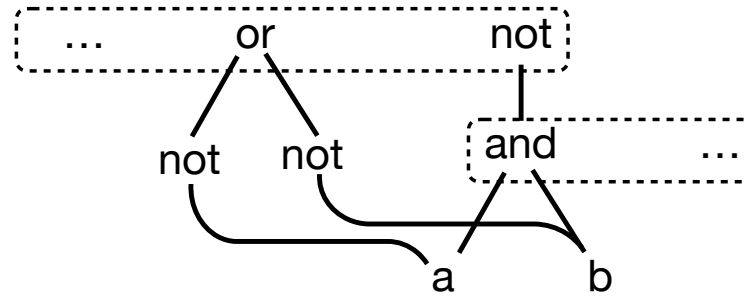
$$\text{not } (a \text{ and } b) ==> (\text{not } a) \text{ or } (\text{not } b)$$

160

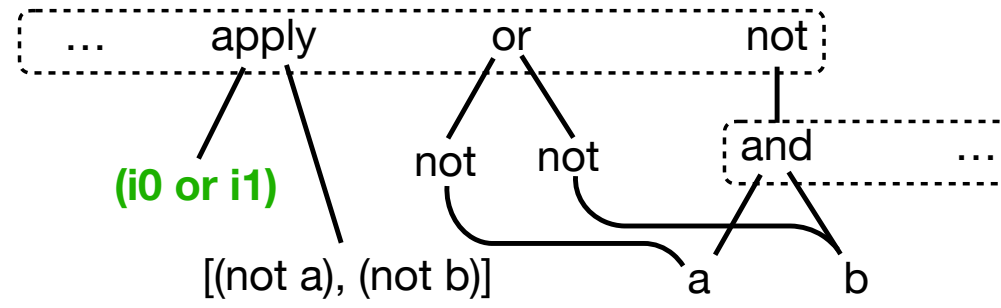
So for example, if we apply one of Demorgan's laws...



...onto our egraph...



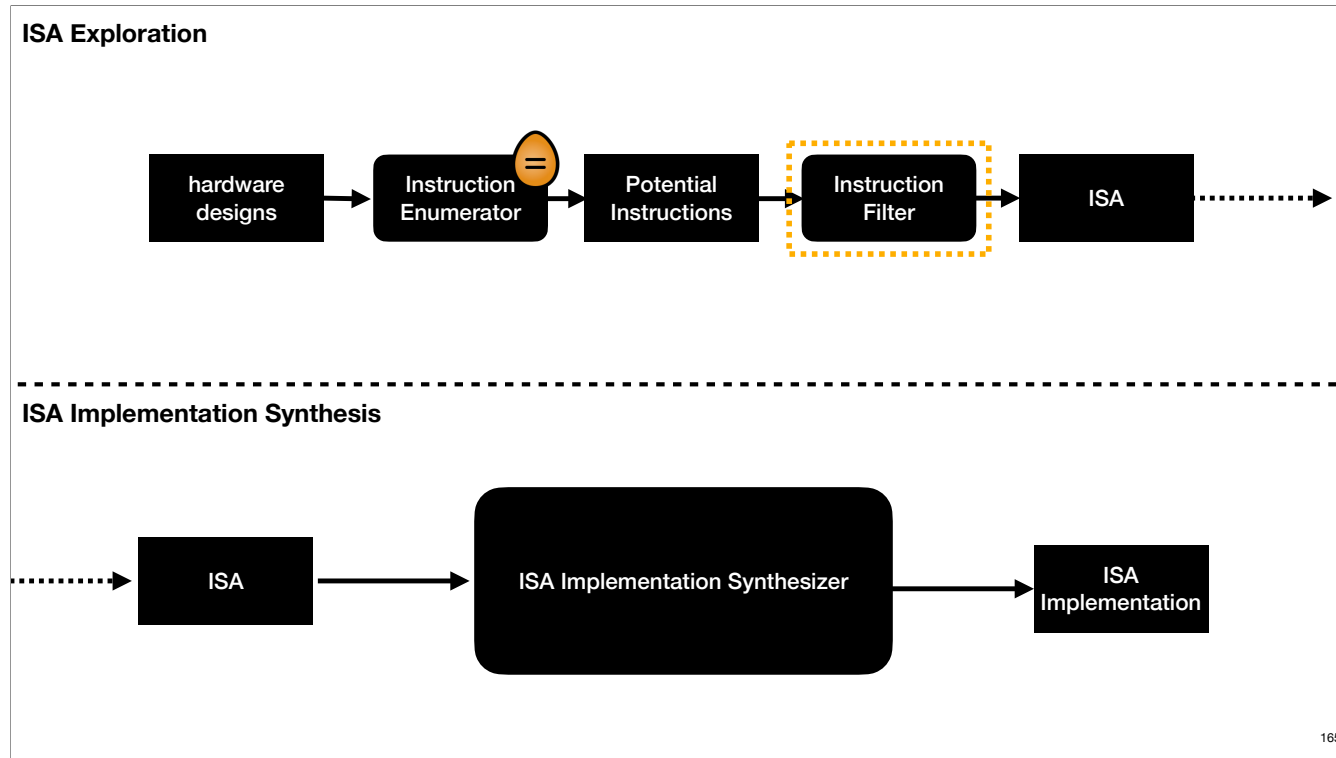
...we get a completely new expression. If we are running the enumeration rewrites simultaneously, the rewrites will enumerate the instructions in this new expression.



Our potential
instructions!

i0 and i1
not (i0 and i1)
not i0
i0 or i1

We can pull all of the instructions we found out of the egraph—this becomes our list of potential instructions for the ISA.



Next, we filter the instructions to produce our final ISA.

i0 and i1
not (i0 and i1)
not i0
i0 or i1

In this step, we filter down the potentially long list of instructions based on user preference.

i0 and i1
~~not (i0 and i1)~~ ← Too specialized — filter it out!
not i0
i0 or i1

167

For example, we might filter out this instruction as being too specialized.

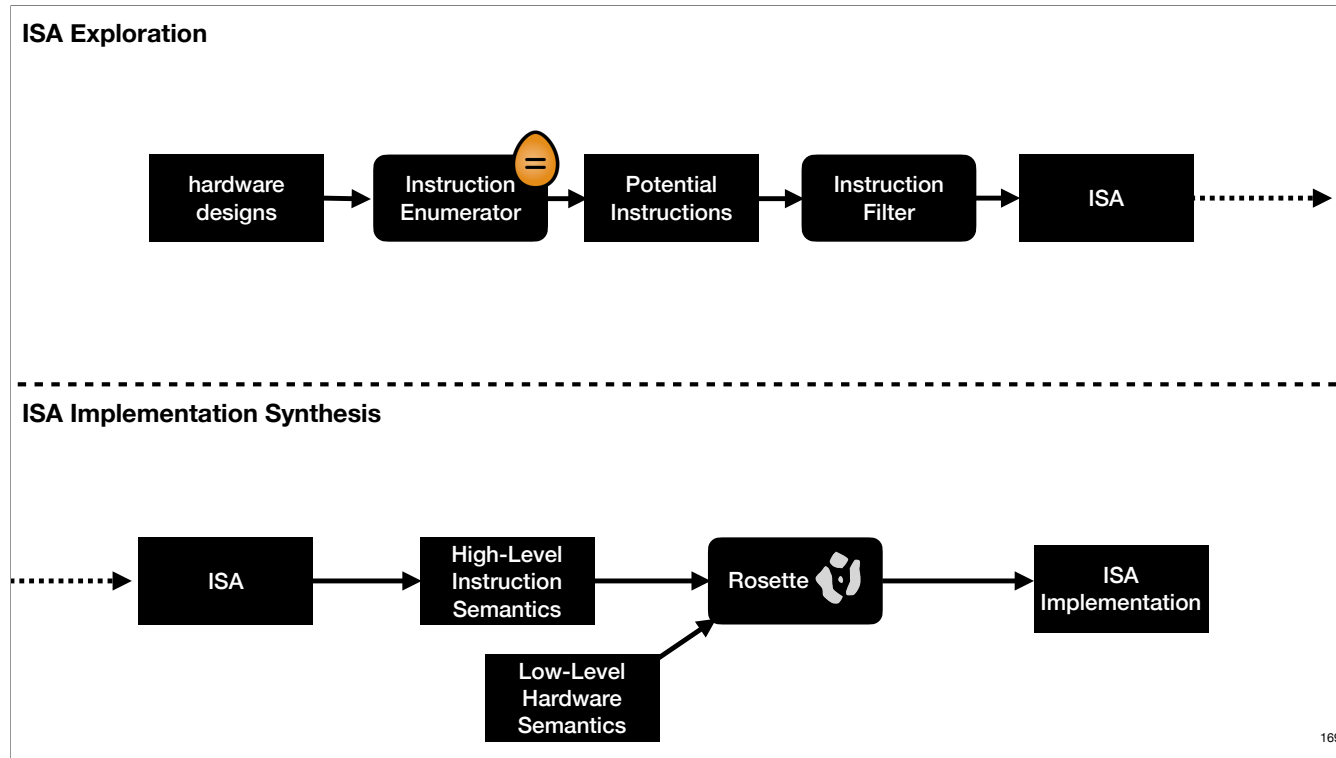
In this toy example, we're likely not going to want to filter out any of these instructions, but as our list of instructions gets long, filtering will be important to keep the size of the isa reasonable.

ISA Exploration



ISA Implementation Synthesis





At the core of the ISA implementation synthesizer is a tool called Rosette.

