SmoothE: Differentiable E-Graph Extraction

Yaohui Cai
yc2632@cornell.edu
Cornell University
Ithaca, New York, USA

Kaixin Yang
ky427@cornell.edu
Cornell University
Ithaca, New York, USA

Chenhui Deng
cd574@cornell.edu
Cornell University
Ithaca, New York, USA

Cunxi Yu
cunxiyu@umd.edu
University of Maryland, College Park
College Park, Maryland, USA

Zhiru Zhang
zhiruz@cornell.edu
Cornell University
Ithaca, New York, USA

Abstract

E-graphs have gained increasing popularity in compiler optimization, program synthesis, and theorem proving tasks. They enable compact representation of many equivalent expressions and facilitate transformations via rewrite rules without phase ordering limitations. A major benefit of using e-graphs is the ability to explore a large space of equivalent expressions, allowing the extraction of an expression that best meets certain optimization objectives (or cost models). However, current e-graph extraction methods often face unfavorable scalability-quality trade-offs and only support simple linear cost functions, limiting their applicability to more realistic optimization problems.

In this work, we propose *SmoothE*, a differentiable e-graph extraction algorithm designed to handle complex cost models and optimized for GPU acceleration. More specifically, we approach the e-graph extraction problem from a probabilistic perspective, where the original discrete optimization is relaxed to a continuous differentiable form. This formulation supports any differentiable cost functions and enables efficient searching for solutions using gradient descent. We implement SmoothE in PyTorch to leverage the advancements of the modern machine learning ecosystem. Additionally, we introduce performance optimization techniques to exploit sparsity and data parallelism. We evaluate SmoothE on a variety of realistic e-graphs from five different applications using three distinct cost models, including both linear and non-linear ones. Our experiments demonstrate that SmoothE consistently achieves a favorable trade-off between scalability and solution quality.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707262

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → *Compilers*; **General programming languages**.

Keywords: Machine learning for systems; Compilers; Programming languages; Equivalence graph

ACM Reference Format:
Yaohui Cai, Kaixin Yang, Chenhui Deng, Cunxi Yu, and Zhiru Zhang. 2025. SmoothE: Differentiable E-Graph Extraction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707262

1 Introduction

Term rewriting [17], widely employed in compiler optimizations [8, 34, 44] and theorem proving [15, 18], transforms programs into functionally equivalent but more efficient forms. Traditional methods apply the rewrites sequentially in a predetermined order, significantly affecting performance—a challenge known as the *phase ordering* problem [44, 50].

Equality saturation addresses the phase ordering issue by using the *equivalence graph (e-graph)*, a data structure that compactly represents a set of expressions (i.e., *e-nodes*) and their equivalence relations (i.e., *e-classes*) [6, 33]. The rewrite rules are applied collectively, encoding all functionally equivalent solutions on a single e-graph. This enables the selection of the most cost-efficient (or performant) one during the *e-graph extraction* process. With the emergence of state-of-the-art open-source equality saturation tools such as `egg` [50] and `egglog` [56], e-graph has been successfully used for tensor graph transformation [53], sparse linear algebra optimization [49], code optimization [29, 41], digital signal processor (DSP) compilation [45, 48], circuit datapath synthesis [10, 12–14, 47], and floating-point arithmetic [11, 35].

Extracting a high-quality solution from an e-graph is challenging due to theoretical complexity (proven to be NP-hard [42, 55] in general) and practical efficiency, particularly given the typically large graph size. While a number of exact and heuristic e-graph extraction methods have been

proposed, existing approaches still have significant limitations that restrict their applicability to even larger and more realistic optimization problems:

(1) **Unfavorable scalability-quality trade-off:** Most existing work uses integer linear programming (ILP) formulations for e-graph extraction [10, 12, 14, 41, 47, 49, 53], which suffer from limited scalability. Conversely, heuristic algorithms like greedy and iterative methods [35] may yield sub-optimal results. Additionally, current approaches are primarily designed for CPU execution and cannot leverage modern parallel computing devices like GPUs.

(2) **Inadequate support of realistic cost models:** Current e-graph extraction algorithms rely on linear cost models during optimization, where the overall cost is a weighted sum of individual term costs. This simplified model may fail to capture the complexities of real-world optimization problems. Additionally, these algorithms cannot be easily extended to incorporate data-driven, learned cost models.

To address these challenges, we propose *SmoothE*, a differentiable e-graph extraction framework. More specifically, we employ a probabilistic approach to the extraction problem by transforming the binary decisions of e-node selection into continuous variables that represent selection probabilities. This transformation allows us to leverage gradient descent for optimization, creating an efficient and highly parallelizable method. Our approach further enables the optimization of e-node selection probabilities in parallel, significantly accelerating the process on devices like GPUs. After continuous optimization, we perform sampling to convert these optimized probabilities into discrete selections, ensuring that all constraints are met for a valid e-graph extraction.

SmoothE distinguishes itself from existing extraction methods in several key aspects. First, it provides a differentiable formulation that enables scalable global optimization on parallel computing resources, such as GPUs. Second, the proposed optimization technique can incorporate complex non-linear cost models, including those based on machine learning (ML), which allows it to capture intricate dependencies and interactions that simpler models cannot. Third, SmoothE can seamlessly integrates with existing ML frameworks like PyTorch, ensuring rapid and practical implementation. We believe this flexibility and scalability make SmoothE a powerful tool for e-graph extraction, capable of achieving high-quality solutions with fast execution times.

The major contributions of this paper are as follows:

- We present SmoothE, the first differentiable approach to e-graph extraction. SmoothE relaxes the discrete optimization problem of e-node selection into a continuous, probabilistic formulation, enabling parallelizable global optimization that supports any differentiable objectives.
- We implement SmoothE in PyTorch and further introduce a set of performance optimization techniques to exploit vectorized processing and sparsity to improve the efficiency of GPU execution. We also introduce seed batching, which runs a batch of optimizations with different random seeds in parallel, improving both solution quality and GPU utilization.
- We present a comprehensive comparative study of existing e-graph extraction methods, including ILP-based formulations, iterative heuristic methods, and our approach, on five different real-world datasets using the conventional linear cost model. Our results demonstrate that SmoothE achieves a similar level of solution quality to ILP baselines, with second-scale execution time on GPUs.
- We demonstrate that, unlike existing e-graph extraction methods, SmoothE can flexibly handle non-linear cost models, including learning-based ones, while maintaining high solution quality.

The rest of the paper is organized as follows: Section 2 provides background on e-graphs and existing extraction methods. Section 3 introduces our SmoothE differentiable extraction approach. Section 4 details SmoothE implementation and GPU optimization. Section 5 evaluates various extraction methods on a set of real-world and synthetic datasets. We conclude in Section 6.

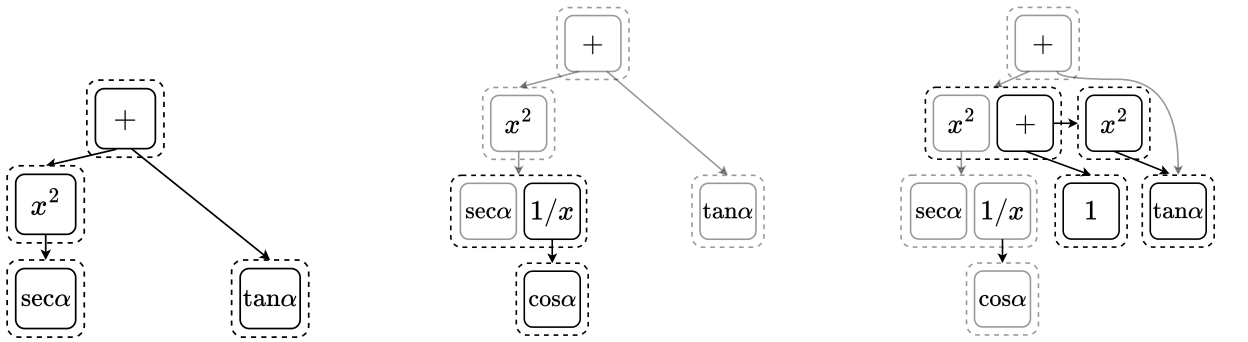
2 Preliminaries

E-graph is an extended union-find [43] data structure compactly representing many equivalent terms. Originally developed for automated theorem provers (ATPs) [15, 32], e-graphs haven been recently popularized by egg [50], which provides a faster and flexible implementation for equality saturation.

In an e-graph, all functionally equivalent terms are organized in the same equivalent classes, known as **e-classes**. Nodes within each e-class that represent values or operators are called **e-nodes**. Edges in e-graphs are directed, which point from e-nodes to their child e-classes, indicating the dependency between operators and operands.

With **equality saturation**, an input program and a set of rewrites are given. First, an initial e-graph is created from the input. To apply rewrites, the patterns on the left-hand side of the rewrites are matched and the terms on the right-hand side are added. Note that the process is usually additive, meaning the left-hand side is still represented in the e-graph. When the rewriting saturates, the e-graph has encoded every possible equivalent programs based on the given rewrites. The primary benefit of using e-graphs is the ability to encode and explore a large space of equivalent terms, allowing the extraction of a term that best meets certain optimization objectives without phase ordering limitations.

