fuzzy-eqsat: Equality saturation with approximations

Jeff Zhang
McGill University
Montreal, Canada

***Abstract*—Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.**

I. Introduction

This paper explores the feasibility of introducing rewrite rules representing **approximations** for linear algebra operations, such as matrix multiplication, using equality saturation. The concept of compilers automatically inserting approximations for operations, trading precision for performance, is not new. A prominent example of this is the `-ffast-math` flag in GCC [1], which speeds up floating point operations by allowing the compiler to make many assumptions that do not respect the IEEE standard.

The approach explored in this paper requires that values in the expression being optimized are known ahead of time, which is highly restrictive. Nevertheless, this remains applicable for optimizing neural network inference, as the model parameters are all known after training is complete.

II. Background

A. Equality saturation

The fundamental data structure used in equality saturation is the **e-graph**. E-graphs are a graph data structure that contain a set of **e-classes** uniquely identified by an ID. E-classes are equivalence classes each containing a set of equivalent **e-nodes**. E-nodes represent a term in the language, with child terms being e-class IDs.

An e-graph compactly represents a very large number of equivalent expressions for a given expression subject to a set of **rewrite rules**. For example, we might have a rewrite rule that replaces division by 2 by a bitshift:

$$\frac{x}{2} \to x \ll 1$$

Equality saturation works by repeatedly applying these rewrite rules until:

1) The graph is saturated (applying further rewrites would not result in any changes)
2) A timeout is reached

Note that a saturated e-graph represents all possible equivalent expressions reachable from the initial expression using the given rewrite rules.

Then, the best possible expression can be extracted from the e-graph according a specific **cost function** which outputs a cost value given an e-node. However, the extraction problem is known to be NP-Hard [2], as such computing an exact solution is intractable for large e-graphs.

Common solutions are greedy approximation algorithm (taking the minimum cost e-node from each e-class) and integer linear programming (computes an exact solution, but can be slow).

Due to the constraints of the cost model chosen in this paper, a genetic algorithm was used for extraction.

B. Genetic algorithms

A genetic algorithm is a heuristic search method for optimization problems inspired by the biological process of evolution.

The set of parameters that encode a solution are called a **chromosome**, and a single parameter is called a **gene**. It works by randomly generating a **population** of initial solutions. These solutions are evaluated according to some **fitness function**. Then, the best solutions are **selected** via some to "reproduce" by combining their chromosomes via some **crossover** method to produce the next population, with the hope being that combining two good solutions will produce an even better solution. Finally, random **mutation** is applied gene by gene in the new offspring to introduce more diversity in the solutions. [3]

This process is repeated for some number of iterations, and the best solution is returned.

III. Methods

This approach to equality saturation comes with the restriction that all variable values in the expression are known ahead of time, since otherwise it is not possible to compute a relative error for approximations. The method was applied to the problem of neural network optimization for inference on two classification models.

1) A simple multilayer perception with 1 hidden layer
2) LeNet 5 [4], one of the earliest convolutional neural networks

Both of these models are trained on the task of handwritten digit recognition using the MNIST dataset [5].

A. Rewrite rules

The two approximation rewrite rules used were truncated singular value decomposition (SVD) and pruning. In addition to these, a basic rewrite rule for matrix multiplication associativity is also applied since this can give small cost reductions for free:

$$A(BC) \to (AB)C$$

a) ***Truncated SVD:*** The singular value decomposition of an $m \times n$ matrix A is a decomposition of the form

$$A = U\Sigma V^T$$

where U is an $m \times r$ matrix, Σ is an $r \times r$ diagonal matrix, and V is an $n \times r$ matrix, and $r = \min(m, n)$.

The truncated SVD is obtained by taking only the first k columns of each matrix, for some $1 \leq k < r$, giving us an approximation for A

$$A_k = U_k \Sigma_k V_k^T$$

This gives the **best rank-k** approximation for A by the Eckart-Young theorem [6].