Rosette is a synthesis tool which allows us to ask:

Rosette is a synthesis tool which allows us to ask:

For all inputs a and b,

Rosette is a synthesis tool which allows us to ask:

For all inputs a and b ,
find an implementation of `low-level-FPGA-impl` such that

Rosette is a synthesis tool which allows us to ask:

For all inputs a and b ,

find an implementation of `low-level-FPGA-impl` such that

`low-level-FPGA-impl(a, b) == high-level-instr-impl(a, b)`

Rosette is a synthesis tool which allows us to ask:

For all inputs a and b ,

find an implementation of `low-level-FPGA-impl` such that

`low-level-FPGA-impl(a, b) == high-level-instr-impl(a, b)`

To do so, we need to define the **high-level semantics** of the instruction,

Rosette is a synthesis tool which allows us to ask:

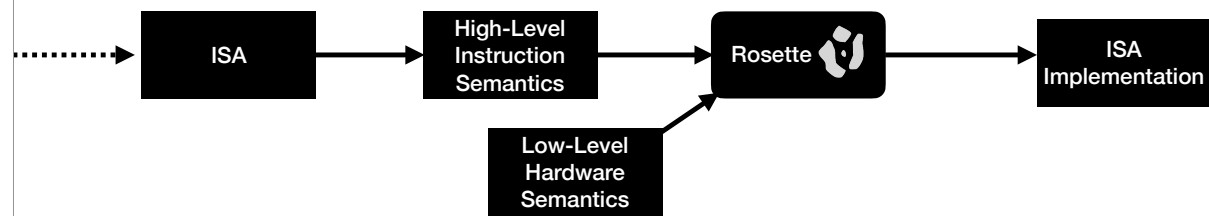
For all inputs a and b ,
find an implementation of `low-level-FPGA-impl` such that
 $\text{low-level-FPGA-impl}(a, b) == \text{high-level-instr-impl}(a, b)$

To do so, we need to define the **high-level semantics** of the instruction,
and the **low-level semantics** of the FPGA.

ISA Exploration



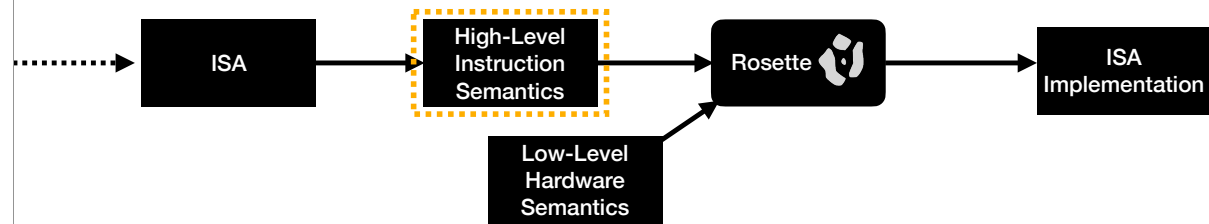
ISA Implementation Synthesis



ISA Exploration



ISA Implementation Synthesis



4.3.2 Bitwise Operators

bvnot
bvand
bvor
bvxor
bvshl
bvlsr
bvashr

4.3.3 Arithmetic Operators

bvneg
bvadd
bvsub
bvmul
bvsdiv
bvudiv
bvsrem
bvurem
bvsmmod

4.3.4 Conversion Operators

concat

4.3.2 Bitwise Operators

```
(bvnot x) → (bitvector n)      procedure  
x : (bitvector n)
```

Returns the bitwise negation of the given bitvector value.

Examples:

```
> (bvnot (bv -1 4))  
(bv #x0 4)  
> (bvnot (bv 0 4))  
(bv #xf 4)  
> (define-symbolic b boolean?)  
> (bvnot (if b 0 (bv 0 4))) ; This typechecks only when b is false,  
(bv #xf 4)  
> (vc)                      ; so Rosette emits a corresponding assertion.  
(vc #t (! b))
```

```
(bvand x ...+) → (bitvector n)      procedure  
x : (bitvector n)  
(bvor x ...+) → (bitvector n)  
x : (bitvector n)  
(bvxor x ...+) → (bitvector n)
```

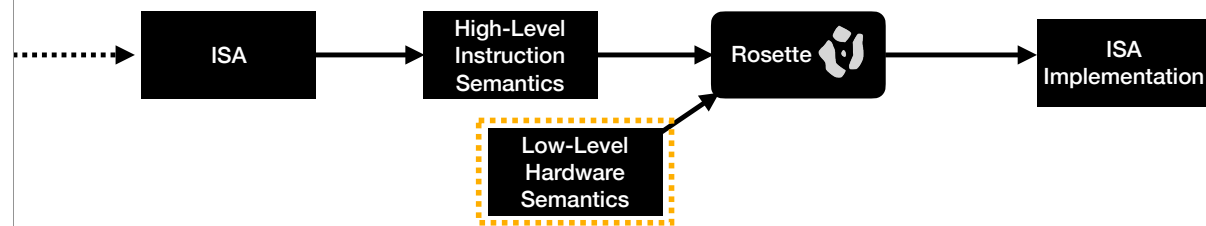
i0 and i1
not (i0 and i1)
not i0
i0 or i1

$i0 \text{ and } i1 \longrightarrow (\text{bvand } i0 \ i1)$
 $\text{not } (i0 \text{ and } i1) \longrightarrow (\text{bvnot } (\text{bvand } i0 \ i1))$
 $\text{not } i0 \longrightarrow (\text{bvnot } i0)$
 $i0 \text{ or } i1 \longrightarrow (\text{bvor } i0 \ i1)$

ISA Exploration



ISA Implementation Synthesis



To capture architecture-level semantics of FPGAs, we simply build an *interpreter* for each FPGA component!


```
(define (lut memory inputs)  
  (let* ([inputs (zero-extend inputs (bitvector (length (bitvector->bits memory))))])  
    (extract 0 0 (bvlshr memory inputs))))
```

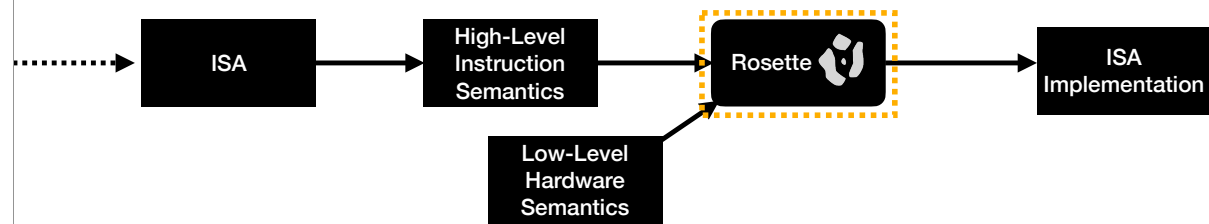
```
(define (lut memory inputs)
  (let* ([inputs (zero-extend inputs (bitvector (length (bitvector->bits memory)))))])
    (extract 0 0 (bvlshr memory inputs))))
```

```
(define (ultrascale-plus-lut6-2 memory inputs)
  (let* ([lut5-0 (lut (extract 63 32 memory) (extract 4 0 inputs))]
        [lut5-1 (lut (extract 31 0 memory) (extract 4 0 inputs))]
        [06 (if (bitvector->bool (bit 5 inputs)) lut5-0 lut5-1)]
        [05 lut5-1])
    (list 05 06)))
```

ISA Exploration



ISA Implementation Synthesis



i0 and i1

not (i0 and i1)

not i0

i0 or i1

For all inputs a and b ,
find an implementation of `low-level-and-impl` such that
`low-level-and-impl(a,b) == high-level-and-impl(a,b)`

For all inputs a and b ,
find an implementation of `low-level-and-impl` such that
`low-level-and-impl(a,b) == (bvand a b)`

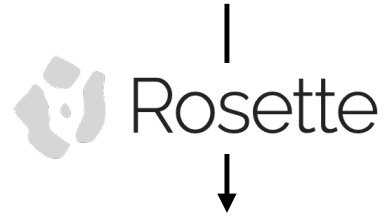
For all inputs a and b,
find a setting of memory such that
`(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)`

For all inputs a and b,
find a setting of memory such that
`(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)`

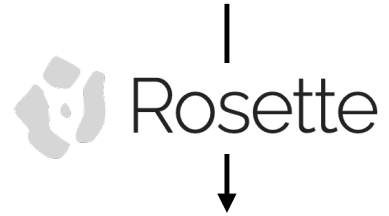
184

So we plug that into rosette, and rosette gives us an answer!

For all inputs a and b,
find a setting of memory such that
`(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)`



For all inputs a and b,
find a setting of memory such that
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)

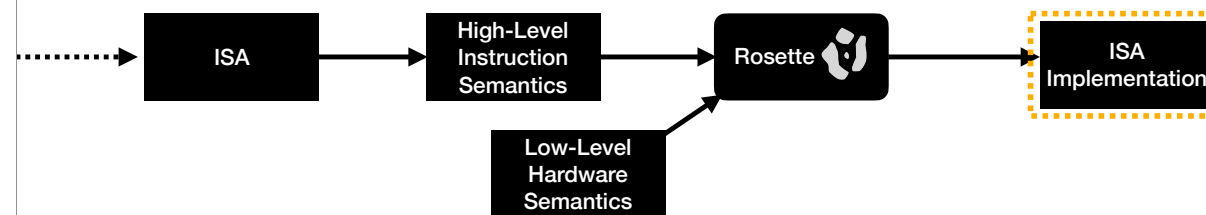


memory:
(bv #x000000000000000000000000 64)

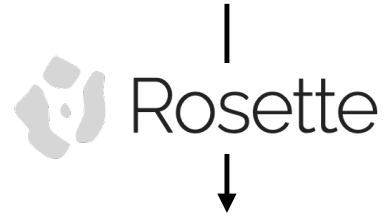
ISA Exploration



ISA Implementation Synthesis

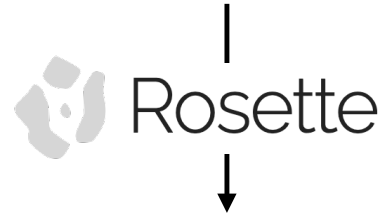


For all inputs a and b,
find a setting of memory such that
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)



memory:
(bv #x000000000000000000000000 64)

For all inputs a and b,
find a setting of memory such that
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)



memory:
(bv #x000000000000000000000000 64)

```
module and(a, b, out);  
  LUT2 #(  
    .INIT(4'h8)  
  ) _0_ (.I0(a), .I1(b), .O(out));  
endmodule
```


For all inputs a and b,
find a setting of memory such that
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)



memory:
(bv #x000000000000000008 64)

module and(a, b, out);
 LUT2 #(
 .INIT(4'h8)
) _0_ (.I0(a), .I1(b), .O(out));
endmodule

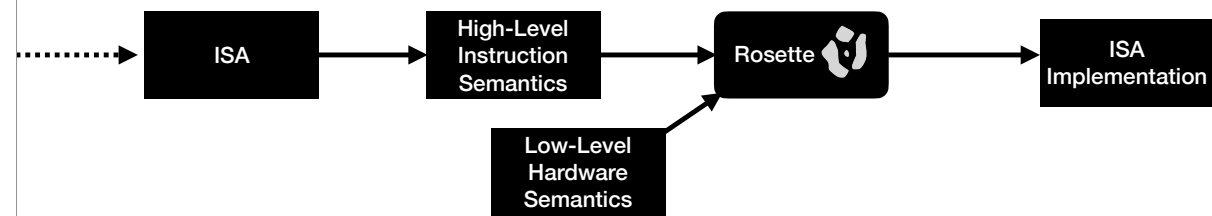
To support new FPGA architectures...