Figure 2 shows a concrete example of e-graphs and the equality saturation process. The initial e-graph in Figure 1a represents a simple term $\sec^2 \alpha + \tan \alpha$. By applying rewrite rule $\sec \alpha \longrightarrow 1/\cos \alpha$, the e-graph expands the e-class representing $\sec \alpha$ to include $1/\cos \alpha$, as illustrated in Figure 1b.



(a) Initial e-graph contains $\sec^2\alpha + \tan\alpha$ **(b)** After applying $\sec\alpha \rightarrow 1/\cos\alpha$ **(c)** After applying $\sec^2\alpha \rightarrow 1 + \tan^2\alpha$

Figure 1. An e-graph example with two rewrites — α is the primary data input to the program; x^2 and $1/x$ denote a square and reciprocal function, respectively; e-classes are dashed boxes, each containing e-nodes as solid boxes; edges originate from e-nodes to their children e-classes, indicating dependencies; the e-class containing + is the root; modifications by rewrites are highlighted in black.

Then $\sec^2\alpha \rightarrow 1 + \tan^2\alpha$ is applied, which expands the e-class representing $\sec^2\alpha$ as shown in Figure 1c.

E-graph extraction aims to to extract an optimized term from an e-graph after rewrites, based on a user-defined cost model. The goal is to produce a functionally equivalent but improved implementation of the original input program. The e-graph extraction problem is proven to be NP-hard when common sub-expressions are considered [42, 55]. Both exact optimization methods based on ILP and heuristic algorithms have been proposed, offering different trade-offs between solution quality and scalability. Next, we will provide a more formal statement of the e-graph extraction problem, before describing the popular optimization methods.

Notations and Problem Formulation. We first introduce the notations used throughout the rest of the paper.

- Let $\{n_i\}_{i=0}^{N-1}$ denote the set of all e-nodes in the e-graph.
- Let $\{m_j\}_{j=0}^{M-1}$ denote the set of all e-classes, where the root e-class containing the top-level operator is indexed by 0.
- Each e-class m_j contains a set of e-nodes: $m_j = \{n_k\}$ and $|m_j|$ denotes the cardinality of this set.
- ch_i denotes the set of child e-classes for e-node of index i.
- pa_j denotes the set of parent e-nodes that depend on e-class of index j.
- $ec(i)$ returns the index of the e-class that contains e-node of index i, namely, $n_i \in m_{ec(i)}$.
- Let $s \in \{0,1\}^N$ represent the binary decision variable for e-node selection: $s_i = 1$ means that the i-th e-node is selected after extraction, and $s_i = 0$ indicates otherwise.
- $f_b(\cdot) : \{0,1\}^N \rightarrow \mathbb{R}$ denotes a function that translates the e-node selection into a real-valued cost. Here the subscription $_b$ means the input vector is assumed to be binary.
- We call a cost function linear if and only if it can be written in the following form: $f_b(s) = u^T s, u \in \mathbb{R}^N$, where u_n is the cost associated with e-node n.

To ensure the extracted program is functionally equivalent to the input, the following constraints must be satisfied [44]:

(a) Exactly one e-node is selected from the root e-class.
(b) If an e-node is selected, exactly one e-node must be selected from each of its child e-classes.
(c) No cycles are included in the extracted e-graph.

Constraint (b) ensures that if an operator e-node is selected, all its the operand e-classes are also present in the extracted graph. Together with constraint (a), these ensure the extracted program is functionally equivalent to the input program. In this paper, we call them ***completeness constraints***. A cycle within an e-graph would imply that an operand of an operator eventually depends on the operator itself, creating a circular dependency. In many cases, this would lead to an infinite loop during evaluation, rendering the program incorrect or non-terminating. Thus the ***acyclicity constraint*** is required in (c). Next, we introduce two most popular e-graph extraction methods: ILP and heuristic algorithm.

ILP or mixed-integer linear programming (MILP) formulations have been used for a number of e-graph extraction tasks [12, 40, 47, 49, 53]. A representative form [26] is introduced by TenSat [53], which formulates the e-graph extraction problem as follows:

$$\text{minimize}_{s\in\{0,1\}^N} f_b(s) = u^T s \tag{1a}$$

$$\text{s.t.} \sum_{n_k \in m_0} s_k = 1 \tag{1b}$$

$$\forall i, \forall j \in ch_i, s_i \leq \sum_{n_k \in m_j} s_k \tag{1c}$$

$$\forall i, s_i \in \{0,1\}^N \tag{1d}$$

$$\forall i, \forall j \in ch_i, t_{ec(i)} - t_j - \epsilon + A(1 - s_i) \geq 0 \tag{1e}$$

$$\forall j, t_j \in [0,1]^M \tag{1f}$$

Here s_i means whether e-node n_i is selected or not; t_j encodes the order of e-classes being selected; $\epsilon < 1/M$ is a small constant and $A > 1+\epsilon$ is a large constant. Eq. (1b), (1c), and (1d) correspond to completeness constraints. Eq. (1e) and (1f) enforce a topological order on all the chosen e-classes, ensuring no cycles exist within the extracted e-graph.

While the ILP approach is generally flexible and produces an optimal solution if the solver terminates before timing out, it does not scale well on large e-graphs. Even state-of-the-art commercial solvers can easily take hours to solve medium-sized extraction problems.

Heuristic Methods. A popular heuristic method for fast e-graph extraction involves updating estimated costs on e-classes and selecting e-nodes iteratively using a queue-based worklist [35]. The algorithm begins by initializing the cost of all e-classes to infinity. The goal is to minimize the cost for the root e-class. An aggregated cost of selecting an e-node n_i is defined as the sum of its predefined individual cost and the costs of all its child e-classes in ch_i. Since all e-nodes within an e-class are functionally equivalent, the cost associated with the e-class is determined by the minimum aggregated cost of selecting any e-node within it. To get the cost of the root e-class, a working queue is initialized with all leaf e-nodes (those without child e-classes). When an e-node n_i is dequeued, we update its aggregated cost. If the new cost is lower than the current cost of its belonging e-class $ec(i)$, we update the cost of the e-class as well as the selected e-node within it accordingly. The parent e-nodes $pa_{ec(i)}$ are then added to the queue. This process repeats until the queue is empty. This algorithm is sometimes referred to as a greedy approach in some existing implementations.

A major limitation of the heuristic algorithm tends to overestimate costs by ignoring the reuse of common subexpressions, resulting in suboptimal solutions. Figure 2 illustrates an example where a common subexpression impacts the performance of the heuristic algorithm, where the $\tan\alpha$ can be reused by two parent e-nodes to save the cost. However, the heuristic algorithm (Figure 2b) fails to identify this reuse opportunity resulting in sub-optimal solution.

Other Extraction Methods. Several task-specific e-graph extraction methods have been proposed, e.g., NSGA-II [16], a genetic algorithm, is introduced in HL-HELM [52]. The authors of [26] propose a MAXSAT-based formulation for e-graph extraction. However, it requires enumerating all the cycles within the e-graph, whose number grows exponentially with the number of e-nodes. Tensat [53] prunes e-graphs by removing all cycles as a preprocessing step, allowing the acyclicity constraint to be ignored and significantly reducing the time required by ILP. Babble [5] similarly prunes candidate patterns using approximations. However, such preprocessing reduces the feasible solution space, potentially compromising the quality of the final solution.

Limitations of Linear Cost Models. Current e-graph extraction methods predominantly assume linear cost models for their simplicity, where the overall cost is a weighted sum of the per-e-node cost estimates. For some tasks, costs such as resource usage can be reasonably approximated using an additive linear function. However, linear models cannot capture the higher-order interactions at the sub-graph level among different e-nodes. Therefore, there are many scenarios where using a non-linear cost model would be beneficial. For instance, the recent work E-syn [7] uses e-graph to improve technology mapping-aware logic synthesis. Since a linear cost function cannot capture the clustering effects of multiple operations during technology mapping, E-syn generates a pool of extraction candidates and uses an ML model based on XGBoost [9] to rank the solutions. Another example is ROVER [14], which relies on RTL simulation to obtain toggle rates for optimizing power consumption of arithmetic circuits during the e-graph extraction process. In this case, a non-linear proxy cost model [54, 58] could significantly speed up the optimization.

3 A Differentiable Approach to E-Graph Extraction

In this section, we describe how we formulate the e-graph extraction problem in a differentiable manner by transforming the original discrete optimization into a continuous and probabilistic process, which we call SmoothE. We break down the original discrete optimization problem into four key components: 1) variables for e-node selection, 2) objective function, 3) completeness constraints, and 4) acyclicity constraint, and transform each component separately. In the following, we first discuss how each component is transformed (Section 3.1 through Section 3.4). Afterwards, in Section 3.5, we describe the optimization of the continuous problem and how we obtain the discrete solutions.