Given a matrix multiplication AB, we rewrite A into its truncated SVD.

$$AB \to U_k \Sigma_k V_k^T B$$

If A is $m \times n$, and B is $n \times p$, then the total cost is mnp multiplications. When multiplying by the truncated SVD instead, the total cost is

- $V_k^T B$: knp multiplications
- $\Sigma_k\big(V_k^T B\big)$: kp multiplications since Σ_k is diagonal
- $U_k\big(\Sigma_k\big(V_k^T B\big)\big)$: kmp multiplications

$knp + kp + kmp$ can be much smaller than mnp if k is sufficiently small. In the rewrite implementation, the truncated SVD for several k are inserted in a linearly decreasing manner. The step size is equal to $\lfloor \log_2(\min(m, n)) \rfloor$. For example, for a 784×100 matrix, $\lfloor \log_2(\min(784, 100)) \rfloor = 6$, thus the truncated SVDs for $k = 94, 88, 82, ...$ are merged into the node's eclass. This rewrite is only applied once per e-class because since applying an SVD multiple times is redundant. This rewrite is also only applied to terms whose values can be known ahead of time, such as plain variables. An example of an e-graph with this rewrite applied is shown in Fig. 2.

b) ***Pruning:*** Pruning truncates small values in a matrix to 0. If Z is the amount of zeroes in matrix A, then to compute AB where A is $m \times n$, and B is $n \times p$, the number of multiplications required is $(mn - Z)p$ since we don't have to consider the zero entries in A when using a sparse matrix representation.

The pruning rewrite rule is applied to terms that can be known ahead of time such as plain variables or terms of their SVD. It simply introduces a new variable for the pruned value. This is done instead of adding a prune term to the language because the rewrite

$$A \to \text{prune}(A, k)$$

introduces a cycle into the e-graph, which would complicate extraction significantly. The actual rewrite is

$$A \to P_{i,k}$$

where $P_{i,k}$ is a fresh variable name that has the value of matrix A where all x in A such that $x < 10^k$ are truncated to 0. Similar to the truncated SVD, multiple pruning thresholds are inserted into the eclass. The values of k chosen were −3, −2, and −1.

B. Representation

The two neural network architectures used are both feedforward neural networks, as such they can easily be converted into a single expression for optimization.

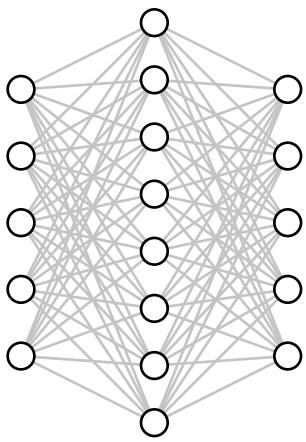


Fig. 1. A multilayer perception with 1 hidden layer.

`fuzzy-eqsat` uses the `egg` equality saturation library written in Rust, with the following grammar for expression rewrites.

Num	n		*integer literals*
Expr	e	$::= x$	*variable*
		$\mid e + e$	*matrix addition*

|	$e * e$	*matrix multiplication*
|	$\text{diag\_mul}(e_1, e_2)$	*diagonal matrix mul.*
|	$\text{svd\_u}(x, n)$	*U matrix of truncated SVD*
|	$\text{svd\_d}(x, n)$	*Σ matrix of truncated SVD*
|	$\text{svd\_vt}(x, n)$	*V^T matrix of truncated SVD*
|	$\text{ReLU}(e)$	*ReLU function*
|	$\tanh(e)$	*hyperbolic tangent function*
|	$\text{softmax}(e)$	*softmax function*

The grammar includes standard matrix addition and multiplication, `diag_mul` is a special case of matrix multiplication where e_1 is a diagonal matrix. This is faster to compute than normal matrix multiplication so it is discounted in the cost model.

The `svd_u`, `svd_d` and `svd_vt` terms correspond to the U_k, Σ_k, V_k^T matrices of the truncated SVD respectively. `ReLU`, `tanh`, and `softmax` are the common activation functions applied columnwise to matrices.

The network shown in Fig. 1 matches the architecture of the first classification model tested, albeit with fewer neurons. In `egg`, which uses an s-expression syntax, this network corresponds to the expression:

$$(\text{softmax}\ (+\ (*\ w_1\ (\text{ReLU}\ \ (+\ (*\ w_0\ x)\ b_0))\ b_1))$$

where w_i and b_i correspond to the weight matrix and bias vector for the i-th layer respectively, and x corresponds to the model input.

