Q-gym: An Equality Saturation Framework for DNN Inference Exploiting Weight Repetition

Cheng Fu
cfu@ucsd.edu
UC San Diego
La Jolla, CA, USA

Hanxian Huang
hah008@ucsd.edu
UC San Diego
La Jolla, CA, USA

Bram Wasti
bwasti@fb.com
Meta AI
Menlo Park, CA, USA

Chris Cummins
cummins@fb.com
Meta AI
Menlo Park, CA, USA

Riyadh Baghdadi
baghdadi@mit.edu
MIT
Cambridge, MA, USA

Kim Hazelwood
kimhazelwood@fb.com
Meta AI
Menlo Park, CA, USA

Yuandong Tian
yuandong@fb.com
Meta AI
Menlo Park, CA, USA

Jishen Zhao
jzhao@ucsd.edu
UC San Diego
La Jolla, CA, USA

Hugh Leather
hleather@fb.com
Meta AI
Menlo Park, CA, USA

ABSTRACT

The high computation cost is one of the key bottlenecks for adopting deep neural networks (DNNs) in different hardware. When client data are sensitive, privacy-preserving DNN evaluation method, such as homomorphic encryptions (HE), shows even more computation cost. Prior works employed weight repetition in quantized neural networks to save the computation of convolutions by memorizing or arithmetic factorization. However, such methods fail to fully exploit the exponential search space from factorizing and reusing computation. We propose Q-gym, a DNN framework consisting of two components. First, we propose a compiler, which leverages equality saturation to generate computation expressions for convolutional layers with a significant reduction in the number of operations. Second, we integrate the computation expressions with various parallelization methods to accelerate DNN inference on different hardware. We also employ the efficient expressions to accelerate DNN inference under HE.

Extensive experiments show that Q-gym achieves 19.1% / 68.9% more operation reductions compared to SumMerge and original DNNs. Also, computation expressions from Q-gym contribute to 2.56× / 1.78× inference speedup on CPU / GPU compared to OneDNN and PyTorch GPU on average. For DNN evaluation under HE, Q-gym reduces the homomorphic operations by 2.47× / 1.30× relative to CryptoNet and FastCryptoNet for HE tasks with only 0.06% accuracy loss due to quantization.

PACT '22, October 10–12, 2012, Chicago,IL
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**.

ACM Reference Format:
Cheng Fu, Hanxian Huang, Bram Wasti, Chris Cummins, Riyadh Baghdadi, Kim Hazelwood, Yuandong Tian, Jishen Zhao, and Hugh Leather. 2022. Q-gym: An Equality Saturation Framework for DNN Inference Exploiting Weight Repetition. In *Proceedings of PACT '22: International Conference on Parallel Architectures and Compilation Techniques (PACT) (PACT '22).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep Neural Networks (DNNs) are becoming the *de-facto* solutions for various computer vision tasks [14, 33]. However, due to the large computation and storage cost of convolutional layers, DNNs are difficult to be integrated with many computation environments, such as mobile devices. To mitigate the computation and storage constraints for efficient DNN deployment, mainstream approaches include Quantizing DNNs [66, 68, 70] and leveraging DNN sparsity [23, 35, 36].

One key characteristic of quantized DNNs (QNNs) is *weight repetition*, i.e., weights in different layers are repetitions of a small number of Q unique ones (e.g., $\{-0.18, 0, 0.18\}$ and $Q = 3$). Leveraging such a regularity, a state-of-the-art approach SumMerge [43] reduces computation by two measures: **i)** Factorization – for example, factorizing a dot-product $ax+by+az+aw$ to be $a(x+z+w)+by$ and **ii)** Computation reuse – for instance, when computing two dot products $r_0 = ax + ay + az + aw$ and $r_1 = bx + ay + bz + bw$, SumMerge first calculates a *partial sum* $e_0 = x + z + w$ and reuses it in the downstream computations ($r_0 = ae_0 + ay$, $r_1 = be_0 + ay$).

While SumMerge indeed reduces the computation, it may not fully exploit weight repetition. As shown in Figure 1, while SumMerge leverages partial sums, it cannot reuse *partial products* (e.g., by, cz), leading to sub-optimal reduction. The underlying reason is straightforward: finding the best strategy to reduce the cost of an arithmetic expression is a combinatorial optimization problem, and the simple greedy heuristics leveraged by SumMerge may be insufficient.