3.1 Continuous Variables for E-Node Selection

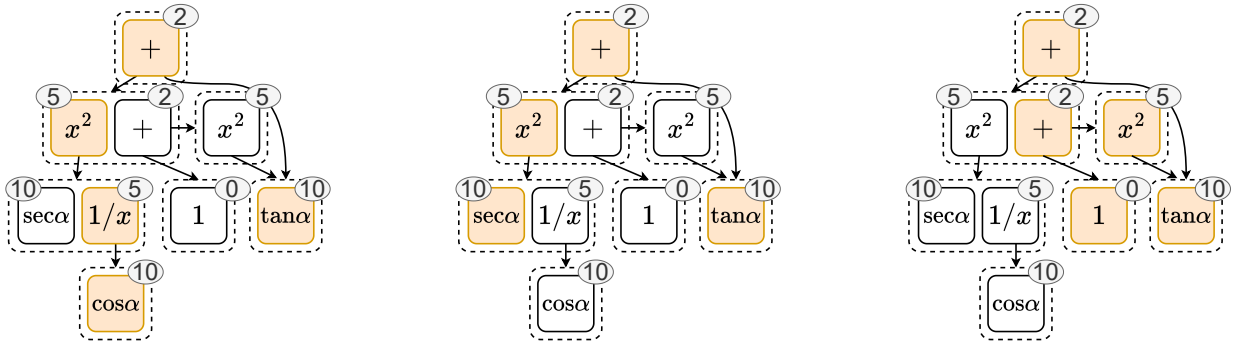
We first relax the variable to be optimized (i.e., s) to be a continuous variable and interpret its value as the probability of e-node being selected in the extracted graph, namely,

$$p_i \coloneqq P(n_i \text{ is chosen}), \tag{2}$$

where $p \in [0,1]^N$ is a real-valued probability vector, and is the variable to be optimized in the relaxed form. To obtain the discrete solution variable s from p, we use a sample process, which will be introduced in Section 3.5.

3.2 Differentiable Objective Function

The binary cost function $f_b(s)$ introduced in the previous section is non-differentiable, as it takes a binary vector as input. In SmoothE, we assume that the cost function (or optimization objective) is relaxed into a *differentiable* form $f(\cdot)$ with continuous input $[0,1]^N$. Note that in this case, the linear cost $f(p) = u^T p$ is naturally differentiable.



(a) Solution 1: $\tan\alpha + (1/\cos\alpha)^2$, total cost = 2 + 5 + 5 + 10 + 10 = 32

(b) Solution 2: (heuristic): $\sec^2\alpha + \tan\alpha$, total cost = 2 + 5 + 10 + 10 = 27

(c) Solution 3: (optimal): $\tan\alpha + 1 + \tan^2\alpha$, total cost = 2 + 2 + 0 + 5 + 10 = 19

Figure 2. Three extraction solutions for e-graph in Figure 2 — the colored boxes indicate the selected e-nodes after extraction; The circled numbers are the costs associated with each e-node; $\tan\alpha$ is a common subexpression that can be reused by $1 + \tan^2\alpha$ and $\tan\alpha$; Figure 2b shows that the heuristic method fails to reuse this term, resulting in a sub-optimal solution.

3.3 Handling Completeness Constraints

We first handle the completeness constraints (a) and (b) introduced in Section 2. For constraint (a), we use the same form as Eq. (1b) in the ILP formulation, which ensures the probabilities of choosing any e-nodes in the root e-class sum up to 1. For constraint (b), we also adopt the same form as Eq. (1c), which enforces the probability of choosing an e-node n to be smaller than the sum of the probabilities of e-nodes in its child e-classes. These constraints only specify the sum probabilities of e-nodes within each e-class. However, how to assign these probabilities remains unknown. Furthermore, the assignment needs to be differentiable.

To tackle this problem, we introduce an intermediate variable cp to assign probabilities of e-nodes within each e-class. cp represents the conditional probability of choosing e-node n_i given that its belonging e-class $m_{ec(i)}$ is selected:

$$cp_i \coloneqq P(n_i \text{ is chosen} \mid m_{ec(i)} \text{ is chosen}) = \frac{p_i}{P(m_{ec(i)} \text{ is chosen})} \tag{3a}$$

$$cp_k \in [0,1], \quad \sum_{n_k \in m_{ec(i)}} cp_k = 1, \tag{3b}$$

Eq. (3b) imposes necessary constraints on the conditional probability cp: it must be a real number between 0 and 1, and cp of all e-nodes from the same e-class sum up to 1. We use a softmax function to ensure Eq. (3b) in practice.

Then we change the variable to be optimized from p to cp and use a differentiable function $\phi : cp \mapsto p$ to compute probabilities p according to conditional probabilities cp. After that, the optimization problem changes to the following:

$$\text{minimize}_{cp \in [0,1]^N} f(p), \tag{4a}$$

$$\text{s.t. } p = \phi(cp), \tag{4b}$$

$$\text{acyclicity constraint}, \tag{4c}$$

Next we will discuss how to compute function ϕ differentiably. Computing ϕ exactly on a cyclic graph can be done with the Junction Tree algorithm [36]. However, it has a exponential time complexity of $O(2^k)$, where k is the size of the largest clique, making it not scalable for large problems. Thus loopy belief propagation (LBP) [28, 31], as an approximate method, is often used to compute such ϕ in graphical models with cycles. For acyclic graphs, LBP is guaranteed to converge to the exact value in a linear time complexity. However, for cyclic graphs, convergence is theoretically assured only under specific conditions [27]. Despite this, it has been observed that LBP typically converges effectively in practice [31]. LBP iteratively updates the beliefs of the target node based on the conditional probabilities of its neighboring nodes; these beliefs are then used to compute marginal probabilities. Thus we adapt BP to the e-graph setting for computing ϕ.

More concretely, by rewriting Eq. (3a), we have:

$$p_i = cp_i \cdot P(m_{ec(i)} \text{ is chosen}), \tag{5}$$

which means computing ϕ is essentially equivalent to calculating $P(m_{ec(i)} \text{ is chosen})$. However, the probability of choosing an e-class (i.e., $m_{ec(i)}$) depends on its parent e-nodes and their correlations, which can be very complex. Therefore, we make an assumption about the correlations among the parent e-nodes of any e-class. First, we discuss the assumption that all parent e-nodes are independent. Subsequently, we will consider alternative assumptions.

With the independence assumption, we have:

$$P(m_{ec(i)} \text{ is chosen}) = 1 - \prod_{n_k \in pa_{ec(i)}} P(n_k \text{ is not chosen}) \tag{6a}$$

$$= 1 - \prod_{n_k \in pa_{ec(i)}} (1 - p_k), \tag{6b}$$

which requires knowing the probability of all the parent e-nodes. In e-graphs without cycles, we can compute Eq. (5) and (6) following a topological order. This means before computing the probability of an e-class, probabilities of all the parent e-nodes are computed first.

However, for e-graphs containing cycles, circular dependencies prevent topological ordering, making the above sequential update impossible. To tackle this problem, we propose a parallel schedule: first, the probabilities of all e-classes are initialized to 0, except for the root e-class, which is set to 1 since it is always selected. Then by applying Eq. (5) and (6) to all e-nodes and e-classes in parallel, we can propagate the probability from the root e-class to the whole e-graph without an explicit order. The process repeats until convergence of the probability values.

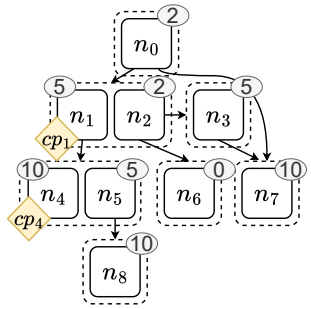
When probabilities converge, Eq. (5) and (6) will hold for all e-nodes, indicating that the completeness constraints are met. While parallel schedule imposes greater computational demands, it is friendly for GPU acceleration by eliminating compute dependencies and leveraging parallel computing units through vectorization (to be discussed in Section 4.2).

In contrast to independence assumption, an alternative option is to assume the parent e-nodes of $m_{ec(n)}$ are fully (positively) correlated — if the e-node with the highest probability is not chosen, none of other e-nodes will not chosen either. Therefore, we only consider the parent e-node with the highest probability:

$$P(m_{ec(i)} \text{ is chosen}) = \max_{k \in pa_{ec(i)}} p_k, \tag{7}$$

Alternatively, we can also use a hybrid assumption — a case in between of independence and fully positively correlated assumption. Then the probability of e-class $m_{ec(i)}$ is computed as the arithmetic average of the results from two previous assumptions. No matter which assumption is adopted, this whole process is differentiable as every step within it is differentiable.

Figure 3 illustrates a concrete example how to compute probabilities ϕ. In this example, there are only two e-classes containing more than one e-node: the e-class containing e-nodes n_1 and n_2, and the one containing e-nodes n_4 and n_5. For each of these e-classes, we only need to solve cp for one of the two e-nodes since their sum always equals 1. Assuming we pick cp_1 and cp_4 as variables to be optimized, we can use them to compute the unconditional (or marginal)



n	cp_n	p_n
0	1	1
1	cp_1	cp_1
2	$1 - cp_1$	$1 - cp_1$
3	1	$1 - cp_1$
4	cp_4	$cp_1 cp_4$
5	$1 - cp_4$	$cp_1 - cp_1 cp_4$
6	1	$1 - cp_1$
7	1	1
8	1	$cp_1 - cp_1 cp_4$

Figure 3. e-graph from Figure 2 with hypothetical probabilities — in each e-node, operator is replaced with numbered e-node ID. Diamonds show conditional probabilities (cp) as variables to be optimized. All conditional probabilities and computed unconditional probabilities (p) are summarized on the right.

probabilities p_1 and p_7 in the following way:

$$p_1 = cp_1 \cdot P(m_{ec(1)} \text{ is chosen}) = cp_1 \cdot p_0 = cp_1,$$
$$p_7 = cp_7 \cdot P(m_{ec(7)} \text{ is chosen}) = cp_7 \cdot \max(p_0, p_3) = 1$$

Here n_0 and n_3 are fully correlated.