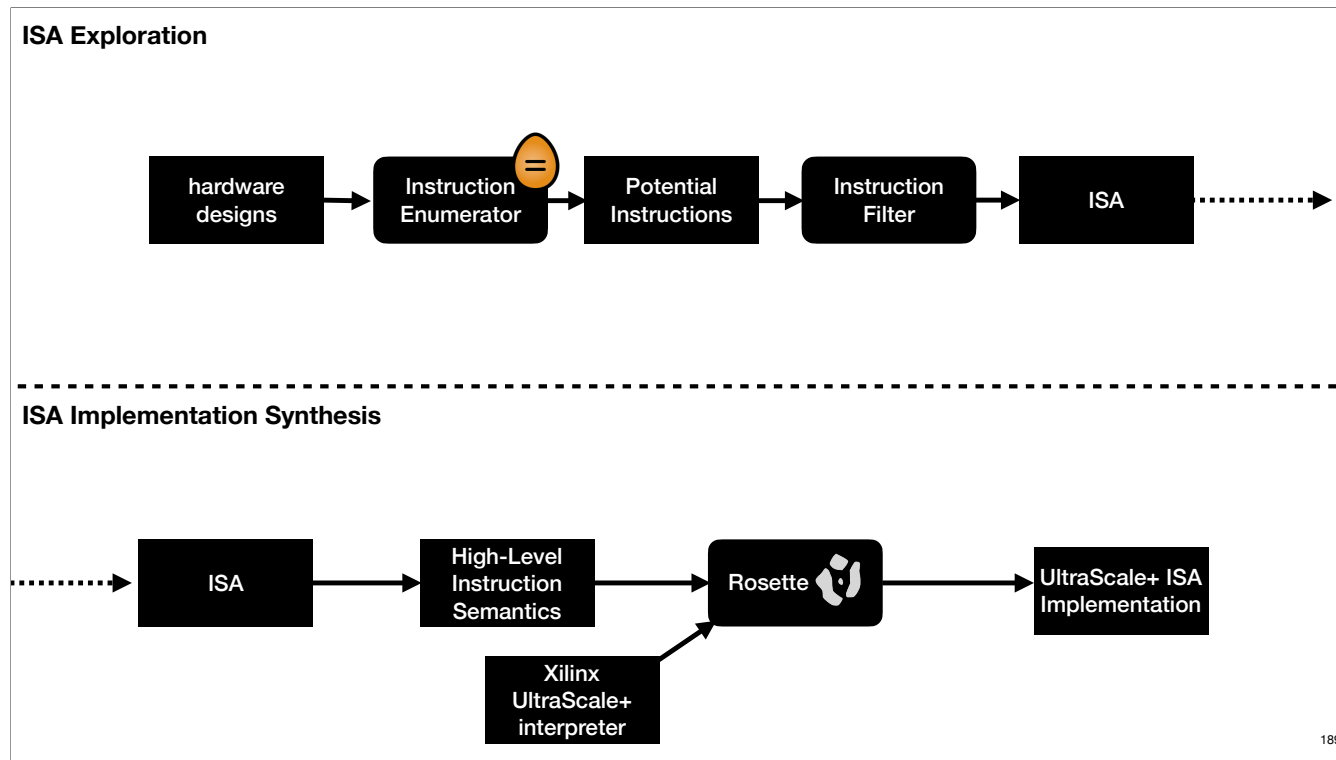
**To support new FPGA architectures...
...just provide an interpreter!**

ISA Exploration



ISA Implementation Synthesis





189

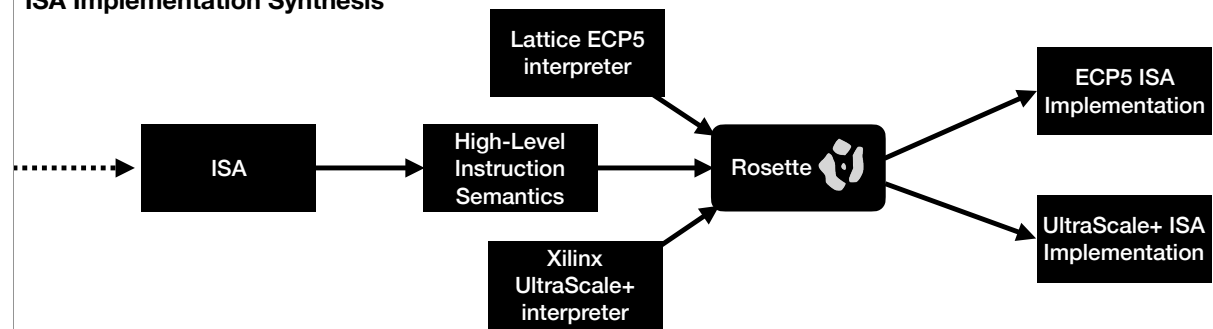
So we demonstrated synthesizing for Xilinx ultrascale+, but if I wanted to synthesize (Build)

For lattice ECP5, which is another FPGA, I would just need to provide an interpreter for ECP5's lookup tables.

ISA Exploration



ISA Implementation Synthesis



Finally, to compile a new design, we just need to:

Finally, to compile a new design, we just need to:

1. Insert it into the egraph

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

If the design isn't covered with the current ISA, we can:

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

If the design isn't covered with the current ISA, we can:

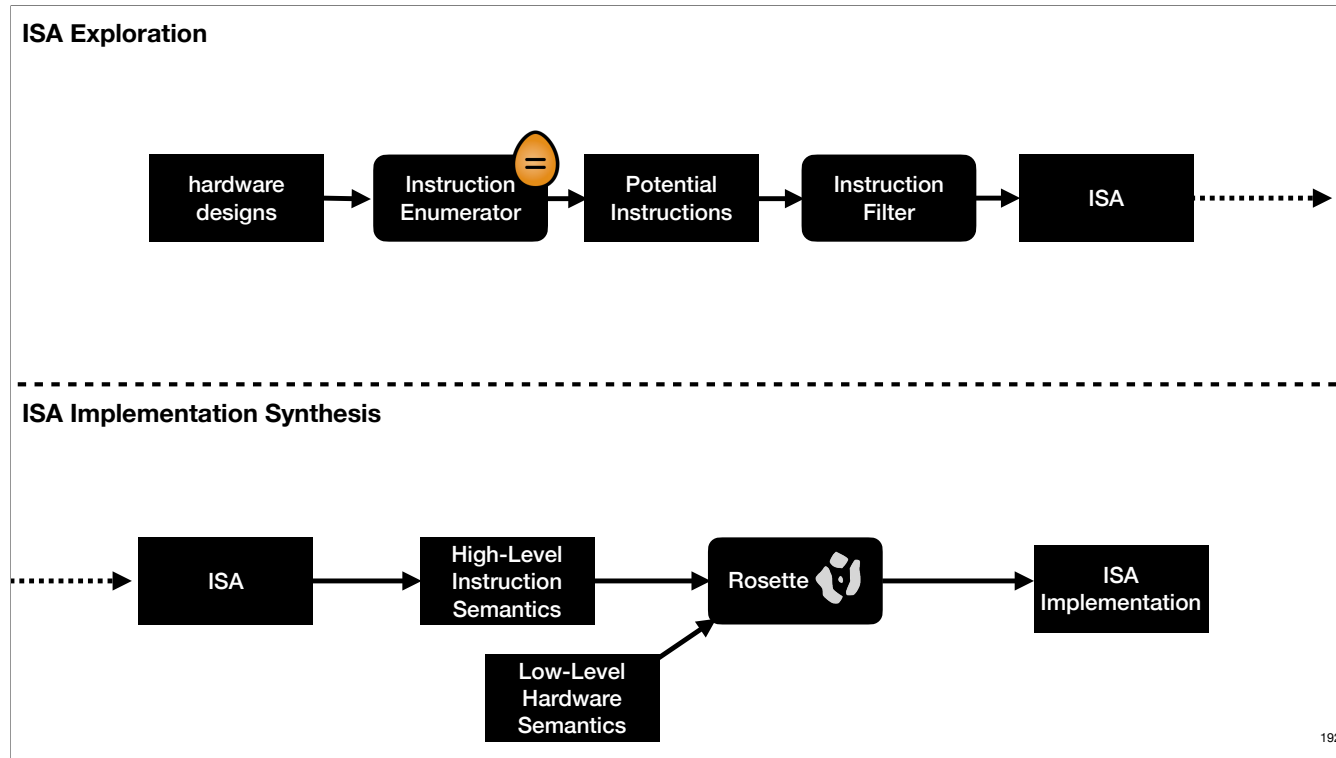
- Run rewrites to find alternative implementations of the design

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

If the design isn't covered with the current ISA, we can:

- Run rewrites to find alternative implementations of the design
- Find a minimal set of new instructions to add to the ISA



So that is lakeroad: a tool that automatically defines and implements ISAs for FPGAs.