The model parameters are constants, but the model input evidently can't be known ahead of time. This can be resolved by treating x as a random variable which is defined by some distribution. The value assigned to x is the test set matrix from the classification problem's data set, since this approximates the distribution of possible model inputs. This is functionally the same as batch inference.

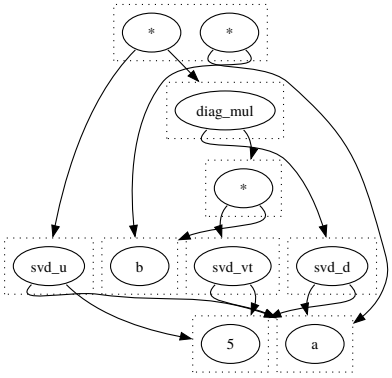


Fig. 2. E-graph for the expression (* a b) after applying the SVD rewrite

C. Cost model

The cost model used differs from typical equality saturation in many ways.

`egg` provides functionality to attach arbitrary data to eclasses during e-graph saturation via the `Analysis` trait.

First, during the analysis phase, a *canonical* expression value is computed for each eclass, which is derived from the variable values. The root e-class's canonical value corresponds to the actual model output given the test set.

The cost function implemented not only outputs an integer cost, but also the actual value of the e-node and its relative error compared to the e-class's canonical value. The relative error η between the canonical value A and its approximation A' is computed in terms of the Frobenius norm:

$$\eta = \frac{\|A - A'\|_F}{\|A\|_F}$$

The cost function itself can be configured by passing a maximum allowed relative error. The ordering between costs is in part determined using this value.

```
impl PartialOrd for CostWithErrorBound {
    fn partial_cmp(&self, other: &Self) →
Option<Ordering> {
        if self.error > self.max_error &&
other.error > self.max_error {
            self.error.partial_cmp(&other.error)
        } else if self.error > self.max_error {
            Some(Ordering::Greater)
        } else if other.error > self.max_error {
            Some(Ordering::Less)
        } else {
            self.cost.partial_cmp(&other.cost)
        }
    }
}
```

D. Genetic algorithm implementation

The e-graph extraction problem can be formulated as follows: Given an e-graph, pick an enode from each e-class so that the resulting expression's cost is minimized.

Chromosome: A solution to the extraction problem is simply a mapping from e-class IDs to indices, with the indices corresponding to a specific e-node in that e-class. This is represented using a hashmap.

Population size: The population size used was 100, which is somewhat small however the cost function is quite expensive to compute due to the size of the matrices. 100 was chosen to strike a balance between extraction time and quality of solutions produced.

Convergence: The algorithm stops once 5 iterations have passed without finding a new best solution, or after a maximum of 20 total iterations.

Selection: Since costs only have a relative order and not a single fitness value, rank-based selection is most suitable. Exponential rank selection is employed to more heavily favor the best solutions. The probability of choosing rank i (where rank 1 is the best solution) is given by

$$P(i) = \frac{0.9^{i-1}}{\sum_{k=1}^{n} 0.9^{k-1}}$$

where n is the population size (in this case 100).

Crossover: Crossover is done using simple single point crossover. The 2 parent solutions are flattened into an array, and a random index is chosen to be the crossover point. The two slices past the crossover point are then swapped, producing 2 children, as shown in Fig. 3.

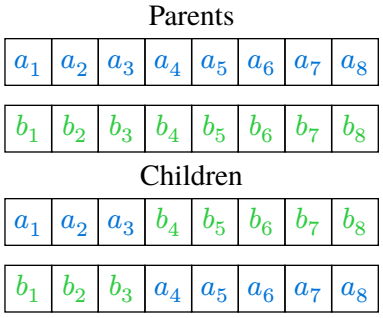


Fig. 3. Single point crossover, with crossover point 3.

Mutation: The mutation rate chosen was 5%, mutation simply randomly selects a different index for an e-class. This rate is rather arbitrary, but seems to produce good results.