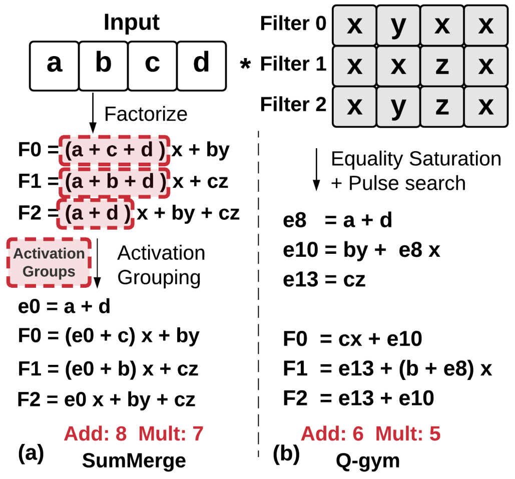

Figure 1: Comparison between Q-gym and a state-of-the-art approach SumMerge [43] with an example. (a) SumMerge first factorizes each individual dot-product and then performs greedy searches for the partial sum that can be reused by different activation groups. Partial products ($b \cdot y$, $c \cdot z$) are not reused. (b) Q-gym leverages both partial sums and partial products (e.g., $e10$ as a partial sum and product, is reused in the follow-up computation of $F0$ and $F2$).

Another technique for reducing the cost of convolutions is Winograd [31] which uses fast FFT to reduce computation operations in convolutional layers. Although this can have a significant effect on the number of operations, it does not take advantage of weight reuse and so does not achieve optimal reduction.

In this paper, we propose *Q-gym*, a framework to accelerate DNN inference by fully exploiting weight repetition. Q-gym comes with two components: a *compiler* that finds smart ways to reduce addition and multiplication counts in DNN evaluation, and a set of *downstream tasks* that leverage the compiled output to reduce its wall-clock time.

Q-gym's Compiler. To reuse both partial sum and product results, our compiler aggressively explores the search space by leveraging a data-structure called *e-graph* [15, 38] that represents a large number of equivalent expressions in a compact manner (Figure 2). Thanks to its compact representation, the search space is easier to navigate. To identify the optimal solution in the search space, our compiler performs three steps. (i) Following *equality saturation* [63], we expand the e-graph by repeatedly applying a set of rewrite rules (Table 2). (ii) Then, we extract the optimal computation expressions using integer linear programming (ILP) [52]. (iii) With large convolutional layers, e-graph expansion hardly saturates. In order to fully explore the search space with such layers, we employ a *pulsed searching* [27] algorithm, which iteratively applies the first two steps until convergence of the computation cost. In the simple example illustrated in Figure 1, our compiler identifies its optimal solution. For large convolution layers, Q-gym can still find more optimized solutions compared to the greedy-based method.

We also discover that the input activations overlap during the 'sliding' computation of weight kernels. That means computations can be reused between the sliding convolutions. With the proposed powerful searching algorithm, our compiler identifies a better solution by leveraging an extended search space compared with SumMerge [43] and Winograd [31].

Q-gym's Downstream Tasks. With the reduced DNN operations generated by our compiler, we can accelerate multiple downstream tasks, such as DNN evaluation on CPU/GPU and DNN evaluation under Homomorphic Encryption (HE) [19].

We propose an acceleration scheme to exploit the parallelism and locality in CPU/GPU to coupe with Q-gym's efficient expressions. In particular, by replacing the inner-dot product of a convolution layer with our expressions, we can exploit the reduction in operations to yield speedup. With different configurations in parallelization, our acceleration scheme can fully utilize the computation resource available. Our acceleration scheme can also effectively integrate with other parallelization schemes, such as loop tiling, vectorization, and multithreading.

We also propose to use Q-gym's efficient expressions for DNN evaluation under HE for preserving data privacy. It is compatible with mainstream HE libraries while significantly reducing the evaluation overhead.

Experiments. Extensive experiments show that our framework effectively accelerates various DNN applications. For QNN where the number of unique weight (Q) $Q \leq 12$, Q-gym reduces the number of FLOPs (#FLOPs) [1] by 68.9% / 19.1% compared to naïve QNN / SumMerge on average. On CPU ($Q \leq 3$), Q-gym achieves a speedup of 1.83× / 2.56× compared to SumMerge / OneDNN [2]. On GPU ($Q \leq 3$), Q-gym achieves a speedup of between 1.64× / 1.78× relative to SumMerge / PyTorch GPU [40]. For DNN evaluation under HE, Q-gym shows 59.5% / 22.9% HE operation reduction compared to CryptoNet [19] and Faster CryptoNet [13] with only -0.06% accuracy reduction due to quantization.

Summary. We summarize our contributions as follows:

- We propose Q-gym, a framework for quantized neural networks that can yield efficient computation dataflow for various downstream applications.
- Q-gym leverages an iterative searching algorithm, e-graph representation, ILP formulation, and elaborated search space to accelerate QNN inference.
- We implement back-end frameworks of Q-gym for multiple sub-domains for QNNs deployment tasks, such as CPU / GPU inference and HE applications. We combine Q-gym with other acceleration schemes and HE protocols.
- We corroborate Q-gym's general applicability and superior performance across various quantized DNN models.