If we use the hypothetical cost in Figure 3, the final cost of this e-graph would be: $f(p) = u^T p = 19 + 13cp_1 - 5cp_1 cp_4$. It is obvious that when $cp_1 = 0$, $f(p)$ achieves minimum: $f(p^*) = 19$. This corresponds to the optimal binary solution 3 in Figure 2c. By optimizing $f(p)$, we can obtain the probability corresponding to the optimal solution.

3.4 Handling Acyclicity Constraint

Enforcing the acyclicity constraint during e-graph extraction presents a significant challenge due to the combinatorial nature of the problem, especially within the scope of this research. To address this, we make use of a method called NOTEARS [57], which reformulates the combinatorial problem of extracting a directed acyclic graph into a continuous optimization problem.

To explain the core idea of NOTEARS, we include the following theorem and a brief proof:

Theorem 3.1. *Let A_t be the transition matrix for a connected directed graph G with d vertices. G is acyclic if and only if*

$$h(A_t) = \text{tr}(e^{A_t}) - d = 0, \tag{8}$$

where $\text{tr}(\cdot)$ is the trace operator, e^{A_t} is the matrix exponential operator, d the number of nodes in this graph.

Proof. For any positive integer k, $A_t^k[i, j] = 0$ is the sum of products along all k-hop paths from i-th node to j-th node. Since A_t is non-negative, $\text{tr}(A_t^k) = 0$ if and only if there are no k-hop cycles in the graph. Using Taylor expansion for the matrix exponential, we have

$$\text{tr}(e^{A_t}) = \text{tr}(I) + \text{tr}(A_t) + \frac{1}{2!}\text{tr}(A_t^2) + \cdots \geq d, \tag{9}$$

and the equality is attained if and only if G has no cycles of any number of hops. □

In our case, the transition matrix A_t captures the probabilities of dependency among e-classes, where $A_t[i, j]$ is the probability of e-class m_i depends e-class m_j. Specifically, $A_t[i, j]$ is equal to the sum of the conditional probabilities of all e-nodes n_k in m_i that depends on m_j: $A_t[i, j] = \sum_{k, n_k \in m_i, m_j \in ch_k} cp_k$. To enforce acyclicity, the NOTEARS term $h(A_t)$ is added as a penalty term to the primary objective, f, where a coefficient λ is used to control its relative scale. Thus we obtain the final form of the continuous optimization as follows:

$$\text{minimize}_{\ cp \in [0,1]^n} \mathcal{L} = f(p) + \lambda \cdot h(A_t) \tag{10a}$$

$$\text{s.t. } p = \phi(cp) \tag{10b}$$

Here $\mathcal{L}$ is the objective function, λ is a hyper-parameter to control the scale of the NOTEARS term. With a sufficiently large λ, it is provable that the probability of the solution to Eq. (10) containing cycles is zero.

3.5 Optimization and Sampling

At a high level, we use gradient descent to optimize the objective function (i.e., Eq. (10a)). Subsequently, a sampling stage is used to extract a discrete solution based on the probabilities assigned to the e-node selection variables.

In the optimization stage, we search for the solution that minimizes the objective function of the relaxed problem in Eq. (10). First, to obtain the conditional probability cp, we apply softmax function to a free variable θ of the same shape of cp. Then, unconditional probability p is computed according to cp with function ϕ using hybrid assumptions by default. Finally, p is used to compute the loss function $\mathcal{L}$. The gradient of the loss function is backward propagated to update the free parameters θ, thus updating cp.

In the sampling stage, we extract a discrete binary solution s according to the conditional probabilities cp assigned by the optimizer. Sampling is performed after each iteration of optimization. Concretely, we use the following sampling schedule: To satisfy constraint (a), we start from the selection of the root e-class m_0. For each selected e-class m_j, we select e-node $n_k^* \in m_j$ with the largest probability $cp_k^* = \max_{k \in m_j} cp_k$. For each selected e-node, we select all its child e-classes. Thus constraint (b) is also satisfied. This process is repeated until all selected e-nodes have no unselected dependent e-classes. Note that the sampling process only guarantees completeness constraints, (a) and (b). For the acyclicity constraint, we rely on the additional objective in the optimization process introduced in Section 3.4. SmoothE will terminate if (1) the cost of sampled solution does not improve for a certain number of iterations (i.e., based on a patience parameter), or (2) after a predefined number of iterations (i.e., a timeout).

4 Implementation and Performance Optimization

We introduce the algorithmic framework of SmoothE in the previous section. In this section, we discuss how we achieve a high-performance implementation on GPUs by exploiting sparsity and data parallelism.

4.1 Vectorization and Sparsity

Since our approach is fully differentiable, adopting PyTorch [37] is an attractive option for leveraging the modern machine learning ecosystem and GPU acceleration for gradient computation. To optimize GPU utilization, it is essential to vectorize intensive computations and memory operations. Additionally, e-graphs exhibit high sparsity, which is evident from the average edge density in the datasets we evaluated (refer to Table 1). Therefore, exploiting sparsity is also crucial to minimize memory usage and enhance overall efficiency.

In our implementation, the e-nodes and e-classes are represented by a vector of shape $N \times 1$ and $M \times 1$, respectively, where M is the number of e-classes and N is the number of e-nodes. Since the size of the e-graph is usually large and the edge density is low, we avoid instantiating any dense tensors equal or larger than shape of $\min(M, N)^2$. For example, the function $ec(i)$, which maps e-classes to child e-nodes, is represented by a $\{0, 1\}^{M \times N}$ sparse tensor. In this case, translating from e-classes to child e-nodes can be performed by a sparse matrix–vector multiplication (SpMV) operator. The implementation of ch_i and pa_j follows a similar approach.

4.2 Seed Batching

It is challenging to parallelize graph algorithm on GPUs due to irregular memory access [3] in traditional parallel schemes. Naive data parallelism requires dividing variables to different devices, which would incur non-trivial communication overhead among devices, because all e-nodes and e-classes are interconnected.

Therefore, we adopt a different technique called *seed batching* to fully utilize GPU power. We refer one instantiation of θ as one *seed*, whose initialization point depends on the random seed. With different θ, the conditional probability cp starts from different initializations. Since the optimization and sampling results are affected by the initialization, by optimizing and sampling more than one seed, we expect to find a seed leads to a solution with better quality.

At the beginning of optimization, B seeds are initialized, which later are optimized in parallel using a batch fashion. The discrete solution s with the lowest cost is selected from discrete solutions sampled from all B seeds. During sampling, we sample discrete solutions from all B seeds in parallel and select the one with the lowest cost as the final solution.

Table 1. Dataset Statistics — $\#G$ denotes the number of different e-graphs in this dataset. $d(v)$ represents the average e-node degree. $\max(N)$ and $\max(M)$ indicate the maximum number of e-nodes and e-classes, respectively, in any e-graph within this dataset. Avg. Density represents the average edge density of all e-graphs in this dataset.

Dataset	Task Description	$\#G$	$d(v)$	$\max(N)$	$\max(M)$	Avg. Density	Representative Workload(s)
diospyros [48]	DSP vectorization	12	2.5	218933	9584	4.8×10^{-3}	Linear algebra kernels
flexc [51]	CGRA mapping	14	1.8	19830	4892	2.5×10^{-4}	Bzip2 [38], FFmpeg [46]
impress [47]	FPGA HLS	3	2.0	102030	90312	4.7×10^{-5}	Large integer multiplication
rover [12]	Datapath	9	5.5	16960	2852	1.4×10^{-3}	DSP and graphics kernels
tensat [53]	Tensor graph	5	2.3	57800	34800	2.6×10^{-4}	ResNet-50 [25], BERT [19]
set	NP-hard problem	4	1.0	996738	104632	1.2×10^{-2}	Minimum set covering
maxsat	NP-hard problem	6	1.8	3851	3781	4.0×10^{-4}	Maximum satisfiability

4.3 Matrix Exponential Optimization

The computation of NOTEARS, which enforces the acyclicity constraint, primarily relies on the matrix exponential operation. Modern implementations of matrix exponential typically use the Padé approximation [20], which primarily involves matrix-matrix multiplication and solving linear systems. In practice, it often becomes the bottleneck of the entire optimization process. This is due to the fact that linear system solvers are typically memory-bound, posing challenges for efficient GPU optimization. We make efforts from two aspects to reduce the overhead of this operator:

Strongly Connected Component (SCC) Decomposition. Any directed graph can be decomposed into sub-graphs such that within each sub-graph and every node is reachable from every other node. These sub-graphs are known as SCCs and cycles can only occur within a single SCC. We decompose the e-graph into SCCs and sum up NOTEARS terms in Eq. (8) from each SCC. As long as the sum of NOTEARS terms is 0, each SCC will be cycle-free, ensuring that the extracted e-graph is acyclic. This significantly reduces the complexity of the matrix exponential computation.