IV. Experiments

Many different configurations were tested for both neural networks by:

- Varying the maximum allowed relative error (2%, 5%, 10%, 25%)
- Applying only the truncated SVD rewrite, only the pruning rewrite, and both rewrites

Extraction is run 3 times on each configuration, taking the best solution found out of the 3 runs, in order to minimize noise due to the randomness inherent to genetic algorithms.

Note that in the case of LeNet 5, since the equality saturation includes no rewrites for convolutions, the expression being optimized is only the last three fully connected layers of the network, since those are the relevant parts that can be optimized.

The experiments were run on a Lenovo Thinkbook G16 with a AMD Ryzen 7 8845H, running Arch Linux 6.17.

V. Discussion

Overall, `fuzzy-eqsat` is able to find good solutions for the two models that give significant reductions in cost without sacrificing too much on model accuracy.

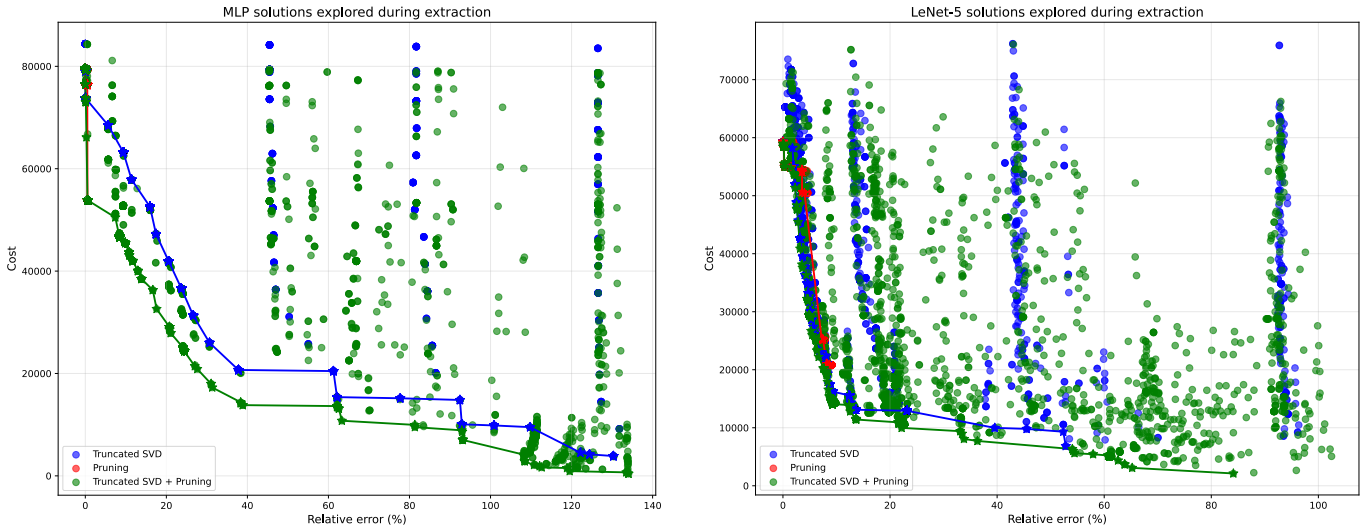
Looking at Table I, in the MLP case, classification accuracy sees no decrease at all until the relative error

TABLE I
Cost and error of optimized MLP expression for each rewrite rule configuration
Base classification accuracy: 97.02%
Base cost: 79620

	Truncated SVD				**Pruning**				**Truncated SVD + Pruning**			
Max rel. error	Cost	Rel. error	Rel. cost	Accuracy	Cost	Rel. error	Rel. cost	Accuracy	Cost	Rel. error	Rel. cost	Accuracy
2%	73790	0.047%	-7.3%	97.02%	53806	0.644%	-32.42%	97.03%	53806	0.644%	-32.42%	97.03%
5%	73790	0.047%	-7.3%	97.02%	53806	0.644%	-32.42%	97.03%	53806	0.644%	-32.42%	97.03%
10%	63170	9.410%	-20.66%	97.03%	53806	0.644%	-32.42%	97.03%	45424	9.862%	-42.95%	97.05%
25%	36620	23.70%	-54.00%	96.01%	53806	0.644%	-32.42%	97.03%	24338	24.42%	-69.43%	95.93%