Q-gym is the first framework to fully exploit weight repetition to accelerate privacy-preserving DNN inference. Also, this is the first work that applies the idea of equality saturation to simplify the arithmetic of DNN computation.

[1]The number of FLOPs in this paper refers to the total number of adds and multiplies for computing a convolution layer.

2 BACKGROUND

Background on CNN. While Q-gym is broadly applicable to any DNNs that can be represented by dot-products, we focus on deep convolutional neural networks [21, 32] (referred to as DNNs in the rest of the paper) as examples in this paper to make our proposal more concrete.

DNNs are commonly used for image classification [21, 30], object detection [48, 49], and image segmentation [50]. A convolutional layer implements a set of weight kernels to detect features in the input image. A weight kernel is defined by a set of weights, W, and a bias term, B. Each convolutional layer applies K kernels of dimension $R{\times}S{\times}C$ on the input with dimension $H{\times}W{\times}C$, resulting in output feature maps with dimension $(H-R+1)\times(W-S+1)\times K$. C denotes the number of input channels. H and W denote input height and weight. $R\times S$ denotes kernel shape. Formally, suppose the input activation is IA and weight is L, then the output activation O of this convolution operation is the dot product between input IA and parameters L, added by the bias B, and followed by a non-linear activation function $g(\cdot)$. For a layer with a unit stride, this can be represented as:

$$O(x,y,k) = g(B + \sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1} L(r,s,c,k)\cdot IA(x+r,y+s,c)) \quad (1)$$

For a quantized DNN (QNN), we denote the number of unique weights as Q. For a sparse DNN, we denote the sparsity ratio – the ratio between the number of non-zero weights and the number of parameters – as Sp.

Background on Homomorphic Encryptions. A broad range of applications, such as medical [26, 45], and fraud detection [5, 41], require privacy and confidentiality in client data. Homomorphic Encryption (HE) [18] is an ideal solution for such applications. When using HE for DNN inference, the *Model vendors* provide DNN model M and evaluation function f that computes $\square$ which satisfies $E(x_i)\square M = E(f(x_i, M), k_{pub})$. After the *cloud* completes the model inference, the client can decrypt (D) the result using the private key k_{pri} through Eq.(2). In this way, both parties can keep their data private during the DNN evaluation.

$$f(x_i, M) = D(E(f(x_i, M), k_{pub}), k_{pri}). \quad (2)$$

The key bottleneck of applying HE on DNN is that homomorphic multiplications and additions are extremely computationally expensive [13, 25]. An evaluation of CryptoNet [19] requires 250 seconds on the MINST dataset. The inference time of DNN under HE is proportional to the number of HE operations [13] (Detailed in Sec. 5.2).

In this paper, we identify that exploiting weight repetition can greatly reduce the computation overhead for popular HE protocols. The efficient evaluation function f from Q-gym reduces the number of HE operations by up to 2.47× / 1.30× compared to the state-of-the-art methods (FastCryptoNet/CryptoNet).

3 MOTIVATION

In this section, we discuss the motivations for proposing Q-gym.

Weight Quantization is Not Fully Explored. Weight quantization is a widely-used method to reduce the storage and computation overhead for deploying DNNs (See Section 7). Most existing works for QNN acceleration, such as BitFusion [54] and FLIM [58], leverage the shorter bit width of weights to accelerate each multiplication and addition. However, we identify that weight repetition features in QNNs are not fully explored for computation reduction in existing works. Specifically, we can bypass computations by factorizing the computation or reusing partial results. (Figure 1).

Table 1: Comparison between methods that reduce QNN computation costs by leveraging weight repetitions. *The temporal reuse scheme is described in Section 4.5.

Methods	Algorithms		
	AGR	SumMerge	Q-Gym
Evaluated Hardware	CPU	CPU	CPU+GPU+HE
HE Application	×	×	✓
Temporal Reuse*	×	×	✓
Bypass partial product	×	×	✓
Average FLOPs Reduction	<49.8%	49.8%	**68.9%**
Max CPU/GPU Speedup	-	3.1× / 1.6×	**5.9×** / **3.1×**

Previous Works Employing Weight Repetition is Sub-optimal. Activation group reuse (AGR) [22] is the first method that exploits weight repetition for efficient inference. The dot-product between input and each weight kernel will be factorized into expressions with reduced multiplications (Figure 1(a)). The operands of factorized expressions formed *activation groups* and common subexpressions between groups can be shared to reduce computation. Starting from the first activation group, AGR greedily extracts the maximum overlapping term between the first and the rest of the terms in the activation groups. Then, it recursively searches the second term and so on.

SumMerge [43] also adopts factorization and activation grouping schemes. Different from AGR, SumMerge (1) iterates all activation groups pairs and generates shared 'sub-computation' terms between activation groups in the activation grouping phase and (2) employs a "maxscore" – defined as the number of times each 'sub-computation' term appears across activation groups – to determine the sub-computation terms to be reused across. SumMerge iterates between (1) and (2) until no shared sub-computation terms are left.

