Towards automated super-optimization for Taichi using Equality Saturation

Deyuan (Mike) He
Department of Computer Science, Princeton University
Term rewriting is extremely common in Compilers (example *manual* rewrites from *alg_simp.cpp*):

- $a \times 2 \rightarrow a \ll 1$
- $a / \text{pot} \rightarrow a \gg \log_2(\text{pot})$
- $a / \text{const} \rightarrow a \times (1 / \text{const})$
Background

However, determining the order of applying rewrite rules is HARD!

$M_1$ [N, M]  
$M_2$ [M, K]  
$M_3$ [K, P]  
$M_4$ [K, Q]
Background

1. Common Subexpression Elimination (CSE)
2. Associativity of Matrix Multiplication (Assoc)

\[
\begin{align*}
M_1 & \ast M_2 \ast M_3 \\
M_1 & \ast M_2 \ast M_4
\end{align*}
\]

Cost: $2NMK + NKP + NKQ$ multiplications
Background

CSE Rewrite

$M_1 \times M_2 \times M_3$

$M_1 \times M_2 \times M_4$

$M_{12} = M_1 \times M_2$

$M_{12} \times M_3$

$M_{12} \times M_4$

2NMK + NKP + NKQ multiplications

NMK + NKP + NKQ multiplications
Background

NMP + NMQ + MKP + MKQ multiplications

NMP + NMQ + MKP + MKQ multiplications

Assoc Rewrite

M1 * M2 * M3

M1 * M2 * M4

2NMK + NKP + NKQ multiplications
Background

Case 1 (Associativity better than CSE):
NMK + NKP + NKQ > NMP + NMQ + MKP + MKQ

e.g.
N = 2 M = 2 K = 8 P = 2 Q = 2

Before optimization: 128
⇒ CSE(96) > Assoc(80)
Background

Case 1 (CSE better than Associativity):
NMK + NKP + NKQ < NMP + NMQ + MKP + MKQ

e.g.
N = 2 M = 16 K = 4 P = 1 Q = 1

Before optimization: 544
⇒ CSE(144) < Assoc(192)
Compilers may have hundreds of passes.

How to determine the order to ensure the product program is Optimal (or close)?
Compilers may have hundreds of passes.

How to determine the order to ensure the product program is Optimal (or close)?

Interleaving Passes? Phase Ordering Problem?
Equality Saturation

Equality Saturation (EqSat) is a technique to solve this problem by memoizing all the equivalences discovered by rewrite rules.
Phase 1: Execute Rewrites

\[ M_1 \times M_2 \times M_3 \]
\[ M_1 \times M_2 \times M_4 \]

Associativity

\[ M_1 \times M_2 \times M_3 \]
\[ M_1 \times M_2 \times M_4 \]

\[ M_1 \times (M_2 \times M_3) \]
\[ M_1 \times (M_2 \times M_4) \]
Phase 1: Execute Rewrites

M₁ * M₂ * M₃
M₁ * M₂ * M₄

M₁ * (M₂ * M₃)
M₁ * (M₂ * M₄)

CSE

M₁ * M₂ * M₃
M₁ * M₂ * M₄

M₁ * (M₂ * M₃)
M₁ * (M₂ * M₄)

M₁₂ ⇐ M₁ * M₂
M₁₂ * M₃
M₁₂ * M₄
Phase 1: Execute Rewrites

Saturation: no more equivalence can be found by applying the rewrite rules (in *any* order)

This is the end of Phase 1
Phase 2: Extraction

Extraction: select the optimal term from the candidate set using a cost model

E.g.: number of multiplications
Phase 2: Extraction

Extraction: select the optimal term from the candidate set using a cost model

E.g.: number of multiplications

Efficient Implementations?
egg: Fast, Extensible Equality Saturation on EGraphs

1. E-Classes (dashed boxes): A set of equivalent terms
2. E-Nodes (solid boxes): Operators, variables or literals

- Represents the term “2”
- Represents the term “1+1”
Rewrite Rules

Syntactic Rewrites: an initial pattern and a target pattern

\[ ?x \times 2 \Rightarrow ?x \ll 1 \]

Apply

Binds to \(?x\)
Rewrite Rules

Syntactic Rewrites: an initial pattern and a target pattern

\[ ?x \times 2 \Rightarrow ?x \ll 1 \]

**Apply**

**Instantiate ?x \ll 1**

Binds to ?x

\[ a \times 2 + 1 \]
Rewrite Rules

Syntactic Rewrites: an initial pattern and a target pattern

\(?x \times 2 \Rightarrow ?x \ll 1\)

Apply

\text{rw!}("times-2-shift"; "(smult ?x 2) \Rightarrow (bitshl ?x 1)"),

Binds to \(?x\)

Instantiate \(?x \ll 1\)
E-Class Analysis

Rewrite rules are **syntactic**, meaning that it is not always valid in terms of **semantics**

\[ ?x / ?x \Rightarrow 1 \quad \text{if } ?x \text{ does not evaluate to 0} \]

\[ \text{pow}(2, ?x) \Rightarrow 1 << ?x \quad \text{if } ?x \text{ is an integer} \]

\[ d(?c) \Rightarrow 0 \quad \text{if } ?c \text{ is a constant} \]
E-Class Analysis