TABLE II
Metrics for optimized LeNet expression for each rewrite rule configuration
Base classification accuracy: 98.56%
Base cost: 59348

	Truncated SVD				**Pruning**				**Truncated SVD + Pruning**			
Max rel. error	Cost	Rel. error	Rel. cost	Accuracy	Cost	Rel. error	Rel. cost	Accuracy	Cost	Rel. error	Rel. cost	Accuracy
2%	55112	1.988%	-7.138%	98.51%	54825	1.463%	-7.621%	98.56%	54825	1.480%	-7.621%	98.56%
5%	32426	4.950%	-45.36%	98.37%	50461	4.027%	-14.97%	98.49%	29377	4.925%	-50.50%	98.40%
10%	16232	9.305%	-72.65%	98.19%	20823	8.780%	-64.91%	98.22%	14083	9.434%	-76.27%	98.22%
25%	12931	23.13%	-78.21%	94.86%	20823	9.058%	-64.91%	98.18%	9982	22.16%	-83.18%	95.38%



a) MLP expression

b) LeNet-5 expression

Fig. 4. Explored solutions for MLP and LeNet-5 with each rewrite configuration. Points marked with a star and joined with a line mark the Pareto front.

surpasses 10%, in some cases even very slightly increasing, however this increase is insignificant. The truncated SVD rewrite by itself is fairly effective, giving a 20.66% cost reduction without compromising on accuracy, and a 54% reduction with only a 1% accuracy loss.

When only applying pruning on the MLP, each error threshold gave exactly the same solution with a 32.42% cost reduction while only having a relative error of 0.644%.

This indicates that there are many parameters in the MLP that have little impact on the final result, so pruning can be applied freely without tradeoffs. Since the model is small and pruning does not make the e-graph much bigger, the solution space is quite small, which explains why the same solution is obtained in all cases. This can be seen in Fig. 4 a), there are only a few red points on the very left side.

The optimization is most effective when the two rewrite rules are combined, since a matrix can be decomposed into its SVD and the resulting decomposition matrices can be pruned, further reducing cost. In the 2% and 5% max error cases, the solution is the same as pruning by itself, however when the max error is relaxed to 10% or 25%, the extractor finds a solution that is cheaper than either of the individual rewrite cases with similar accuracy.

In the case of LeNet-5, the accuracies in Table II show that this model is more sensitive to changes in the output, as the accuracy begins to fall before reaching a 10% relative error unlike the MLP, and the model suffers heavy accuracy losses of more than 3% when the relative error exceeds 20%.

Truncated SVD is notably more impactful in this case, giving a 72.65% reduction in cost while only incurring a 0.32% accuracy loss. Pruning gives different solutions unlike the MLP case but with larger error, indicating that the parameters are more meaningful compared to the MLP. This makes sense because LeNet-5 is a convolutional model and has fewer parameters than the MLP, so each neuron activation "represents" more information.

Combining the two rewrites for LeNet-5 is noticeably less effective compared to the MLP, resulting only in a 4-5% cost reduction. Indeed, Fig. 4 b) shows that the pareto fronts for truncated SVD and truncated SVD + pruning are very close together, unlike the MLP case where there is a sizable gap.

In all cases, it is evident that going past 10% error results in diminishing returns in terms of cost reduction. This can be seen in Fig. 4 b), the pareto front has a very steep downwards slope up to 10% error before flattening out quickly. The MLP does not seem to hit these diminishing returns until around 40% error.

The scatter plot for the LeNet-5 case is significantly denser than the MLP plot, this is because the solution space is much larger and thus more generations are required for convergence.

There is a clear pattern of vertical lines in both figures when only applying the truncated SVD rewrite (the blue points). These are likely solutions where a small matrix is heavily truncated which heavily influences the output while not reducing cost very much, which is combined with various truncations of a larger matrix, leading to this "vertical line" pattern.

VI. Limitations

The most obvious limitation is the requirement that all values are known ahead of time. This greatly restricts the number of possible usecases, though it is unclear how this requirement could be relaxed.