However, SumMerge is inefficient in the following aspects. i) SumMerge is still a greedy heuristic. Extracting the sub-computation term with the best 'maxscore' in each iteration may not be the optimal solution. ii) Factorization can guarantee an upper bound on multiplications ($Q \cdot K$) for the convolution layer L. However, the search space of SumMerge is limited, as it only searches for common sub-expressions between activation groups. In other words, SumMerge does not fully exploit the search space of potential computation reuse. For example, it does not reuse the computed result to bypass partial products (e.g., by, cz in Figure 1). Also, with the increase of Q, the performance of SumMerge drops drastically (Section 6.1) as the size of each activation group reduces. iii) With the increasing size of weight kernels, the computation of "maxscore" is extremely expensive and incurs a large compilation time. In Figure 1, SumMerge and AGR yield the same results. In Table 1, we summarize the comparisons between Q-gym and AGR/SumMerge.

Applying Weight Repetition in Acceleration is Non-trivial. As shown in Eq.(1), the computation of a convolution layer involves 6 loops (H, W, R, S, C, K), and how to parallelize the computation has been a long-time problem [12, 46, 54]. The most common techniques involve 1) using loop tiling [54] to get better data locality. 2) Multi-threading and 3) SIMD vectorization.

These techniques cannot be directly used in our scenario as Q-gym changes the computation dataflow between the input and weight kernels. In this paper, we carefully design our parallelization scheme so that we can combine the efficient arithmetic compiled from Q-gym with the common acceleration methods on general-purpose hardware (Detailed in Sec. 5.1).

4 COMPILER DESIGN

4.1 Overview

As discussed in Section 3, the previous method of SumMerge [43] with a greedy heuristic cannot fully explore the search space and fails to maximize computation reuse. We propose Q-gym that leverages equality saturation (Section 4.2) to resolve the limitations of SumMerge. Q-gym's compiler incorporates a two-phase design: *exploration* and *extraction*. During the exploration stage (Section 4.3), Q-gym applies a set of rewrite rules on the computation expressions represented in an efficient data structure that can compactly represent a large number of equivalent expressions. For the extraction stage (Section 4.4), Q-gym formulates the selection of expressions as an ILP problem and finds more low-cost solutions compared to the greedy method. To handle an even larger search space, Q-gym also uses a *pulsed searching* algorithm [27] that iterates the *exploration* and *extraction* until convergence (Section 4.5). We also observe that inputs are overlapped during the sliding window computation of output. As such, we propose a *temporal reuse* search space that can further reduce the operations (Section 4.5).

4.2 Equality Saturation

Equality saturation [57, 62] is a method to resolve the exponential time and space requirement of traditional graph rewriting. Leveraging the efficient data structure *e-graph* underlying the equality saturation, the rewriting can be applied to the e-graph simultaneously without interfering with each other. Also, the e-graph is a compact representation of equivalent representations, which resolves the memory space limitation in traditional rewriting. In this paper, we propose to use the idea of equality saturation to reduce QNN operations. Many other applications leveraging equality saturation are discussed in Section 7.

In this subsection, we introduce the e-graph and the rewriting rules applied in Q-gym.

E-graphs. An e-graph is a data structure that compactly represents equivalent expressions. An e-graph contains a set of *e-classes* and a set of *e-nodes*. Each e-class represents a set of equivalent terms that can be computed from any of its e-node children. Each root e-class represents the final computed term from the input and a weight kernel.

Figure 2(a) shows an e-graph that represents the computation of an input ($[a, b]$) and two simple weight kernels ($[x, x]$ and $[x, y]$). The expressions that compute these computations are given at the bottom of the graph, defining e8 and e6 respectively. After applying rule rewriting (Table 2) to *saturation* (defined in Section 4.3), the e-graph will be expanded to Figure 2(b).

Table 2: Rewrite Rules used in Q-gym for DNN arithmetic simplification. e_i denotes an e-class.

Description	Source	Target
Mult Commutative	$e_i \times e_j$	$e_j \times e_i$
Add Commutative	$e_i + e_j$	$e_j + e_i$
Add Associative I	$(e_i + e_j) + e_k$	$e_i + (e_j + e_k)$
Add Associative II	$e_i + (e_j + e_k)$	$(e_i + e_j) + e_k$
Mult Distributive	$(e_i + e_j) \times e_k$	$e_i \times e_k + e_j \times e_k$
Mult Factorise	$e_i \times e_k + e_j \times e_k$	$(e_i + e_j) \times e_k$

In Q-gym, an e-node is one operator (i.e., '*' and '+') associated with two operands. The input value of each e-node's operand is a child e-class. Formally,

(i) *An e-graph represents a term if any of its e-classes do.*

(ii) *An e-class represents a term if any of its equivalent child e-nodes do. All terms represented by an e-class are equivalent.*

(iii) *An e-node $f(c_0, c_1)$ represents a term $f(e_0, e_1)$ if each e-class e_i represents c_i.*

In Figure 2(b), 'e13' is an e-class that represents both $b + a$ and $a + b$. In Figure 2(a), the e-node marked in red represents the term $x * a$ as e-classes e_0 and e_1 represent x and a.