E-Class Analysis: fully-customizable program analysis data attached to EClasses. E.g. Type checking / inference

Checked_type: f32

\[ \times \]

Checked_type: f32

\[ a \]

2

+ 

1

Checked_type: i32

Checked_type: i32

Bottom-up initialization: parents (E-Node) have access to children (E-Classes) analyses.
E-Class Analysis enables conditional rewrites

\[ ?v \times 2 \Rightarrow ?v << 1 \quad \text{if} \ is\_integer(?v) \]

is_integer(?v): Checks the E-Class analysis matched to ?v whether it holds an integral value.
E-Class Analysis enables conditional rewrites

\[
?v \times 2 \Rightarrow ?v \ll 1 \quad \text{if } is\_integer(?v)
\]

is_integer(?v): Checks the E-Class analysis matched to ?v whether it holds an integral value.

Since the E-Class matched to ?v has Checked_type: f32, this rewrite rule won't be fired.
E-Class Analysis

E-Class Analysis enables conditional rewrites

```rust
fn is_integer(x: Var) → impl Fn(&mut EGraph, egg::Id, &egg::Subst) → bool {
    move |egraph: &mut EGraph<ChiIR, ChiAnalysis>, _id: Id, subst: &Subst| match &egraph[subst[x]].data.analysis_info {
        AnalysisInfo::DType(dt: &DataType) ⇒ match dt {
            DataType::Int(_) | DataType::UInt(_) ⇒ true,
            _ ⇒ false,
        },
        _ ⇒ false,
    }
}
```

Since the E-Class matched to ?v has Checked_type: f32, this rewrite rule won’t be fired.
Benefits behind

1. Observation: even there are rules that keep the EGraph from saturating\(^1\), we are able to explore a large space of equivalences efficiently and automatically

2. Verifying individual rule guarantees soundness of their compositions\(^2\)
3. Lower the difficulty of contributing optimization rewrites
4. Enable facilitating new backend by adding tiling / offloading rewrites

---

1: This is the case for most applications because of expansive rules, e.g. \(?x \Rightarrow \text{transpose}(\text{transpose}(?x))\)
2: we are focusing on functional rewrites so far
Extraction in egg

Extraction: Given a root E-Class, pick the “best” term (minimizing the sum of costs of E-Nodes given by a cost model)

Numbers in E-Nodes are example costs
Extraction in egg

Extraction: Given a root E-Class, pick the “best” term (minimizing the sum of costs of E-Nodes given by a cost model)

Numbers in E-Nodes are example costs
Extraction in egg

Implementations
- Greedy: Pick the minimum one at each level
  - 😊 Easy to implement
  - 😞 Don’t know about CSE (sharing)
- Integer Linear Programming (ILP)
  - 😊 Sound minimum
  - 😞 Timeout; does not work well with cycles
A CHI IR subset in egg

Thanks to egg’s extensibility, we are able to encode a (functional) *subset* of CHI IR in egg

```
"sadd" = SAdd([Id; 2]),
"sminus" = SMinus([Id; 2]),
"smult" = SMult([Id; 2]),
"sdiv" = SDiv([Id; 2]),
"smod" = SMod([Id; 2]),

"gt" = Gte([Id; 2]),
"lte" = Lte([Id; 2]),
"gt" = Gt([Id; 2]),
"le" = Lt([Id; 2]),
"eq" = Equals([Id; 2]),

"land" = LAnd([Id; 2]),
"lor" = LOr([Id; 2]),
"lnot" = LNot([Id; 1]),
"lxor" = LXor([Id; 2]),
```

We mostly focus on matrices: major workload

Full language definition is available here: [https://github.com/AD1024/egg-taichi/blob/main/src/language.rs](https://github.com/AD1024/egg-taichi/blob/main/src/language.rs)
CHIAnalysis

DataType Analysis: DType of the expression

Constant Info: Option<ConstData>; whether the expression yields a constant

Merging

```
promote_dtype(DType₁, DType₂)
pick_compare(Const₁, Const₂)
```

promote_dtype follows taichi’s typing rule
pick_compare chooses a Some value; if both are Some variant, then compare them
Rewrites examples

**Scalar Rewrites**

\[(\text{sadd } ?x \ ?y) \Rightarrow (\text{sadd } ?y \ ?x)\]

\[(\text{smult } (\text{sadd } ?x \ ?y) \ ?z) \Rightarrow (\text{sadd } (\text{smult } ?x \ ?z) (\text{smult } ?y \ ?z))\]

\[(\text{pow } 2 \ ?x) \Rightarrow (\text{bitshl } 1 \ ?x) \quad \text{if } \text{is_integer}(?x)\]

**Matrix/Vector Rewrites**

\[(\text{transpose } (\text{transpose } ?x)) \Rightarrow ?x\]

\[(\text{transpose } (\text{add } ?x \ ?y)) \Rightarrow (\text{add } (\text{transpose } ?x) (\text{transpose } ?y))\]

\[(\text{matmul } ?x (\text{matmul } ?y \ ?z)) \Rightarrow (\text{matmul } (\text{matmul } ?x \ ?y) \ ?z))\]
More Rewrites