**Automatically generating compiler backends
from explicit, formal hardware models**

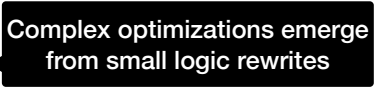

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

**Automatically generating compiler backends
from explicit, formal hardware models**

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Complex optimizations emerge
from small logic rewrites

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations, 
- reduces development time, and 
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations, 
- reduces development time, and 
- enables verification. 

Proposed Evaluation

Proposed Evaluation

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Paper 2: ISA Exploration and Lakeroad End-to-End

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Paper 2: ISA Exploration and Lakeroad End-to-End

- Goal: Demonstrate ability to enumerate a large space of interesting instructions; demonstrate fast compilation using the egraph.

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Paper 2: ISA Exploration and Lakeroad End-to-End

- Goal: Demonstrate ability to enumerate a large space of interesting instructions; demonstrate fast compilation using the egraph.
- We will run Lakeroad end-to-end on a large corpus of hardware benchmarks (from sources like MachSuite)

In Closing

**Automatically generating compiler backends
from explicit, formal hardware models**

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

**So far, I have provided evidence for this thesis
through Glenside and its application in 3LA.**

**So far, I have provided evidence for this thesis
through Glenside and its application in 3LA.**

**I plan to demonstrate this thesis once more
through Lakeroad.**

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

Winter Quarter 2023: Fulfill final TA requirement

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

Winter Quarter 2023: Fulfill final TA requirement

Winter/Spring Quarter 2023: Deal with Lakeroad and 3LA resubmissions

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

Winter Quarter 2023: Fulfill final TA requirement

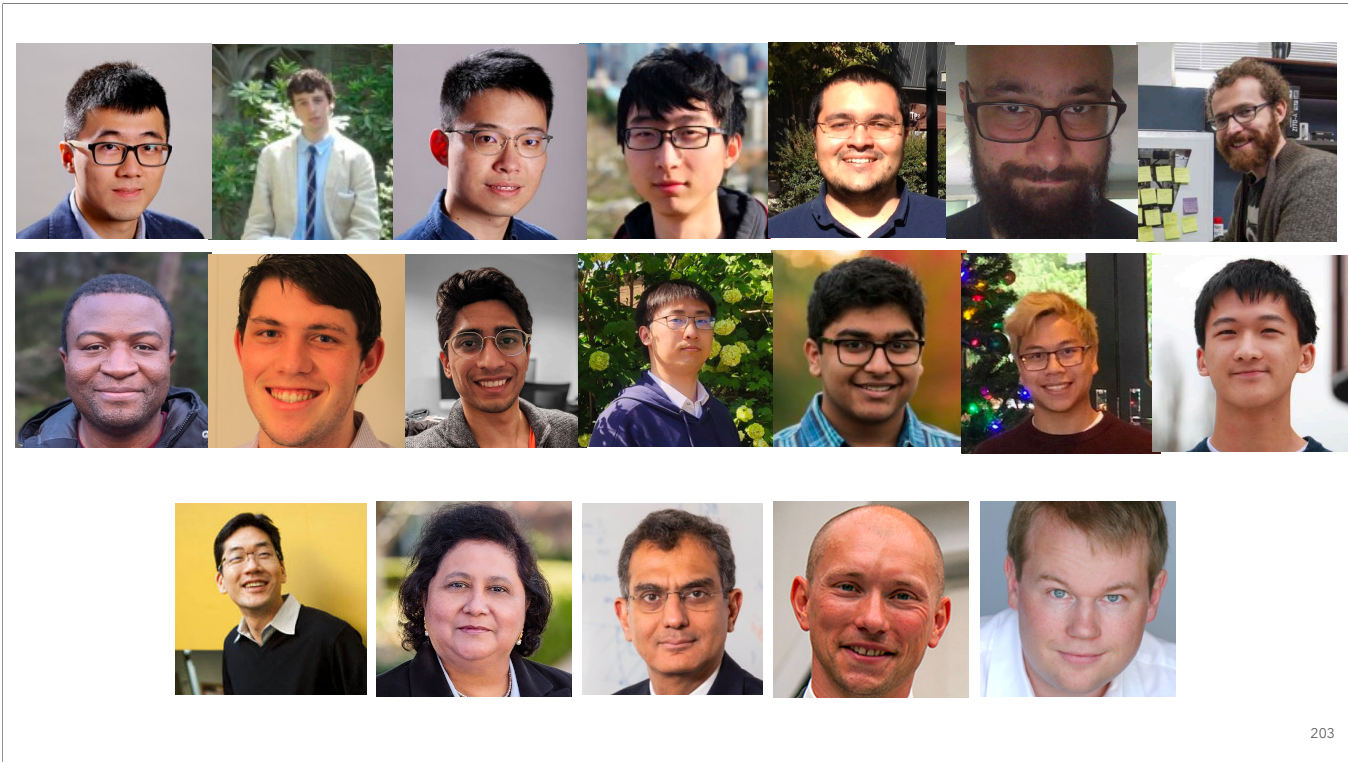
Winter/Spring Quarter 2023: Deal with Lakeroad and 3LA resubmissions

Spring Quarter 2023: Write thesis and defend

Acknowledgements









Thank you!

Extra Slides

Rest of Glenside Talk

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
 - Reimplementing Matrix Multiplication with Access Patterns
 - Implementing 2D Convolution with Access Patterns
 - Hardware Mapping as Program Rewriting
 - Flexible Hardware Mapping with Equality Saturation

208

Now that we've defined our core abstraction, access patterns, we will demonstrate how access patterns and the Glenside IR enable rich term rewriting over tensor programs.

We begin by first showing how Glenside cleanly represents two core deep learning kernels: matrix multiplication and 2D convolution.

We then return to our original task, and show how Glenside can be used to implement hardware mapping via program rewriting.

Finally, using equality saturation, a modern program rewriting framework, we demonstrate how Glenside can discover ways to flexibly map other kernels onto hardware that was designed for a specific purpose: in this case, mapping 2D convolution to matrix multiplication hardware.

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
 - Reimplementing Matrix Multiplication with Access Patterns
 - Implementing 2D Convolution with Access Patterns
 - Hardware Mapping as Program Rewriting
 - Flexible Hardware Mapping with Equality Saturation

209

Let's begin by reimplementing matrix multiplication with access patterns and the Glenside IR.

Given matrices A and B, pair each row of A with each column of B, compute their dot products, and arrange the results back into a matrix.

210

Remember, we defined our matrix multiplication algorithm as,

```

(car (prod
  (access A 1)
  (list 1 0)))) ; ((3), (4))

```

211

We will now implement matrix multiplication with Glenside.

In these slides, the Glenside expression will appear on the left, and the access pattern shape of the Glenside expression on that line will appear as a lisp-style comment on the right.

To begin implementing matrix multiplication, we first need to access A as a list of its rows, and access B as a list of its columns.

Accessing A as a list of its rows is straightforward. Given a tensor and a dimension index, the access operator converts a tensor into an access pattern by splitting the tensor's shape at the given dimension index. In this case, A is a three-comma-four shaped tensor, so accessing it at dimension 1 produces this access pattern shape.

This access pattern views A as a three-length vector of four-length vectors—in other words, a list of the rows of A.

(CarLProu

(access A 1)

Access A as a list of its rows

(list 1 0)))

; ((3), (4))

```

(car lProd
 (access A 1)

 (list 1 0)))) ; ((3), (4))

; ((4), (2))

```

212

Next, we must access B as a list of its columns.
 Accessing B at dimension 1, however, gives a list of B's rows.

```

(CartProd
  (access A 1)

  (list 1 0)))) ; ((3), (4))
                  ; ((2), (4))
                  ; ((4), (2))

```

213

Thus, we employ transpose, which we call an access pattern transformer in Glenside.

Given an access pattern and a list of integers, transpose reorders the dimensions of the access pattern.

In this case, we simply swap the dimensions of the access pattern, giving us a 2-length vector of 4-length vectors, or a list of B's columns.

(car lProd
(access A 1)

(list 1 0))) ; ((3), (4))
; ((2), (4))
; ((4), (2))

Access B as a list
of its rows, then
transpose into a
list of its columns


```

(cartProd
  (access A 1)
  (access B 1)
  (list 1 0)))
; ((3, 2), (2, 4))
; ((3), (4))
; ((2), (4))
; ((4), (2))

```

214

Now that we have a list of A's rows and a list of B's columns, we use Glenside's cartesian product transformer to create every row-column pair. Notice that the resulting access pattern is a three-comma-two shaped matrix, each element of which is a two-comma-four shaped matrix. The three-comma-two shape preserves the shape information from A and B, and will give us the final shape of the output matrix. The two-comma-four shape represents a pair of 4-length vectors—one row of A paired with one column of B.

(carlprod

(access A 1)

Create every row-column pair

(access B 1)

(list 1 0)))

; ((3, 2), (2, 4))

; ((3), (4))

; ((2), (4))

; ((4), (2))

```

(dot-prod
  (access A 1)
  (transpose
    (access B 1)
    (list 1 0))))
; ((3, 2), ())
; ((3, 2), (2, 4))
; ((3), (4))
; ((2), (4))
; ((4), (2))

```

215

Finally, we compute the dot product of each row-column pair. Glenside's compute expression maps a given operator over the compute dimensions of the access pattern, i.e. the second shape tuple, completely ignoring the access dimensions, i.e. the first shape tuple.

In this case, the dot product operator reduces each pair of 4-length vectors to a scalar.

And that is our full representation of matrix multiplication in Glenside! As you can see, our representation here is pure, low-level, and conveniently avoids relying on binding.