Batched Approximation Solving linear systems is bottlenecked by irregular memory access, making it challenging to accelerate through batching. To address this, we adopt a batched approximation strategy: given a batch size of B, instead of averaging the matrix exponentials of all B matrices, we approximate by computing the exponential of the average of the B matrices.

$$\frac{1}{B}\sum_{i=1}^{B}\operatorname{tr}(e^{A_t[i]}) \approx \operatorname{tr}(e^{\frac{1}{B}\sum_{i=1}^{B}A_t[i]}), \tag{11}$$

where the $A_t[i]$ is the i-th seed of A_t. This approximation reduces the overhead of the matrix exponential operation by a factor of B.

5 Evaluation

5.1 Experimental Setup

Environment Settings. All the experiments are performed on a Linux server equipped two NVIDIA A100 GPUs with 80 GB memory and two AMD EPYC 9124 CPUs (2×16 cores) running at 3.7 GHz, 1.5 TB of RAM. For softwares, we use PyTorch 2.0.1 with CUDA 11.7 and torch_sparse [21] 0.6.17.

Baseline Methods. We compare SmoothE with three baselines: ILP, the default heuristic in egg, and an improved heuristic algorithm from the extraction gym [22], which we call heuristic+.

We evaluate three ILP solvers: COIN-OR branch-and-cut (CBC) [23], solving constraint integer programs (SCIP) [1], and CPLEX [4]. CBC and SCIP are open-source ILP solvers, while CPLEX is a commercial solver from IBM which supports multi-threaded. The thread number is set to 32 for CPLEX. The ILP formulation used in evaluation is the same as Eq. (1). All ILP solvers are invoked using Python APIs, and we count the time after the solvers are invoked. For heuristic algorithms, we use the original implementations in Rust.

Hyper-parameter Settings We find that using the hybrid assumption by default performs well enough across e-graphs from different datasets. To further improve the results, we conduct a simple grid search for all hyper-parameters, including assumption used, on a randomly selected e-graph from the dataset to improve extraction quality.

5.2 Real-World Datasets

We conduct a comprehensive comparative study of existing e-graph extraction methods and SmoothE on several real-world datasets, along with a adversarial dataset that provides additional insights into when the heuristic is less effective.

Table 1 provides a description of datasets and summarizes key statistics of e-graphs in each dataset. We select five different realistic datasets from prior work that utilizes e-graphs to perform various optimizations for software compilation or hardware synthesis. Specifically, rover [12] focuses on datapath synthesis to minimize area of several arithmetic-intensive kernels [24, 30]; tensat [53] optimizes the tensor computation graphs from deep learning models [19, 39]; flexc [51] optimizes the performance of loop kernels on coarse-grained reconfigurable arrays (CGRAs); impress [47] optimizes the resource usage of large integer multiplications on FPGAs leveraging high-level synthesis (HLS); diospyros [48] searches

Table 2. Comparative results using the linear cost model across 5 realistic datasets — Increases in solution quality are normalized to the oracle obtained by running CPLEX for 10 hours. The time limit is set as 15 minutes for all ILP methods. In each dataset, **time** is reported in second. **worst** indicates the statistic of the worst performing e-graph, and **avg.** reports the geometric mean across e-graphs with feasible solutions. The red number in parentheses indicates the number of e-graphs for which the solver fails to provide a valid solution. The maximum difference for SmoothE is based on 3 runs. For diospyros, flexc, impress, rover, and tensat, we use independent, correlated, correlated, independent, and independent assumption respectively.

Dataset	CPLEX ILP time (fails) worst / avg.	SCIP ILP time (fails) worst / avg.	CBC ILP time (fails) worst / avg.	Heuristic (egg) time (fails) worst / avg.	Heuristic+[22] time (fails) worst / avg.	SmoothE (ours) time (fails) worst / avg.
diospyros	211.8 19.6% / 1.4%	240.5 (1) Failed / 7.5%	350.4 (1) Failed / 2.5%	0.3 0.0% / 0.1%	0.2 0.0% / 0.1%	$8.4_{\pm 0.7}$ $0.1\%_{\pm 0.0\%}$ / $0.1\%_{\pm 0.0\%}$
flexc	5.2 0.0% / 0.0%	41.6 0.0% / 0.0%	585.4 (5) Failed / 66.6%	0.0 2.8% / 1.2%	0.1 2.8% / 1.2%	$19.3_{\pm 1.7}$ $0.7\%_{\pm 0.0\%}$ / $0.2\%_{\pm 0.0\%}$
impress	39.5 0.0% / 0.0%	69.8 0.0% / 0.0%	384.5 220% / 30.8%	0.4 280% / 53.0%	1.0 0.0% / 0.0%	$4.6_{\pm 0.0}$ $0.0\%_{\pm 0.0\%}$ / $0.0\%_{\pm 0.0\%}$
rover	520.4 4.3% / 0.5%	671.8 45.6% / 17.2%	677.6 58.6% / 19.6%	0.1 11.0% / 2.9%	0.1 11.0% / 2.9%	$20.6_{\pm 1.3}$ $4.4\%_{\pm 1.2\%}$ / $0.2\%_{\pm 0.1\%}$
tensat	678.2 9.7% / 1.9%	900.0 (1) Failed / 260%	900.0 (1) Failed / 67.2%	0.4 46.4% / 12.1%	1.3 46.4% / 11.9%	$24.4_{\pm 2.5}$ $17.0\%_{\pm 0.0\%}$ / $4.6\%_{\pm 0.1\%}$

for efficient vectorizations to speed up execution of linear algebra kernels on DSP architectures. Three of these datasets are available in the e-graph extraction gym [22], an open-source repository. We contacted the authors of impress [47] to get their codebase. Using the impress codebase along with open-sourced diospyros, we constructed the e-graphs.

5.3 Adversarial Datasets

To study the limitations of the heuristic methods, we construct two adversarial datasets with e-graphs rich in common sub-expressions. We converted two NP-hard problems, minimum set covering and maxsat, to e-graph extraction. The original problems are obtained from Frequent Itemset Mining Dataset Repository [55] and MaxSAT Evaluations [2], respectively. The conversion process follows the prior works [42, 55]. Compared to e-graphs in realistic datasets, the e-graphs converted from these NP-hard problems contains less graphical information, thus more suitable for ILP solvers.

5.4 Comparison Using Linear Cost Models

Results on Real-World Datasets. We first test on the most widely used linear cost models. For all the real-world datasets, the cost for each operator is application specific. For example, the cost for operators in tensat corresponds to their execution time on GPU, while the cost of operators in impress represent their resource usage on FPGAs.

Since we do not control the internal stopping criteria of ILP solvers and they may take hours or even days to produce results (if they complete at all), we set a hard time limit, 15 minutes, for all ILP solvers.

The results are shown in Table 2. We first run the CPLEX for 10 hours for each e-graph to obtain a good solution as an *oracle* baseline. We then compare the quality of the extracted solutions from different extraction methods with the baseline to obtain the normalized cost increase on each data. CPLEX, a commercial solver with multi-threading support, is able to obtain good solutions on flexc, impress and rover. The open-source ILP solvers, SCIP and CBC, on the other hand fail to obtain reasonable solutions on most datasets. Heuristic algorithm, is able to achieve good solutions on flexc and diospyros. The improved version of it, Heuristic+, can also find descent solutions on impress. This is due to 1) lack of potential common subexpressions and/or 2) reuse of common subexpression does not exist in the optimal terms.

SmoothE generates high-quality solutions on all five datasets. Compared to ILP solvers on all datasets, SmoothE achieves comparable or better quality, except flexc, with an 8× to 37× speedup. Compared with the heuristic baselines, SmoothE consistently provide better extraction quality, while keeping the extraction time much shorter than ILP. In summary, SmoothE offers similar solution quality to ILP baselines with much faster GPU execution, significantly surpassing the extraction quality of heuristic and heuristic+.

Anytime Results. Similar to ILP, SmoothE keeps refining the solution and is able to sample discrete solution at an any given time. In Figure 4, we include an anytime comparison between SmoothE and CPLEX on several e-graphs from two different datasets. We ignore other ILP solvers because they hardly produce a valid solution on this given time scale for these e-graphs. This experiment clearly shows that SmoothE

Table 3. TenSat and Rover results breakdown— the solution produced by the ILP solver before the timeout (15 min) is optimal; otherwise, it is a best-effort solution. **time** is reported in second.