Rewrite Rules. In equality saturation, we expand the egraph using a set of rewrite rules given in Table 2. The set of rewrite rules states the equivalence between each pair of computation arithmetic expressions. During e-graph exploration, rewrite rules will be applied to the e-graph. Specifically, we search for a pattern (source pattern) in the input e-graph that is equivalent to another subgraph pattern (target pattern). Q-gym begins rewriting the e-graph after all the applicable rules are searched.

Note that some of the rewrite rules do not create any new e-class to the e-graph (e.g. the commutative rules). However, they may allow other rules to be applied in the computation (e.g. factorization). Also, these rules increase the connections between e-nodes and e-classes, which enables the extraction phase to find better expressions for computation reuse.

4.3 Exploration Phase

As discussed in Section 4.1, in the first stage of equality saturation, we expand the e-graph using a set of rewrite rules. For example, the exploration phase expands the e-graph from Figure 2(a) to Figure 2(b).

An e-graph is initialized from the input expressions (Figure 2(a)). Then, we search the source pattern of each rewrite rule in the e-graph. If a match is found, it returns the e-class and corresponding substitution (target pattern) to the 'match list' U.

Once all the (e-class, substitution) pairs are found, we begin to modify the e-graph in the 'match list' simultaneously. Each pair of modifications (i.e., e-class and substitution pair) may create a new e-class/e-node in the graph unless the e-class/e-node already exists.

An e-graph is called *saturated* if no more e-class/e-node can be created. The exploration phase will stop when one of the following three conditions is met: 1) The e-graph is saturated. 2) The maximum exploration iterations limit $max\_explore\_iter$ is reached. 3) The maximum e-node limit max_{enode} is reached. Note that the e-graph for Q-gym is finite though possibly very large. Long-running explorations are prevented by $max\_explore\_iter$ and max_{enode}. The pseudo-code of the exploration phase is shown in Algorithm 1.

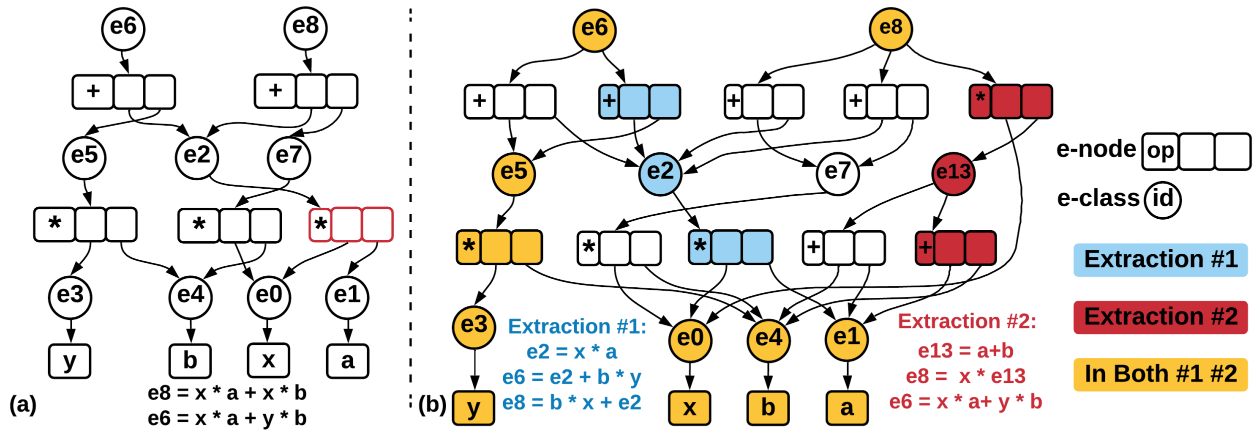

Figure 2: An example of (a) e-graph initialization and (b) e-graph after expansion using rules in Table 2. With different e-class / e-node selected, we can extract different equivalent computation expressions from an expanded e-graph.