(car lProd

(
Compute dot product of every row-column pair

(transpose

(access B 1)

(list 1 0))))

; ((3, 2), ())

; ((3, 2), (2, 4))

; ((3), (4))

; ((2), (4))

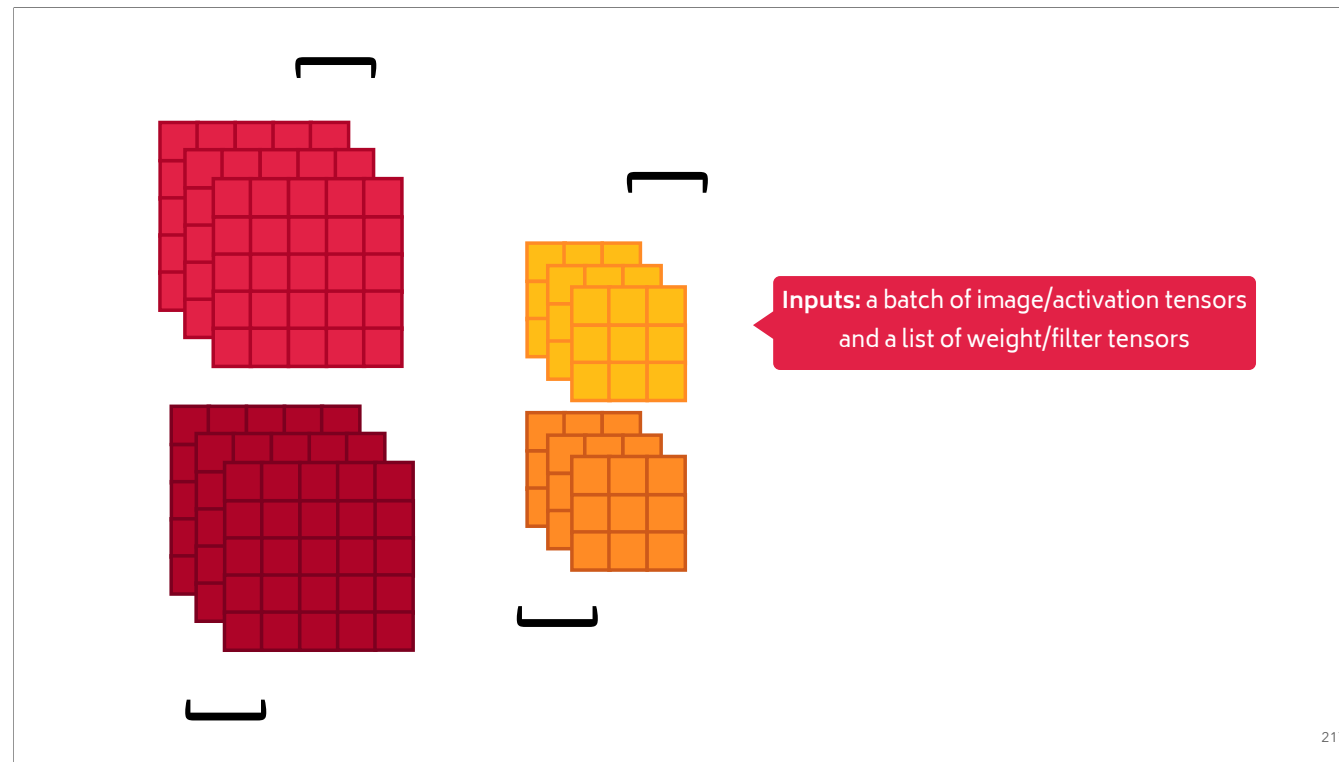
; ((4), (2))

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
 - Reimplementing Matrix Multiplication with Access Patterns
 - Implementing 2D Convolution with Access Patterns
 - Hardware Mapping as Program Rewriting
 - Flexible Hardware Mapping with Equality Saturation

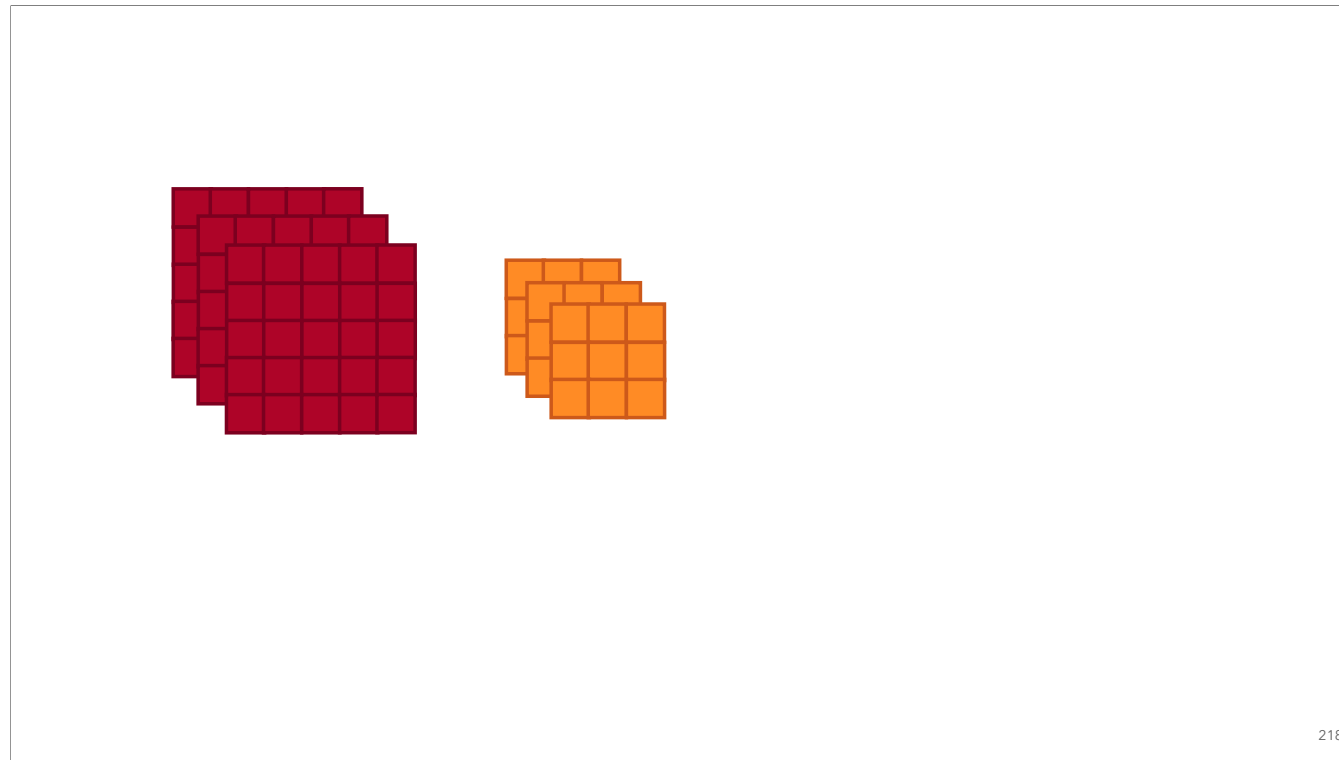
216

Next, we will demonstrate the expressiveness of access patterns by implementing an entirely different kernel—2D convolution—with Glenside.

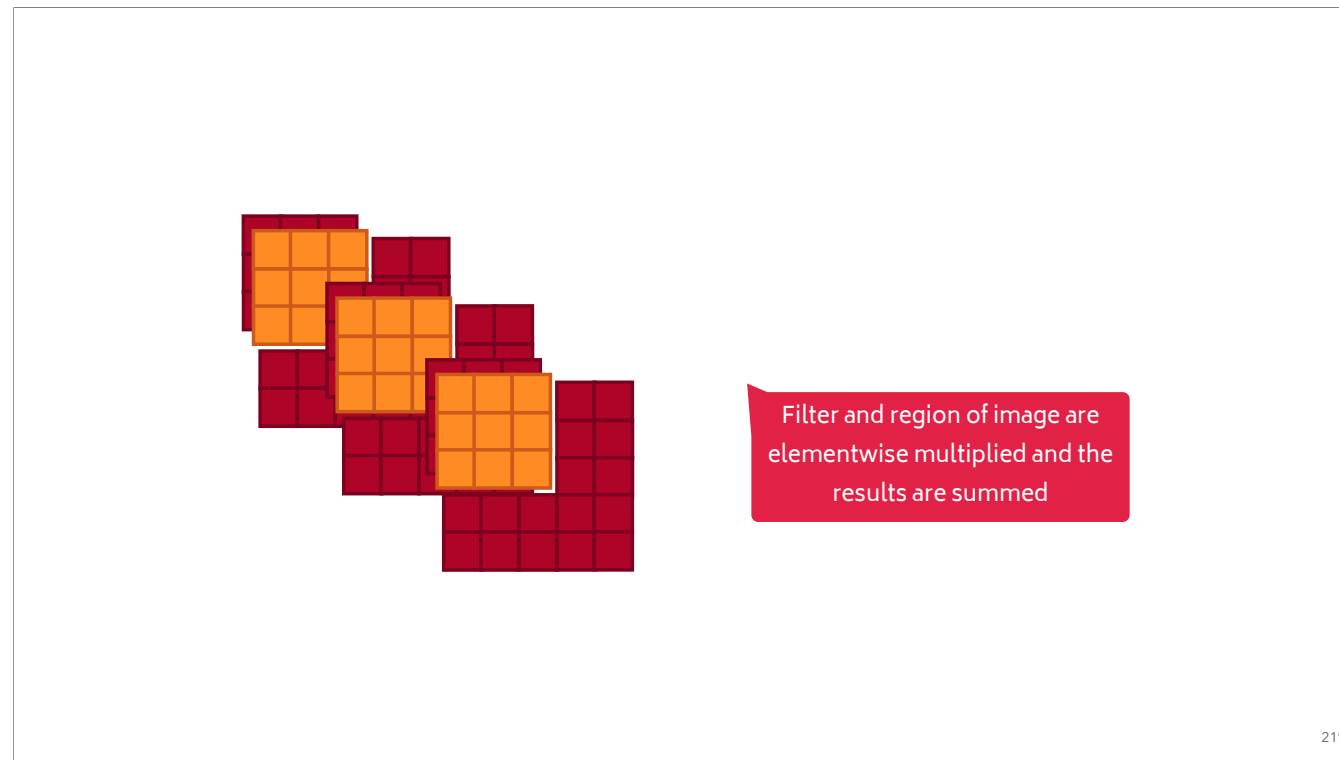


Let's first get a brief refresher on how 2D convolution works.