Dataset	E-Graph	CPLEX cost / time	SCIP cost / time	CBC cost / time	Heuristic cost / time	Heuristic+ cost / time	SmoothE (ours) cost / time
tensat	NASNet-A	11.199 / 16.6	14.098 / 900.0	11.271 / 900.0	16.399 / 3.4	16.399 / 0.5	$11.857_{\pm 0.086}$ / $27.6_{\pm 4.5}$
	NASRNN	1.036 / 900.0	1.590 / 900.0	Fails / 894.3	0.972 / 2.4	0.963 / 1.2	$0.956_{\pm 0.001}$ / $42.4_{\pm 0.7}$
	BERT	0.705 / 900.0	14.151 / 900.0	3.516 / 897.7	0.825 / 0.3	0.825 / 0.1	$0.825_{\pm 0.000}$ / $18.4_{\pm 8.7}$
	VGG	4.851 / 900.0	Fails / 900.0	7.425 / 900.0	4.852 / 0.0	4.851 / 0.0	$4.851_{\pm 0.000}$ / $27.2_{\pm 17.8}$
	ResNet-50	4.386 / 674.4	4.386 / 900.0	4.457 / 899.6	4.397 / 0.1	4.397 / 0.0	$4.387_{\pm 0.001}$ / $6.2_{\pm 0.5}$
rover	fir_5	5936 / 900.0	7549 / 900.0	7296 / 900.0	5936 / 0.0	5936 / 0.1	$5936_{\pm 0}$ / $22.6_{\pm 6.1}$
	fir_6	6191 / 900.0	8641 / 900.0	8317 / 900.0	5936 / 0.1	5936 / 0.1	$5936_{\pm 0}$ / $23.6_{\pm 7.6}$
	fir_7	6011 / 900.0	8512 / 900.0	9617 / 900.0	5936 / 0.1	5936 / 0.2	$5936_{\pm 0}$ / $17.8_{\pm 3.4}$
	fir_8	6011 / 900.0	8509 / 900.0	8994 / 900.0	5936 / 0.1	5936 / 0.2	$5936_{\pm 0}$ / $29.2_{\pm 2.8}$
	box_3	1701 / 47.9	1701 / 900.0	1761 / 900.0	1701 / 0.0	1701 / 0.0	$1701_{\pm 0}$ / $7.8_{\pm 1.1}$
	box_4	1635 / 900.0	1827 / 900.0	1913 / 900.0	1762 / 0.1	1762 / 0.1	$1707_{\pm 20}$ / $35.3_{\pm 2.1}$
	box_5	1819 / 3.9	1819 / 454.6	1819 / 581.4	1819 / 0.0	1819 / 0.0	$1819_{\pm 0}$ / $2.0_{\pm 0.3}$
	mcm_8	1050 / 60.8	1050 / 96.5	1050 / 57.7	1165 / 0.1	1165 / 0.1	$1050_{\pm 0}$ / $21.9_{\pm 1.0}$
	mcm_9	1050 / 70.9	1050 / 95.1	1050 / 59.7	1165 / 0.1	1165 / 0.1	$1050_{\pm 0}$ / $25.3_{\pm 2.9}$

Table 4. Comparative results using the synthetic model across 2 datasets — we use both × and % to represent the normalized increase. For example, 6.3× is equivalent to 630%. Other notations follow Table 2

Dataset	CPLEX ILP time worst / avg.	SCIP ILP time worst / avg.	CBC ILP time worst / avg.	Heuristic (egg) time worst / avg.	Heuristic+[22] time worst / avg.	SmoothE (ours) time worst / avg.
set	33.3 0.0% / 0.0%	17.1 0.0% / 0.0%	32.6 0.0% / 0.0%	1.0 6.3× / 2.6×	0.6 6.3× / 2.6×	$16.5_{\pm 0.7}$ $50.0\%_{\pm 0.0\%}$ / $21.9\%_{\pm 0.5\%}$
maxsat	0.4 0.0% / 0.0%	0.2 0.0% / 0.0%	0.3 0.0% / 0.0%	0.0 2.0× / 2.0×	0.0 2.0× / 2.0×	$2.3_{\pm 0.5}$ $31.1\%_{\pm 0.4\%}$ / $22.9\%_{\pm 0.2\%}$

is able to find a comparable solution, if not better than ILP, within a much faster time.

Results on Adversarial Datasets. On adversarial datasets, all ILP solvers are able to extract the optimal solution within a minute, because these e-graphs converted from the NP-hard problems contains less graphical information, thus more suitable for ILP solvers. The heuristic methods fail to extract solution with reasonable quality, because there are more common sub-expressions in these converted datasets. SmoothE performs much better than the heuristic baselines.

5.5 Comparison Using Non-linear Cost Models

Since the code and datasets from recent efforts using non-linear cost models [7, 14, 52] are not open-sourced, for the evaluation, we generate an adversarial non-linear cost model, multi-layer perceptron (MLP), on the same set of e-graphs as in Section 5.4. The non-linear cost is added as a correction term to linear cost model: $f(x) = f_{\text{linear}}(x) + f_{\text{non-linear}}(x)$

For a fair comparison, we further implement genetic algorithm, a commonly-used meta heuristic. The genetic algorithm can flexibly support non-linear cost models; however, it may get stuck in local minima and fail to effectively explore a large search space. We did not include the genetic algorithm in Table 1 because (1) it does not perform better than the other heuristic baselines in terms of solution quality, and (2) our implementation is purely in Python, making the run time comparison less fair.

We primarily compare with genetic algorithm and ILP*. Because heuristic algorithm and ILP cannot handle non-linear terms. We directly use the oracle solutions obtained in Section 5.4 as an approximation by ignoring the non-linear terms, denoted as ILP*.

Learning-based Cost Model. ML-based models is capable of modeling more complex interactions of any pair of e-nodes in an e-graph. Furthermore, an ML-based model can also act as a fast proxy model to avoid some time-consuming compilation or synthesis. Popular learning-based model are

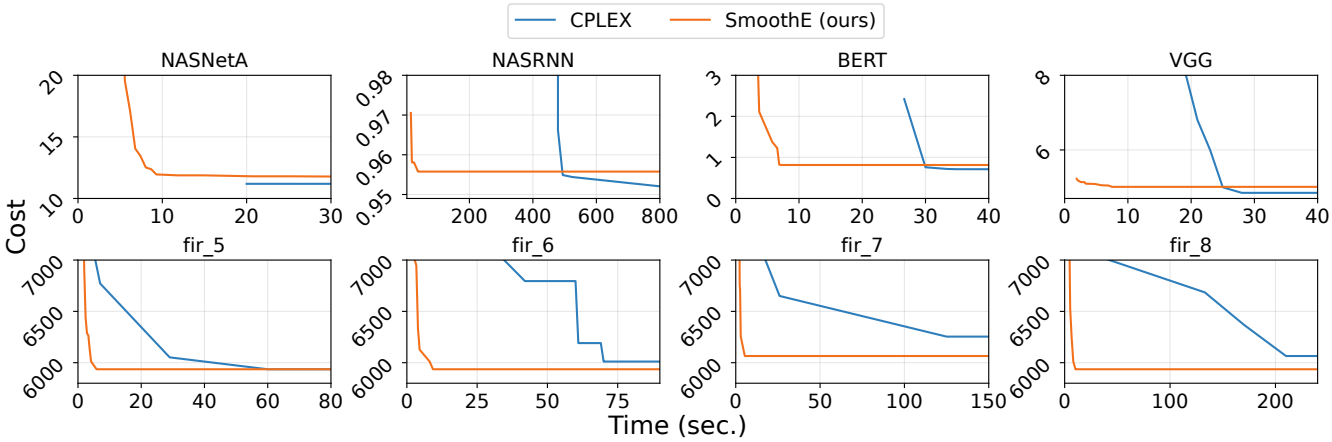


Figure 4. Anytime results — comparing SmoothE with CPLEX. The cutoff time for both algorithm is set to be 15 min. The x-axes are truncated if no further improvement is made.

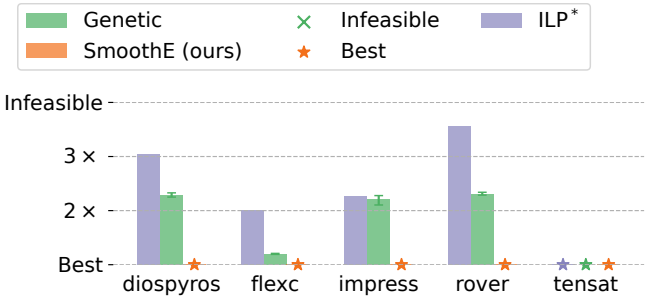


Figure 5. Comparative results using MLP costs across 5 datasets — increases are normalized to SmoothE results. The maximum difference of genetic algorithm is based on 3 runs. **ILP*** means using the oracle solution obtained by ILP.

differentiable, making them easy to incorporate into the end-to-end optimization flow of SmoothE.

In our evaluation, we use a simple multi-layer perceptron with 4 layers: an input layer of size N mapped to 64 neurons, followed by two hidden layers of 64 and 8 neurons each with ReLU non-linearity, and a final output layer of the predicted scalar cost. To train the MLP, we generate a synthetic dataset for each e-graph: the training data consists of random discrete valid solutions, and the target values are set as random negative numbers representing savings for these solutions. Regression is performed to optimize the MLP using this synthetic dataset.

Results on non-linear costs are shown in Figure 5. For all datasets, SmoothE consistently extracts the best solutions. While genetic algorithm is able to extract high-quality solutions on e-graphs in tensat, it fails to explore the large space for other datasets. The solution obtained from the approximated linear model consistently performs worse than SmoothE, because the interactions among e-nodes are more complex which cannot be captured by linear model, making the approximation inaccurate. This experiment showcases that SmoothE is capable of optimizing more complicated cost models other than linear cost models. In terms of the run time, when switching from linear models to non-linear models, the per-iteration update time of SmoothE remains approximately the same, as the cost model does not alter the overall workflow. Although SmoothE requires more iterations of optimization for the non-linear model due to its more complex dependencies, SmoothE finishes the extraction for any e-graph in Figure 5 within a minute.