Algorithm 1 Equality Saturation.

```
Input: Starting e-graph $G$; A set of rewrite rules $R$; Max e-node limit $max_{enode}$; Max exploration iterations $max\_explore\_iter$
Output: Explored e-graph $G$; Nodes selected $L_{node}$; Objective value $cost$.
1: $i \leftarrow 0$
2: while $i < max\_explore\_iter$ do
3:     $G' \leftarrow G, i \leftarrow i + 1, U \leftarrow \emptyset$ // match list
4:     $U \leftarrow U \bigcup$ Rule_Searching$(G, r_j)$ for $r_j$ in $R$
5:     $G \leftarrow$ Applying_Rules$(G, u_j)$ for $u_j$ in $U$
6:     if $G == G'$ or $num\_node(G') \geq n_{max}$ then
7:         break; // Saturate or max node limitation reached
8:     end if
9: end while
10: $G, L_{node}, cost \leftarrow$ Extraction$(G)$
11: return $G, L_{node}, cost$
```

4.4 Extraction Phase

After the exploration phase, the selection of different e-nodes or e-classes from the root node may yield different computation expressions (e.g., Extraction #1 and #2 in Figure 2(b)). The extracted expressions are equivalent to the input term which is guaranteed by the feature of e-graph. The goal of the extraction phase is to select the best-represented term for each root e-class according to a cost model.

Many extraction methods for equality saturation have been proposed, such as greedy algorithms [39] or ILP solutions [60, 61]. One of the key differences between simplifying DNN arithmetic and previous works of equality saturation (Section 7) is that multiple root nodes co-exist during the extraction phase. That is because each weight kernel $(k, k \in \{0, ..., K\})$ in layer L yields different output activation results $(O(x, y, k))$.

In this section, we discuss two strategies to extract low-cost expressions from an expanded e-graph.

Notations. Let i be an e-class where $i = 0, ..., N-1$. Let m be an e-node where $m = 0, .., M-1$. Let $m.op$ denote the operator of the e-node and $m.r_0$ / $m.r_1$ denote the operands of the e-node. Let $root$ be the set of root e-classes.

Let e_i and n_m denote a binary integer which indicates if the corresponding e-node i and e-class m is selected or not.

Let $m.r0$ denote the child e-class of operand $r0$ in e-node m, and let $child(i)$ denote the set of child e-nodes of e-class i, i.e., $\{m | m \in child(i)\}$.

Cost Model. In this paper, we define our cost model for an e-node m as Eq.(3).

$$cost(m) = \begin{cases} C_{add}, & \text{if } m.op = + \\ C_{mult}, & \text{if } m.op = \times \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Here, C_{mult} and C_{add} are constant values. $m.op$ indicates the operator associates with the e-node m. On an Intel i7-8700K CPU, the latency ratio between floating-point (FP) multiplication and FP addition is 3:5. And their reciprocal throughput ratio is 1:2 [17]. Assuming the computation units are fully-pipelined, we choose $C_{mult} = 2$ and $C_{add} = 1$ to make the generated arithmetic expressions more suitable for the deployed hardware. We report complexity for DNN computation as FLOPs, i.e. the sum of additions and multiplications, which are the most commonly used metrics to evaluate the computation complexity of DNN models.

Note that a more complicated and accurate cost model (e.g., dynamic cost) may better characterize the deployed hardware and it is left for future work.

Greedy Extraction Algorithm. We first experiment with a greedy extraction strategy. Q-gym computes the subgraph cost of each e-node using the cost model given in Eq.(3). For each e-class, Q-gym selects the e-node with the smallest subgraph cost. Then, the computation expression for each root e-class is determined.

Note that greedy extraction is not guaranteed to extract the graph with the minimum cost. That is because the e-nodes are selected locally in each eclass to minimize the subgraph cost and it fails to consider the potential computation sharing between e-nodes whose subgraphs may be overlapped.

ILP Extraction Algorithms. Alternatively, we can also formulate the selection of e-classes/e-nodes as an integer linear programming problem. The objective of ILP is to optimize the cost of arithmetic operations:

$$Minimize : f(x) = \sum_{m=0}^{M-1} cost(m) \cdot n_m \tag{4}$$

The constraints of the ILP are listed below:

(1) All the root e-classes ($root$) should be included as they represent the final computation results:

$$e_i = 1, \forall i \in \{i | i \in root\} \tag{5}$$

(2) Also, we need constraints to say that if an e-class i is included, then at least one of its child e-nodes is included. Otherwise, the term represented by the selected e-class cannot be reached.

$$e_i \leq \sum_{m \in child(i)} n_m \tag{6}$$

(3) For each e-node m selected, the child e-class of each operand must be included. This guarantees that the input to the e-node is not empty.

$$n_m \leq e_{m.r0}, \quad n_m \leq e_{m.r1} \tag{7}$$

To recap, the ILP optimization problem can be defined as minimizing Eq.(4) which is subject to constraints Eq.(5 - 7).