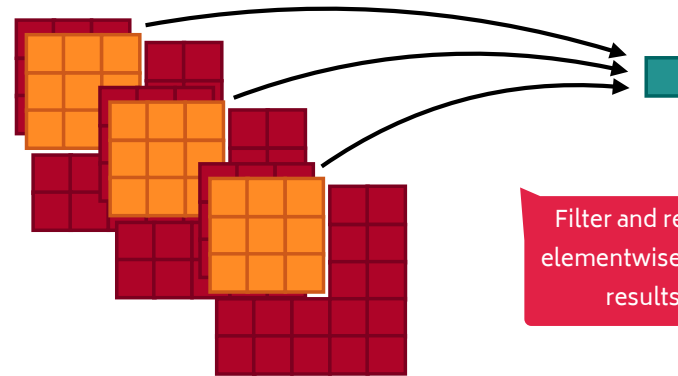
The inputs to 2D convolution are a batch of image or activation tensors, and a list of weight or filter tensors.



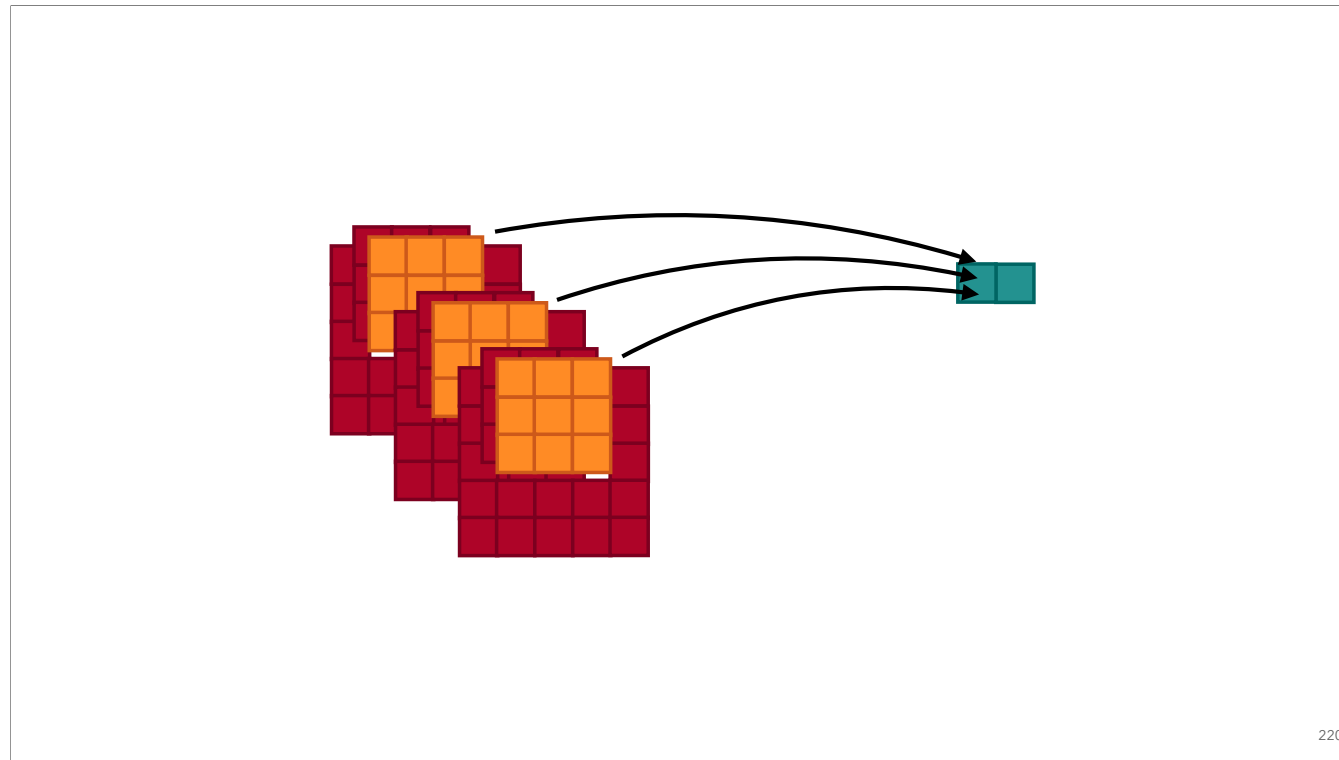
In this example, we will show what happens with just one set of activations and one filter, as the same process is repeated for every set of activations and every filter.



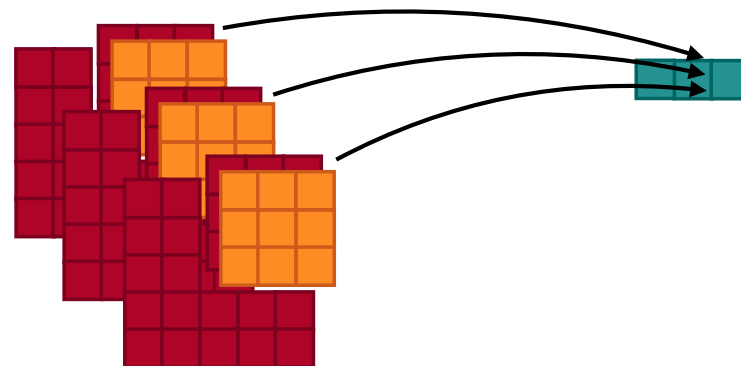
The filter is applied onto a window of the activations by computing an element wise multiplication and sum between the filter and the window of activations.

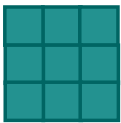


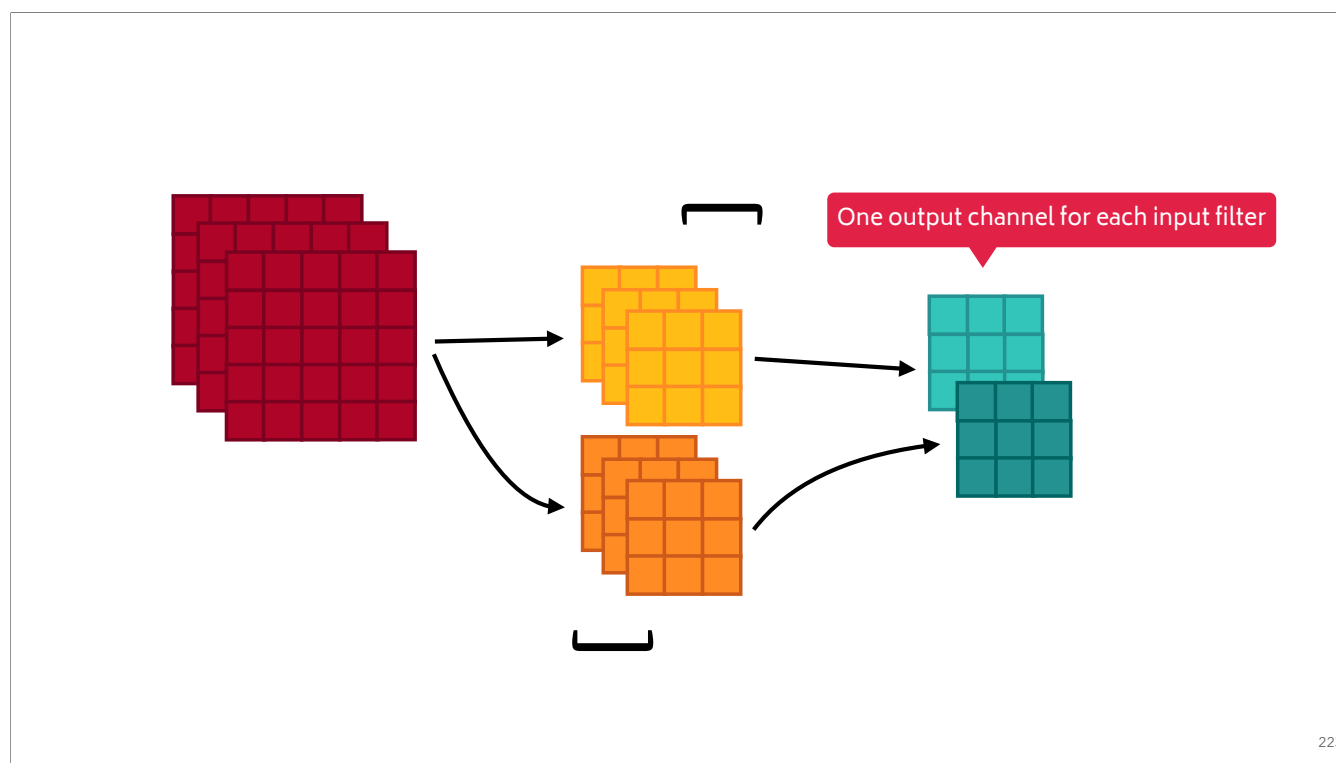
Filter and region of image are
elementwise multiplied and the
results are summed



This element wise multiplication and sum is computed over every possible window of the activations.