Additionally, the approach presented in this paper suffers from scaling issues, as more and more rewrite rules are introduced, the number of e-classes in the e-graph explodes, leading to a very large solution space. Due to the unreliable nature of genetic algorithms, it is likely that the extractor begins to fail to find good solutions, and would require further tweaking of the parameters. The crossover method could also be improved to be aware of the structure of the e-graph rather than simply swapping contiguous slices ordered by e-class IDs. Also, the extraction is quite slow since it involves evaluating many large matrix multiplications.

The cost model formulation could also be improved, the approach to e-node costs taken in this paper suffers from a sharp discontinuity when the

Finally, the cost values of the optimized expressions is simply part of the model and may not actually be faster in practice. Full performance data was not collected, but when running a few tests comparing the runtime of a model that was optimized using the truncated SVD rewrite to the original model, the "optimized" model was slightly slower.

A variety of factors could be the cause of this, but the most likely explanation is the CPU cache. A single matrix is stored in a contiguous block of memory which is very cache-friendly, while multiplying by the decomposed matrices performs three separate matrix multiplications, resulting in more cache misses.

For similar reasons, pruning is unlikely to result in a real speedup unless the resulting matrix is sufficiently sparse.

The models tested in this paper are very small. It is likely that the model being optimized would have to be much larger before a significant speedup could be observed.

VII. Related Work

Regarding introducing approximations into equality saturation, the idea appears to be novel and there is little to no trace of it in the literature, likely due to challenges already discussed.

There has been a fair amount of research devoted to tackling the e-graph extraction problem. Chen et al. [7] propose a solution based on simulated annealing with solution space pruning which achieves impressive results on large e-graphs for hardware synthesis. Rui et al. [8] take a similar approach also based on simulated annealing but combined with ant colony optimization.

Cai et al. [9] formulate the e-graph problem probabilistically, converting the discrete optimization problem into a continuous differentiable form and applying gradient descent.

Both of these approaches could be promising alternatives to the genetic algorithm extractor used in this paper, though the difficulty lies in coming up with a suitable cost model formulation for either of them to be usable since `fuzzy-eqsat`'s cost model is rather unusual.

VIII. Conclusion

This paper presents `fuzzy-eqsat` which introduces approximate rewrite rules to equality saturation, namely truncated SVD and matrix pruning for linear algebra expressions. Experimental results show large cost reductions on basic neural networks under the cost model used, though this does not lead to actual practical runtime improvement.

For future work, it would be interesting to rethink the cost model so that better extraction methods could be used instead of genetic algorithms, improving reliability and scalability. Generalizing the truncated SVD rewrite to tensor train decomposition [10] could also be useful for applying this approach to larger models that have higher dimensional parameter tensors.

References

[1] "Options That Control Optimization ," 2025, [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[2] M. B. Stepp, "Equality Saturation: Engineering Challenges and Applications," pp. 79–83, 2011.

[3] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.

[5] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012, doi: 10.1109/MSP.2012.2211477.

[6] C. Eckart and G. Young, "The Approximation of One Matrix by Another of Lower Rank," *Psychometrika*, vol. 1, no. 3, pp. 211–218, 1936, doi: 10.1007/BF02288367.

[7] C. Chen, G. HU, C. Yu, Y. Ma, and H. Zhang, "E-morphic: Scalable Equality Saturation for Structural Exploration in Logic Synthesis." [Online]. Available: https://arxiv.org/abs/2504.11574

[8] R. Li, L. Li, H. Yu, and Y. Ha, "ESACO: Fast E-Graph Extraction via Orchestrated Simulated Annealing-based Local Search and Ant Colony Optimization-based Global Search," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, no. , p. 1, 2025, doi: 10.1109/TCAD.2025.3611101.

[9] Y. Cai, K. Yang, C. Deng, C. Yu, and Z. Zhang, "SmoothE: Differentiable E-Graph Extraction," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, in ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 1020–1034. doi: 10.1145/3669940.3707262.

[10] I. V. Oseledets, "Tensor-Train Decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011, doi: 10.1137/090752286.