4.5 Pulsed e-graph Searching

When the size of a weight kernel is large, the e-graph can hardly reach saturation during the exploration phase due to hardware memory limitations. To resolve this issue, we propose an efficient searching algorithm by iteratively conducting *exploration* and *extraction*. Specifically, Q-gym expands the e-graph until the number of e-nodes or exploration steps reached its predefined limits (i.e., max_{enode}, $max\_explore\_iter$) and extracts the lowest-cost computation expressions using the ILP or greedy methods. Next, Q-gym will explore a new e-graph starting from the last generated computation expressions as shown in Algorithm 2. Our results show the pulsed searching algorithm can significantly reduce the computation cost throughout iterations (Section 6.1).

Algorithm 2 Pulsed e-graph Searching

```
Input: Input expressions $expr_i$; Set of rewrite rules $R$; Max
    node count $max_{enode}$; Max searching epochs $Epochs$
Output: Output expressions $expr_o$; Final e-graph $G'$; Selected
    node list $L'_{node}$
1: $cost' \leftarrow \infty$
2: $G \leftarrow$ Initialize_Egraph($expr_i$) // Initialize e-graph $G$
3: for $i$ in $0, ..., Epochs$ do
4:     $G, L_{node}, cost \leftarrow$ Equality_Saturation($G, R, max_{enode}$)
5:     $G \leftarrow Rebuild\_egraph(G, L_{node})$ // Remove all unselected
       nodes
6:     if $cost \leq cost'$ do $G', cost', L'_{node} \leftarrow G, cost, L_{node}$,
7: end for
8: $expr_o \leftarrow$ rebuild_expressions($G', L'_{node}$) // rebuild output ex-
   pressions from e-graph
9: return $expr_o, G', L'_{node}$
```

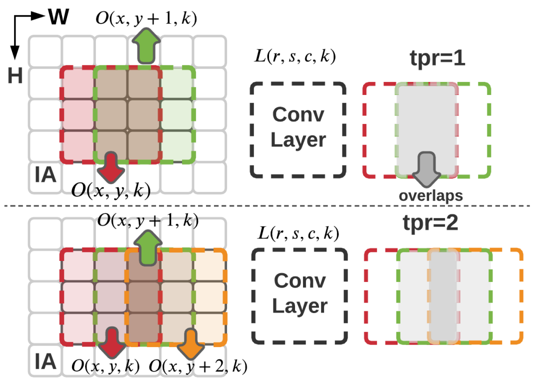

Figure 3: An illustration of temporal reuse search space. ***tpr*** **denotes computations that are explored together. The gray area is the overlapping area of inputs across timesteps. The red / green / orange squares denote the same convolutional layer in different time steps.**

Temporal search space. Q-gym also observes that when $R = S \geq 2$, given a convolutional layer, the input activation overlaps in temporal dimensions when computing $O(x, y, k)$ and $O(x, y + 1, k)$. That means computation reuse can also be applied between the computation of different O. As shown in Figure 3, if Q-gym searches two continuous convolution operations together, Q-gym can potentially find better computation expressions with lower cost in QNNs. We define the number of convolution operations searched together as timesteps tpr.

5 Q-GYM'S DOWNSTREAM TASKS

To exploit the efficient computation expressions generated by our compiler, we test Q-gym's performance on various downstream DNN applications.

5.1 Accelerating CNN on CPU and GPU Systems

The generated efficient expressions from Q-gym can be used to accelerate CNN inference on CPU/GPU. In this subsection, we detail how to use the generated expressions to yield speedup in DNN inference on CPU/GPU.

Mapping Expressions to Code. As shown in Eq.(1), the convolution operation iterates over 6 dimensions (H, W, R, S, C, K). In Q-gym, we unroll the inner for-loop over (R, S, C, K) dimensions. Specifically, the computation over (R, S, C, K) for weight layer L yields K output results (i.e., $O(x, y, k), k \in \{0, ..., K\}$). We replace the unrolled computations with Q-gym's efficient expressions through a common function call. The weight value is hardcoded in the function; so we don't need to load weights during the inference. The output K results from naïve loops over (R, S, C, K) and Q-gym's function call are identical. Note that the convolutions across input dimensions H, W are still repeated in each iteration, so the computation of different output pixels shares the same efficient expressions. No code generation compiler is used.

Combination of Q-gym with Parallelism Methods. By treating the computations along (R, S, C, K) as an atomic function, we can easily adapt the efficient computation expressions with (i) loop tiling, (ii) multithreading, and (iii) vectorization, respectively. In Figure 4, we illustrate how to combine the computation of our generated expressions with (i)(ii)(iii).