This same process is completed for each filter in the set, producing one output channel for each filter.

```

(windows
  (access activations 1)

  Access weights as a list of 3D filters

; ((O), (C, Kh, Kw))

```

224

To implement 2D convolution in Glenside, we begin by accessing the weights as a list of 3 dimensional filters. In this case, there are O filters, where O will be our number of output dimensions, and each filter is of shape C, K_h, K_w , where C represents our number of input channels and K_h and K_w are our filter height and width.

```

(windows
  (access activations 1)

  (list 0 3 1 2)) ; ((N), (C, H, W))

                  ; ((O), (C, Kh, Kw))

```

Access activations as a batch of 3D images

225

Next we access our activations as a batch of N 3 dimensional images, where N is our batch size, and H and W are the input height and width.

```

(windows
  (access activations 1)

  1)
  (list 0 3 1 2)) ; ((N), (C, H, W))

                  ; ((O), (C, Kh, Kw))

```

Form windows over input images

226

Now we must form windows over our activations. As this is a common pattern, also appearing for example in max and average pooling, Glenside provides the windows transformer for this purpose.


```

(window
  (access activations 1)

  1)
  (list 0 3 1 2)) ; ((N), (C, H, W))

  ; ((O), (C, Kh, Kw))

```

These parameters control window shape and strides

227

The windows transformer takes an access pattern, a window shape, and a list of strides.

```

(window
  (access activations 1)

  1)
(list 0 3 1 2))

```

At each location in each new image,
there is a (C, K_h, K_w) -shaped window

```

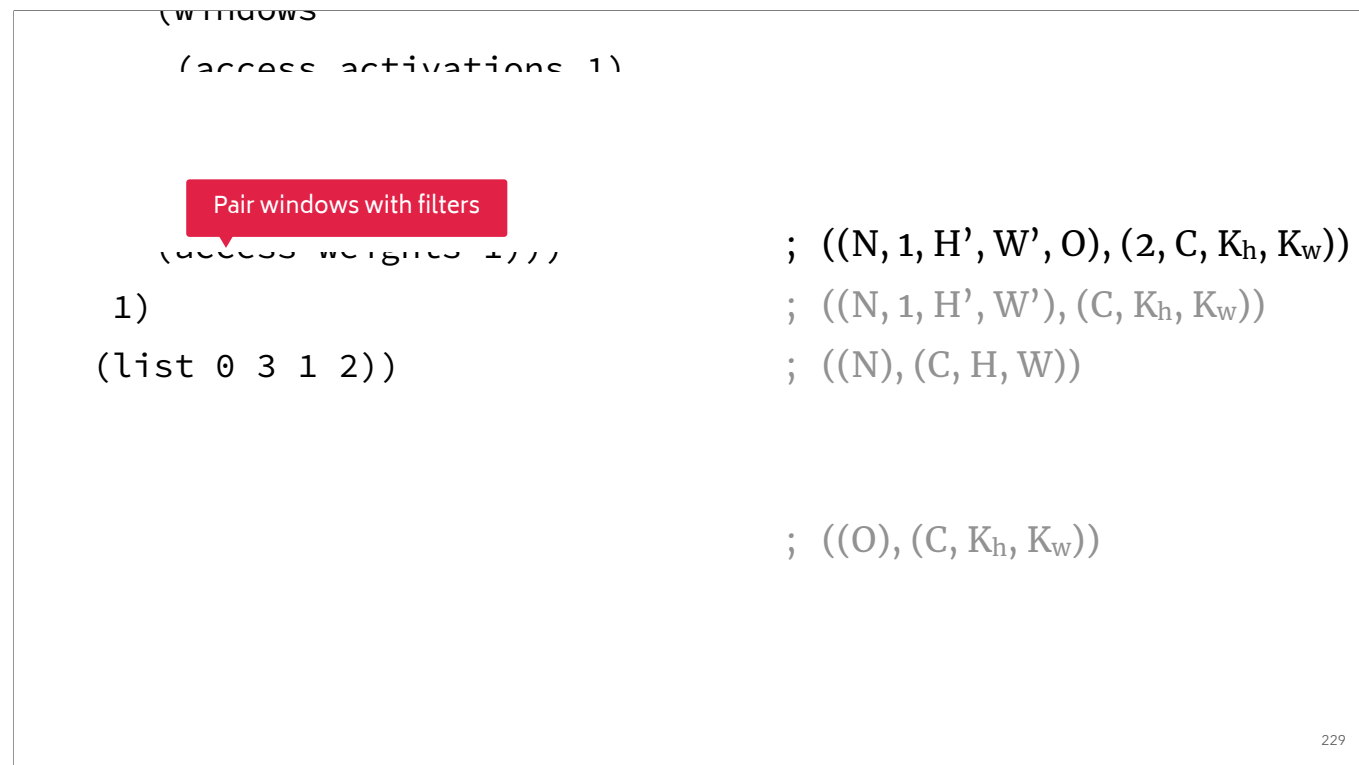
; ((N, 1, H', W'), (C, K_h, K_w))
; ((N), (C, H, W))

; ((O), (C, K_h, K_w))

```

228

The result of the windows transformer is a new batch of images of shape H', W' , where at each location in each new image, there is a C, K_h, K_w shaped window, representing each possible window of each image in the batch.



Next, we use the cartesian product transformer to pair every window of every image with every filter.

```

(windows
  (access activations 1)
  (shape 1 3 3 3 3)) ; ((N, 1, H', W', O), ())
  (access weights 1)) ; ((N, 1, H', W', O), (2, C, Kh, Kw))
1) ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2)) ; ((N), (C, H, W))

; ((O), (C, Kh, Kw))

```

230

Finally, we compute the dot product of each window-filter pair, which element wise multiplies each 3D window with each 3D filter and sums the results to a single scalar value.

This represents the core of the Glenside implementation of 2d convolution.

```

(windows
  (access activations 1)          ; ((N, O, H', W'), ())
  (shape 1 Sh Sw)                 ; ((N, 1, H', W', O), ())
  (access weights 1)))           ; ((N, 1, H', W', O), (2, C, Kh, Kw))
1)                               ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))                 ; ((N), (C, H, W))

                                ; ((O), (C, Kh, Kw))

```

231

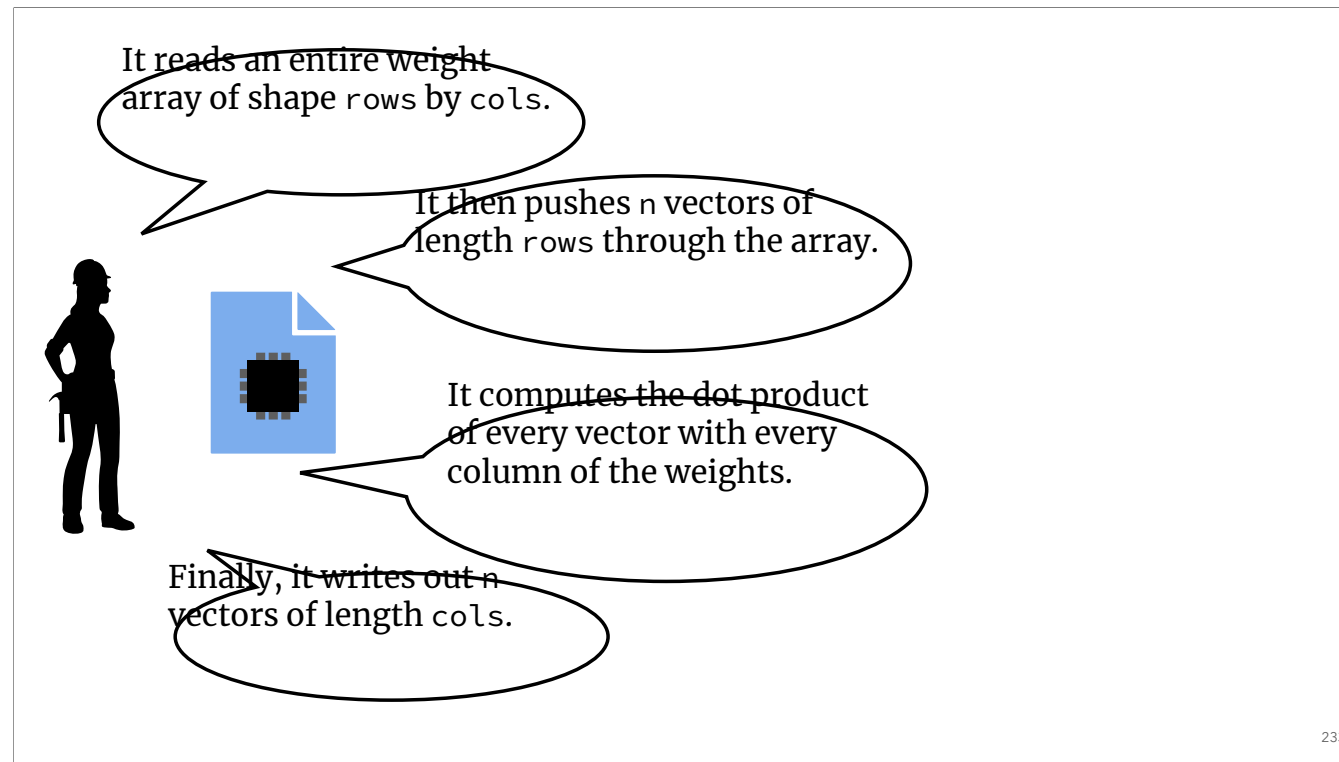
Optionally, we can also remove and rearrange some dimensions so that the output layout matches the input layout.

Outline

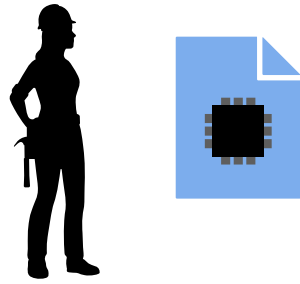
- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
 - Reimplementing Matrix Multiplication with Access Patterns
 - Implementing 2D Convolution with Access Patterns
 - Hardware Mapping as Program Rewriting
 - Flexible Hardware Mapping with Equality Saturation

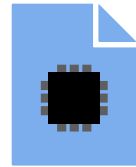
232

Now that we've shown that Glenside can represent common machine learning kernels, let's understand how it can be used for term rewriting. We begin by addressing our original goal.

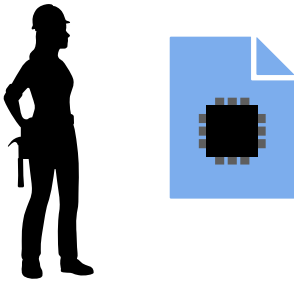


Recall our original idea: that hardware mapping is a program rewriting problem.
Can we turn our hardware designer's description of her accelerator into a searchable pattern?





Can we represent hardware
as a searchable pattern?



```
(compute dotProd
  (cartProd ?a0 ?a1))
where ?a0 is of shape
  ((?n), (?rows))
and ?a1 is of shape
  ((?cols), (?rows))
```

With Glenside, we can!

234

With Glenside, we can!

In Glenside, we can express her hardware as the following pattern, which matches a dot product mapped over the cartesian product of two access patterns `a0` and `a1`. Importantly, we can even express the fact that her hardware expects both access patterns to be vectors of vectors.

```
(compute dotProd
  (cartProd ?a0 ?a1))
  where ?a0 is of shape
    ((?n), (?rows))
  and ?a1 is of shape
    ((?cols), (?rows))
```

→

```
(systolicArray ?rows ?cols ?a0 ?a1)
```

We can directly rewrite to hardware invocations!