Customized rewrites: Constant folding
(bear me with not using a macro for these :p)

```r
rw!("const-fold-add"; "(sadd ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "+".to_string() } } ),
```

```r
rw!("const-fold-mult"; "(smult ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "*".to_string() } } ),
```

```r
rw!("const-fold-div"; "(sdiv ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "/".to_string() } } ),
```

```r
rw!("const-fold-sub"; "(sminus ?x ?y)" => { BinopConstFoldApplier { lhs: "?x".parse().unwrap(), rhs: "?y".parse().unwrap(), op: "-".to_string() } } ),
```
More Rewrites

```rust
impl Applier<ChiIR, ChiAnalysis> for BinopConstFoldApplier {
    fn apply_one(
        &self,
        egraph: &mut egg::EGraph<ChiIR, ChiAnalysis>,
        eclass: egg::Id,
        subst: &egg::Subst,
        _: Option<egg::PatternAst<ChiIR>>,
        _: egg::Symbol,
    ) -> Vec<egg::Id> {
        if let (Some(c1: ConstData), Some(c2: ConstData)) = (  
            ChiAnalysis::get_constant(egraph, id: &subst[self.lhs]),  
            ChiAnalysis::get_constant(egraph, id: &subst[self.rhs]),  
        ) {
    
```

Enables us to check & use analysis data, and then fire a customized rewritten term.

E.g.: if we are folding +, then the resulted term is a constant equal to the sum of two constant data in the E-Class analysis.

Full implementation:
https://github.com/AD1024/egg-taichi/blob/dd5c370395662c55b8d77c3ab601a365219835ce/src/rewrites.rs#L34-L85
Cost Model

For proof-of-concept prototype, we implement a simple cost model

For scalar operations, we use an “estimated” CPU cycle count;
For matrix operations, we use the number of vector dots.

In the future, we will take vectorized instruction into consideration.
Probably use a more precise approach: profiling on the machine running the optimizer.

Implementation:
https://github.com/AD1024/egg-taichi/blob/main/src/extraction.rs
Preliminary Results

We set the constant N to 16

```python
@ti.kernel
def init_mesh():
    for i, j in ti.ndrange(N, N):
        k = (i * N + j) * 2
        a = i * (N + 1) + j
        b = a + 1
        c = a + N + 2
        d = a + N + 1
        f2v[k + 0] = [a, b, c]
        f2v[k + 1] = [c, d, a]
```

\[(\text{cons} \ (\text{smult} \ (\text{sadd} \ (\text{smult} \ i \ N) \ j) \ 2)) \]
\[(\text{cons} \ (\text{sadd} \ j \ (\text{smult} \ i \ (\text{sadd} \ N \ 1)))) \]
\[(\text{cons} \ (\text{sadd} \ 1 \ (\text{sadd} \ j \ (\text{smult} \ i \ (\text{sadd} \ N \ 1)))))) \]
\[(\text{sadd} \ N \ (\text{sadd} \ 2 \ (\text{sadd} \ j \ (\text{smult} \ i \ (\text{sadd} \ N \ 1)))))))))\]

Cost before optimization : 92
We set the constant N to 16
Preliminary Results

A Simple matrix multiplications / element-wise additions (MLP)

16×16 → Linear 16×32 → Linear 32×64 → Linear 64×10
Preliminary Results

A Simple matrix multiplications / element-wise additions (MLP)

\[
\begin{align*}
&\text{(ewadd} \\
&\quad \text{(matmul} \\
&\quad\quad \text{(ewadd} \\
&\quad\quad\quad \text{(matmul input W1)} \\
&\quad\quad\quad\quad \text{(ewadd} \\
&\quad\quad\quad\quad\quad \text{(matmul input W1)} \\
&\quad\quad\quad\quad\quad\quad \text{b1)} \\
&\quad\quad\quad\quad\quad\quad\quad \text{W2)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad \text{b1)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{W3)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{b2)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{W3)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{b3)} \\
&\text{Cost before optimization: 51361}
\end{align*}
\]

\[
\begin{align*}
&\text{(ewadd} \\
&\quad \text{(ewadd} \\
&\quad\quad \text{(matmul} \\
&\quad\quad\quad \text{(ewadd} \\
&\quad\quad\quad\quad \text{(matmul} \\
&\quad\quad\quad\quad\quad \text{(ewadd} \\
&\quad\quad\quad\quad\quad\quad \text{(matmul} \\
&\quad\quad\quad\quad\quad\quad\quad \text{input W1)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad \text{b1)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{W2 W3)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{b2 W3)} \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{b3)} \\
&\text{Cost after optimization: 44140}
\end{align*}
\]
Discussion

1. egg only works well with data-flow based IR, but CHI IR has control flow operators
   a. Encode Loops in terms of mathematical functions (Tate et al.)
   b. Conversion from/to CFG
2. Global effects are hard to handle in egg’s representation
   a. Focus on pure functions / procedures first
   b. Proper effect handling transformations before converting into egg
3. Matrix operations representations in CHI IR
Q & A