5.6 Evaluation of Performance Optimizations

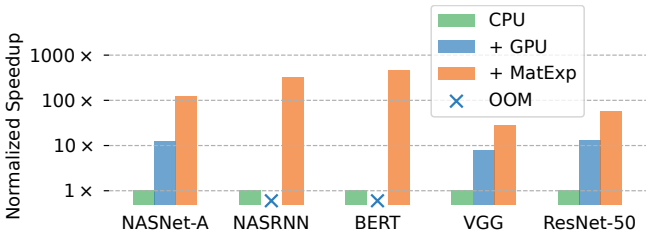


Figure 6. Speedup results normalized to the CPU baseline using tensat dataset — **+GPU** means switching the GPU execution, corresponding to Section 4.1; **+MatExp** corresponds to matrix exponential optimization corresponding to Section 4.3. **OOM** denotes GPU out of memory.

Ablation Study. Figure 6 breaks down how each component described in Section 4 contributes to the efficient execution of SmoothE. It shows the extraction time of SmoothE on e-graphs in tensat with different performance optimizations. For CPU baseline, we use the same algorithm described in Section 3 on a CPU backend without any performance optimization. Optimized GPU execution and matrix exponential optimization each provide approximately a 10× speedup.

The matrix exponential optimization is particularly crucial, not only for latency but also for memory usage. Without it, SmoothE will run out of GPU memory for large e-graphs.

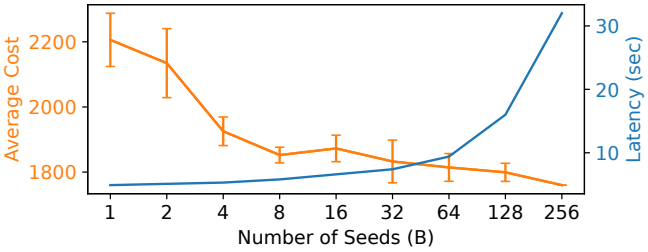


Figure 7. Averaged cost and latency of SmoothE with varying number of seeds (B) — this experiment is performed on an e-graph named box_3 from the rover dataset. The orange line shows the average cost and variance across on 3 runs, while the blue line indicates the latency.

Seed Batching. In Figure 7, we demonstrate the benefits of seed batching. When increasing the number of seeds used, the average cost and variance consistently decrease, meaning better and more consistent solution can be achieved. When the number of seeds is fewer than 64, the latency increase is far less than linear — doubling the number of seeds results in much less than 2× increase in latency, indicating the GPU is underutilized. Eventually, the cost converges to a low value with zero variance when using 256 seeds, meaning that consistent good solutions are extracted.

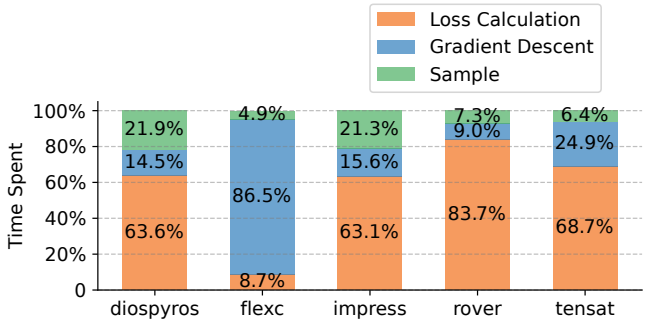


Figure 8. Run time profiling of SmoothE — this experiment shows how each component of SmoothE contributes to the runtime. Both **Loss Calculation** and **Gradient Descent** are parts of optimization. The reported number is the geometric average across all e-graphs within each dataset. All reported times are wall-clock measurements.

Profiling results are presented in Figure 8 to understand how each component of SmoothE contributes to the run time. The sampling process accounts for only 4.8% to 21.8% of the total run time. In contrast, the optimization process constitutes the majority of the run time. In optimization process, loss calculation is the most time-consuming component, except for flexc, where gradient descent is the slowest step.

5.7 Performance Portability

In this subsection, we analyze how the performance of SmoothE changes on lower-end GPUs. This primarily impacts SmoothE by reducing the number of seeds per batch due to smaller memory capacities, potentially affecting extraction quality. As long as the memory usage per seed remains within the GPU's limits—which is true for most e-graphs—SmoothE will still perform effectively.

We compare the performance of SmoothE on an NVIDIA GeForce RTX 2080 Ti and an A100, with results shown in Table 5. Due to the RTX 2080 Ti's 11 GB memory—approximately 8× smaller than the A100's 80 GB—batch size is reduced by 8×. Despite this, SmoothE produces effective solutions for most e-graphs in our evaluation, except for four cases where the memory required per seed exceeds the GPU's capacity. The run time remains largely unchanged and is sometimes even lower, as the compute requirements for optimization are reduced proportionally. Overall, SmoothE performs competitively on lower-end GPUs, though memory constraints may limit its ability to handle larger e-graphs.

5.8 Effectiveness of Sampling

In this experiment, we show that the sampling loss $f_b(s)$ in discrete form is close to the optimization loss $f(p)$ in relaxed form, indicating that the sampling process effectively extracts discrete solutions that are close to the relaxed solutions. In Figure 9, we compare the two losses on four e-graphs from tensat and rover. As can be seen from the figure, the optimization loss remains close to the sampling loss throughout all steps of the optimization process. This consistency highlights the effectiveness of our sampling method.

6 Conclusion

In this work, we propose SmoothE, the first differentiable approach to e-graph extraction. Our approach is well suited for GPU acceleration and can incorporate more expressive and realistic cost models, leading to higher solution quality on more complex extraction tasks. For future work, we plan to incorporate realistic non-linear cost models. We are also interested in leveraging advanced probabilistic graphical models to improve the probability computation.

7 Acknowledgment

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and NSF Awards #2019306, #2212371, #2403135, and an AWS AI Amazon Research Award. We sincerely thank Prof. Pavel Panchekha for his valuable feedback on the paper. We also appreciate the input and discussions from Prof. Chris De Sa, Yixiao Du, Andrew Butt, Zichao Yue, Hongzheng Chen, Samuel Coward, Dr. Theo Drane, Jiahe Shi, and the anonymous reviewers.

Table 5. Comparative results using different GPU targets – notations follow Table 2 and Table 3.

Dataset GPU Type	diospyros time (fails) worst / avg.	flexc time (fails) worst / avg.	impress time (fails) worst / avg.	rover time (fails) worst / avg.	tensat time (fails) worst / avg.
A100	$8.4_{\pm 0.7}$ $0.1\%_{\pm 0.0\%}$ / $0.1\%_{\pm 0.0\%}$	$19.3_{\pm 1.7}$ $0.7\%_{\pm 0.0\%}$ / $0.2\%_{\pm 0.0\%}$	$4.6_{\pm 0.0}$ $0.0\%_{\pm 0.0\%}$ / $0.0\%_{\pm 0.0\%}$	$20.6_{\pm 1.3}$ $4.4\%_{\pm 1.2\%}$ / $0.2\%_{\pm 0.1\%}$	$24.4_{\pm 2.5}$ $17.0\%_{\pm 0.0\%}$ / $4.6\%_{\pm 0.1\%}$
RTX 2080 Ti	$2.5_{\pm 0.1}$ (3) Failed / $0.1\%_{\pm 0.0\%}$	$28.7_{\pm 5.8}$ $1.7\%_{\pm 1.0\%}$ / $0.1\%_{\pm 0.1\%}$	$15.2_{\pm 0.1}$ $0.0\%_{\pm 0.0\%}$ / $0.0\%_{\pm 0.0\%}$	$13.4_{\pm 1.2}$ $11.0\%_{\pm 0.0\%}$ / $1.9\%_{\pm 0.1\%}$	$18.9_{\pm 2.7}$ (1) Failed / $6.2\%_{\pm 0.5\%}$

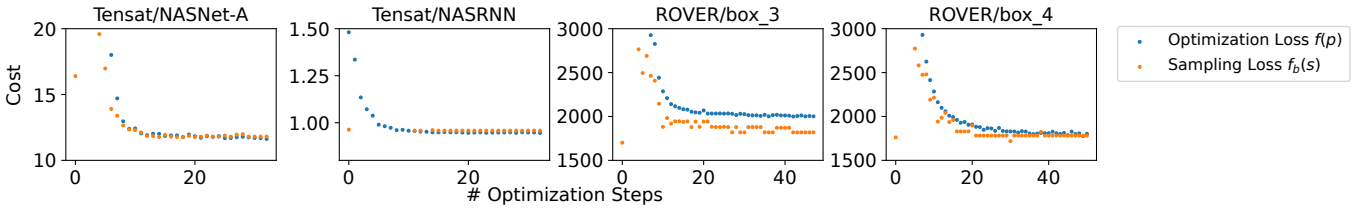


Figure 9. Optimization loss vs. sampling loss

A Artifact Appendix

A.1 Abstract

This artifact includes the SmoothE source code and the necessary scripts to reproduce the experiments in the paper. To facilitate artifact evaluation, we have automated the entire environment setup and experimental processes. Running the experiments requires a machine equipped with an NVIDIA A100 GPU with > 80GB memory. Reproducing SmoothE takes approximately one hour, while executing all baselines in the paper requires an additional four days.