235

We can then directly rewrite this pattern to hardware invocations, represented here as a new systolic array expression. We can even convey hardware parameters such as the number of systolic array rows and columns.

```

((?n), (?rows))
and ?a1 is of shape
((?cols), (?rows))
      (compute-dotProd
        (cartProd
          (access A 1)
          (transpose
            (access B 1)
            (list 1 0))))
      (systolicArray ?rows ?cols ?a0 ?a1))

```

236

Now, we can use this rewrite to map our matrix multiplication implementation to her systolic array.

```

((?n), (?rows))
and ?a1 is of shape
((?cols), (?rows))
      (compute dotProd
      (cartProd
        (access A 1)
        (transpose
          (access B 1)
          (list 1 0))))
      (systolicArray ?rows ?cols ?a0 ?a1))

```

```
((?n), (?rows))  
and ?a1 is of shape  
((?cols), (?rows))
```

```
(systolic systolicArray ?rows ?cols ?a0 ?a1)
```

```
4 2
```

```
(access A 1)
```

```
(transpose
```

```
(access B 1)
```

```
(list 1 0))))
```

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
 - Reimplementing Matrix Multiplication with Access Patterns
 - Implementing 2D Convolution with Access Patterns
 - Hardware Mapping as Program Rewriting
 - Flexible Hardware Mapping with Equality Saturation

239

We have shown how we can use Glenside to formulate a program rewrite to perform hardware mapping. Yet mapping a matrix multiplication to matrix multiplication hardware is perhaps not all that impressive. To show the true power of program rewriting with Glenside, we will now demonstrate the *flexible* mapping of a 2D convolution to matrix multiplication hardware.

```
(windows  
  (access activations 1)  
  (shape C Kh Kw)  
  (shape 1 Sh Sw))  
  (access weights 1)))  
1)  
(list 0 3 1 2))  
  
(compute dotProd  
  (cartProd  
    (access A 1)  
    (transpose  
      (access B 1)  
      (list 1 0)))))
```

240

Looking at the Glenside implementations of 2d convolution—on the left—and matrix multiplication—on the right—we can see a remarkably similar structure.

<pre> (window (access activ (shape C Kh K (shape 1 Sh Sw)) (access weights 1))) 1) (list 0 3 1 2)) </pre>	<div style="background-color: red; color: white; padding: 5px; display: inline-block; transform: rotate(-15deg);"> Convolution and matrix multiplication have similar structure! </div>	<pre> (compute dotProd (cartProd (access A 1) (transpose (access B 1) (list 1 0)))) </pre>
---	---	--

241

Specifically, their core computation is the same: we take the cartesian product of two access patterns, and compute a dot product over the result. The primary difference is in how we form the input access patterns.

```

(window
  (access activations 1)
  (shape C Kh Kw)
  (shape 1 Sh Sw))
  (access weights 1)))
1)
(list 0 3 1 2))

```

```

(compute dotProd
  (cartProd ?a0 ?a1))

```

Can we apply our hardware rewrite? ape

```

((?cols), (?rows))
and ?a1 is of shape
((?cols), (?rows))

```

Knowing that 2D convolution is so similar to matrix multiplication, it begs the question: can we apply our systolic array rewrite?

```

(window
  (access activations 1)
  (shape C Kh Kw)
  (shape 1 Sh Sw))
  (access weights 1))
1) ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))

; ((O), (C, Kh, Kw))

```

(compute dotProd
 (cartProd ?a0 ?a1))
 where ?a0 is of shape
 ((?n), (?rows))
 and ?a1 is of shape
 ((?cols), (?rows))

Our access pattern shapes do not
pass the rewrite's conditions

As it stands, the answer is no. The systolic array rewrite expects access patterns which are vectors of vectors, while the access patterns present in 2D convolution are much more complicated.

```

(window
  (access activations 1)
  (shape C Kh Kw)
  (shape 1 Sh Sw))
  (access weights 1)))
1) ;((?n),(?rows))
(list 0 3 1 2))

```

```

(compute dotProd
  (cartProd ?a0 ?a1))
  where ?a0 is of shape
    ((?n), (?rows))
  and ?a1 is of shape
    ((?cols), (?rows))

```

```

;((?cols),(?rows))

```

Can we flatten our access patterns?

However, if we could somehow flatten the access patterns to be vectors of vectors, we could apply our rewrite. So the question is, can we flatten our access patterns?

`?a → (reshape (flatten ?a) ?shape)`

Flattens and immediately reshapes an access pattern

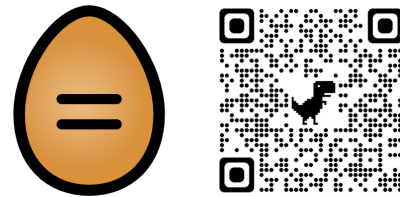
245

The answer is yes, and it begins with a very simple rewrite. This rewrite matches any access pattern, flattening and immediately reshaping the access pattern back to its original shape.

On its surface, it may seem impractical that this rewrite is applied to every possible access pattern. However, using the equality saturation program rewriting framework provided by the egg library, we can efficiently apply this rewrite in all possible locations and store all resulting programs.

`?a → (reshape (flatten ?a) ?shape)`

Flattens and immediately reshapes an access pattern



```

(window
  (access activations 1)
  (shape C Kh Kw)
  (shape 1 Sh Sw))
  (access weights 1)))
1) ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))

; ((O), (C, Kh, Kw))

```

246

When we apply this rewrite, one of the resulting programs is the following.

```

    (reshape (flatten (windows
      (access activations 1)
      (shape C Kh Kw)
      (shape 1 Sh Sw))) ?shape0)
    (reshape (flatten (access weights 1)) ?shape1)))
1) ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))

; ((O), (C, Kh, Kw))

```

247

Notice the reshape and flatten transformers on the two access patterns.

However, our access pattern shapes haven't changed, as even though the access patterns are flattened, they are then immediately reshaped back to their original shapes.


```

(reshape (flatten (windows
  (access activations 1)
  (shape C Kh Kw)
  (shape 1 Sh Sw))) ?shape0)
(reshape (flatten (access weights 0 1 2 3)
  1)
  (list 0 3 1 2))
; ((N, 1, H', W'), (C, Kh, Kw))
; ((O), (C, Kh, Kw))

```

But our access pattern shapes haven't changed!

```

    (reshape (flatten (windows
      (access activations 1)
      (shape C Kh Kw)
      (shape 1 Sh Sw))) ?shape0)
    (reshape (flatten (access weights 1)) ?shape1)))
  1) ; ((N, 1, H', W'), (C, Kh, Kw))
  (list 0 3 1 2))
  ; ((O), (C, Kh, Kw))

```

We need to "bubble" the reshapes to the top

248

To remedy this, we need to bubble the reshapes to the top.

These rewrites "bubble" reshape through cartProd and compute dotProd

```
(cartProd  
  (reshape ?a0 ?shape0)  
  (reshape ?a1 ?shape1)) → (reshape (cartProd ?a0 ?a1) ?newShape)
```

```
(compute dotProd  
  (reshape ?a ?shape)) → (reshape (compute dotProd ?a) ?newShape)
```

249

To do so, we can use these two rewrites, which express the composition commutativity of reshape with cartesian product and compute dot product. Note that these rewrites express general properties of Glenside, and are not specific to the task at hand.

```

    (reshape (flatten (windows
      (access activations 1)
      (shape C Kh Kw)
      (shape 1 Sh Sw))) ?shape0)
    (reshape (flatten (access weights 1)) ?shape1)))
1) ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))

; ((O), (C, Kh, Kw))

```

250

Once we apply these rewrites, our program looks like this.

```

(flatten (windows
  (access activations 1)
  (shape C Kh Kw)
  (shape 1 Sh Sw)))
(flatten (access
  1)
  ; ((N·1·H'·W'), (C·Kh·Kw))
(list 0 3 1 2))

; ((O), (C·Kh·Kw))

```

251

The reshapes have been moved out, and the access patterns are now flattened.

```

(flatten (windows
  (access activations 1)
  (shape C Kh Kw)
  (shape 1 Sh Sw)))
(flatten (access weights 1))) ?shape)
1) ; ((N·1·H'·W'), (C·Kh·Kw))
(list 0 3 1 2))

; ((O), (C·Kh·Kw))

```

((?n), (?rows))
 and ?a1 is of shape
 ((?cols), (?rows))

Our rewrite can now map
 convolution to matrix
 multiplication hardware!

We can now use our rewrite to map 2D convolution to our matrix multiplication hardware!

```
?a → (reshape (flatten ?a) ?shape)

(cartProd
 (reshape ?a0 ?shape0)
 (reshape ?a1 ?shape1)) → (reshape (cartProd ?a0 ?a1) ?newShape)

(compute dotProd
 (reshape ?a ?shape)) → (reshape (compute dotProd ?a) ?newShape)
```

These rewrites rediscover the im2col transformation!

253

This transformation is not new—this is what's called the im2col transformation. With a set of three simple rewrites expressing general properties of Glenside, we are able to rediscover im2col!

In conclusion,

In conclusion,
we have presented **access patterns** as a new tensor representation,

In conclusion,
we have presented **access patterns** as a new tensor representation,
we have used them to build the **pure, low-level, binder free IR Glenside**,

In conclusion,
we have presented **access patterns** as a new tensor representation,
we have used them to build the **pure, low-level, binder free IR Glenside**,
and have shown how they **enable hardware-level tensor program rewriting**.

<https://github.com/gussmith23/glenside>

Glenside is an actively maintained Rust library!
Try it out and open issues if you have questions!



255

Glenside is an actively maintained Rust library! Please try it out and feel free to open issues if you have questions or find bugs.



Lastly, I'd like to thank everyone who worked on this project, my labs, and the funding agencies that made it possible.

Thank you!

257

Thank you for listening, and I'm excited to take your questions!