A.2 Artifact check-list (meta-information)

- **Datasets:** All datasets are available within this artifact.
- **Run-time environment:** Specified in the conda `yaml` file.
- **Hardware:** A machine equipped with NVIDIA A100 GPUs.
- **Metrics:** Cost of the extracted e-graph (lower is better), run time (lower is better).
- **How much time is needed to prepare workflow (approximately)?:** About 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** It takes about an hour to reproduce SmoothE results, and 4 days for the baselines.
- **Publicly available?:** Yes.
- **Code and data licenses (if publicly available)?:** BSD 3-Clause License.
- **Archived (provide DOI)?:** https://zenodo.org/records/14052997

A.3 Description

A.3.1 How to access. Downloadable from: https://zenodo.org/records/14052997

A.3.2 Hardware dependencies. A machine equipped with an A100 NVIDIA GPU with >80 GB memory is highly recommended. Machine equipped with a lower-end GPU can also be used to run the experiments, but performance degradation similar to Table 5 is expected.

A.3.3 Software dependencies. We provide a conda `yaml` file for building the necessary Python dependencies needed by SmoothE. We use CPLEX 22.1.1.0, a commercial solver, for obtaining our baseline and oracle results.

A.3.4 Datasets. We include following datasets used to evaluate SmoothE and the baselines: (1) Realistic linear costs and synthetic MLP costs on diospyros, flexc, impress, rover, and tensat; (2) Linear costs on synthetic datasets converted from NP-hard problems, including set and maxsat.

A.4 Installation

Install all the required dependencies listed in `env.yaml`:

```
$ conda env create -f env.yaml
```

(*Optional*) Follow the CPLEX installation guide for free education version.

A.5 Experiment workflow

A set of Python scripts will generate the tables and figures in the paper. The complete instructions can be found in the `README.md` file included within the artifact repository.

A.6 Evaluation and expected results

We provide scripts for reproducing the results presented in Table 2, Table 3, Table 4, Figure 5, and Figure 4.

We execute SmoothE three times and report the max difference. The outcomes of the ILP and genetic baselines may exhibit variability due to their nondeterministic nature.

References

[1] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

[2] Josep Argelich, Chu-Min Li, Felip Manya, and Jordi Planes. The first and second max-sat evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):251–278, 2008.

[3] Maciej Besta and Torsten Hoefler. Parallel and distributed graph neural networks: An in-depth concurrency analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(5):2584–2606, 2024.

[4] Christian Bliek1ú, Pierre Bonami, and Andrea Lodi. Solving mixed-integer quadratic programming problems with ibm-cplex: a progress report. *Proceedings of the twenty-sixth RAMP symposium*, pages 16–17, 2014.

[5] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*, 7:396–424, 2023.

[6] Chong-Yun Chao and Earl Glen Whitehead. On chromatic equivalence of graphs. *Theory and Applications of Graphs*, pages 121–131, 1978.

[7] Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. *Design Automation Conf. (DAC)*, pages 1–6, 2024.

[8] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. Allo: A programming model for composable accelerator design. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 8:593–620, 2024.

[9] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 785–794, 2016.

[10] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. Seer: Super-optimization explorer for high-level synthesis using e-graph rewriting. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1029–1044, 2024.

[11] Samuel Coward, George A Constantinides, and Theo Drane. Abstract interpretation on e-graphs. *arXiv preprint arXiv:2203.09191*, 2022.

[12] Samuel Coward, George A Constantinides, and Theo Drane. Automatic datapath optimization using e-graphs. *IEEE Symp. on Computer Arithmetic (ARITH)*, pages 43–50, 2022.

[13] Samuel Coward, George A Constantinides, and Theo Drane. Automating constraint-aware datapath optimization using e-graphs. *Design Automation Conf. (DAC)*, pages 1–6, 2023.

[14] Samuel Coward, Theo Drane, Emiliano Morini, and George A Constantinides. Combining power and arithmetic optimization via datapath rewriting. *IEEE Symp. on Computer Arithmetic (ARITH)*, pages 24–31, 2024.

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[16] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Trans. on Evolutionary Computation (TEVC)*, 6(2):182–197, 2002.

[17] Nachum Dershowitz. A taste of rewrite systems. *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991–1992 McMaster University, Hamilton, Ontario, Canada*, pages 199–228, 2005.

[18] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Conf. of the North American Chapter of the Assoc. for Computational Linguistics (NAACL)*, pages 4171–4186, 2019.

[20] Luca Dieci and Alessandra Papini. Padé approximation for the exponential of a block triangular matrix. *Linear Algebra and its Applications*, 308(1-3):183–202, 2000.

[21] Matthias Fey. pytorch_sparse. https://github.com/rusty1s/pytorch_sparse.

[22] Oliver Flatt. Extraction gym. https://github.com/egraphs-good/extraction-gym.

[23] John Forrest and Robin Lougee-Heimer. Cbc user guide. *Emerging theory, methods, and applications*, pages 257–277, 2005.

[24] Oscar Gustafsson. A difference based adder graph heuristic for multiple constant multiplication problems. *Int'l Symp. on Circuits and Systems (ISCAS)*, pages 1097–1100, 2007.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[26] Mike He, Haichen Dong, Sharad Malik, and Aarti Gupta. Improving term extraction with acyclic constraints. *EGRAPHS 2023 workshop*, 2023.

[27] Alexander T Ihler, John W Fisher III, Alan S Willsky, and David Maxwell Chickering. Loopy belief propagation: convergence and effects of message errors. *Journal of Machine Learning Research (JMLR)*, 6(5), 2005.

[28] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[29] Avery Laird, Bangtian Liu, NIKOLAJ BJØRNER, and Maryam Mehri Dehnavi. Speq: Translation of sparse codes using equivalences. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2024.

[30] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *Int'l Symp. on Microarchitecture (MICRO)*, pages 330–335, 1997.

[31] Kevin P Murphy, Yair Weiss, and Michael I Jordan. Loopy belief propagation for approximate inference: an empirical study. *Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 467–475, 1999.

[32] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

[33] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[34] Julie L Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. Verifying and improving halide's term rewriting system with program synthesis. *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 4:1–28, 2020.

[35] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 50(6):1–11, 2015.

[36] Mark Paskin. A short course on graphical models. *The junction tree algorithms*, 2003.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Conf. on Neural Information Processing Systems (NeurIPS)*, 32, 2019.

[38] Julian Seward. bzip2 and libbzip2. *avaliable at http://www. bzip. org*, 1996.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *Int'l Conf. on Learning Representations (ICLR)*, 2015.

[40] Gus Henry 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). *ACM SIGPLAN Machine Programming Symposium (MAPS)*, pages 21–31, 2021.

[41] Gus Henry Smith, Zachary D Sisco, Thanawat Techaumnuaiwit, Jingtao Xia, Vishal Canumalla, Andrew Cheung, Zachary Tatlock, Chandrakana Nandi, and Jonathan Balkind. There and back again: A netlist's tale with much egraphin'. *arXiv preprint arXiv:2404.00786*, 2024.

[42] Michael B. Stepp. *Equality Saturation: Engineering Challenges and Applications*. PhD thesis, University of California San Diego, 2011.

[43] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[44] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*, pages 264–276, 2009.

[45] Samuel Thomas and James Bornholt. Automatic generation of vectorizing compilers for customizable digital signal processors. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 19–34, 2024.

[46] Suramya Tomar. Converting video formats with ffmpeg. *Linux journal*, 2006(146):10, 2006.

[47] Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. Impress: Large integer multiplication expression rewriting for fpga hls. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, pages 1–10, 2022.

[48] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 874–886, 2021.

[49] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. Spores: Sum-product optimization via relational equality saturation for large scale linear algebra. *Int'l Conf. on Very Large Data Bases (VLDB)*, 13(11), 2020.

[50] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*, 5:1–29, 2021.

[51] Jackson Woodruff, Thomas Koehler, Alexander Brauckmann, Chris Cummins, Sam Ainsworth, and Michael FP O'Boyle. Rewriting history: Repurposing domain-specific cgras. *arXiv preprint arXiv:2309.09112*, 2023.

[52] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I Lipton, Zachary Tatlock, and Adriana Schulz. Carpentry compiler. *ACM Trans. on Graphics (TOG)*, 38(6):1–14, 2019.

[53] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Conf. on Machine Learning and Systems (MLSys)*, 3:255–268, 2021.

[54] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. Grannite: Graph neural network inference for transferable power estimation. *Design Automation Conf. (DAC)*, pages 1–6, 2020.

[55] Yihong Zhang. The e-graph extraction problem is np-complete. https://effect.systems/blog/egraph-extraction.html.

[56] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 7:468–492, 2023.

[57] Xun Zheng, Bryon Aragam, Pradeep K Ravikumar, and Eric P Xing. Dags with no tears: Continuous optimization for structure learning. *Conf. on Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[58] Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang. Primal: Power inference using machine learning. *Design Automation Conf. (DAC)*, pages 1–6, 2019.