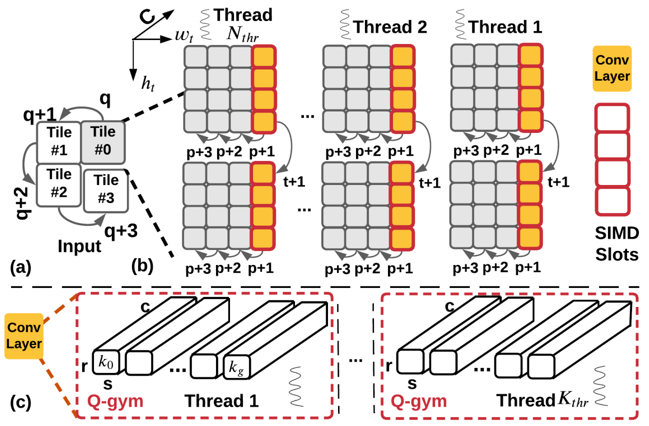

Figure 4: The parallelization mechanism in Q-gym with loop tiling, multithreading, and vectorization. (a) We split the input into small tiles for better data locality. q is an iterator over input tiles. (b) We apply SIMD vectorization along h_t of each tile to reduce the number of loops. We also split the tile along w_t dimension into N_{thr} groups and (c) K kernels into K_{thr} groups. Each group with is handled by different threads. $N_{thr} \times K_{thr}$ is the total number of threads. p, t are iterator over w_t and h_t. For simplicity, we set $r = s = 1$ in this figure.

(i) For loop-tiling, we first split the input activations into tiles with a size of $h_t \times w_t \times C$ to utilize data locality. We exhaustively scan all the possibilities of h_t and select the one that achieves the lowest latency. (ii) For multithreading, we split the w_t into N_{thr} threads to parallelize the computation over w_t. For GPU that allows 1024 threads, we further split the weight kernels along K dimensions into K_{thr} groups (Figure 4(b)). Dividing weight kernels along K dimensions can also reduce the instructions loaded to each core as there is no data dependency across weight kernels. (iii) Vectorization is applied on h_t dimensions to reduce the number of loops along h_t as shown in Figure 4. Each addition and multiplication in Q-gym's expressions are compiled into SIMD instructions that operate on different inputs and the same weights. We automate the vectorization by using the *'#pragma omp'* from openMP.

To evaluate Q-gym on CPU, we run two threads per physical core (e.g., 12 threads in total on an Intel i7-8700K processor) to ensure that each core has sufficient instruction-level parallelism (ILP) to fully exploit the available memory bandwidth ($N_{thr} = 12$ and $K_{thr} = 1$). The size of SIMD slots for CPU can be 4/8 (i.e., AVX-2/-512) for FP32. To evaluate Q-gym on GPU, we employ a larger $K_{thr} \times N_{thr}$ to exploit the maximum number of threads allowed on a GPU (1024 threads for NVIDIA 2080 GPU, detailed in Section 6.2). We implement the CNN acceleration software for Q-gym's efficient expressions using C and CUDA, by employing the parallelization methods shown in Figure 4. The code is compiled with *gcc -O3* on CPU and *nvcc* on GPU. The corresponding evaluation results on GPU and CPU are described in detail in Section 6.2.

5.2 Accelerating HE for DNNs

We identify that the low-cost computation arithmetic from Q-gym's compiler significantly reduces the evaluation time of DNN under homomorphic encryption (HE) as discussed in Section 2. The generated computation dataflow can be easily combined with different HE libraries (i.e.,HELib [4], SEAL [53]) by simply replacing the evaluation function f with the expressions compiled from Q-gym without any modifications to the encryption protocol. Note that Q-gym does not increase the depth of multiplication in HE; so it will not affect the correctness of the output result. We evaluate Q-gym's efficient expressions using two popular HE protocols (BGV [10] / BFV [16]). We also verify the correctness of the Q-gym's expressions using HELib/SEAL. HELib/SEAL with BGV/BFV schemes are also used to implement the CryptoNet/FastCryptoNet, respectively.

Note that the runtime of DNN inference under HE is linear to the number of HE operations. Specifically, there are four types of HE operations in DNN inference: (i) plaintext (PT)-ciphertext (CT) addition, (ii) ciphertext-ciphertext addition, (iii) plaintext-ciphertext multiplication, and (iv) ciphertext-ciphertext multiplication. Because the wall-clock time for executing different types of HE operations is library dependent and is highly relevant to the other settings (e.g., key size, machine settings), in this paper, we use the number of HE operations (HOPs) for comparison. This metric is commonly used for other privacy-preserving DNN methods, such as FastCryptoNet [13].

6 EVALUATION

In this section, we evaluate the performance of our compiler design in reducing QNN computations (Section 6.1) and the performance of Q-gym in accelerating various DNN applications (Section 6.2).

6.1 Algorithm Analysis

Experimental Setup. We implement the algorithm of Q-gym in Rust from egg [62], an open-source equality saturation library. During the e-graph extraction phase, we use Gurobi [1] as the ILP solver. We also reimplemented SumMerge in Python with loop parallelism as the baseline. The compilation time is tested on Intel Core i7-8700K CPU with 6 physical cores. This machine has 32/32 KiB of L1 instruction/data cache, 256KiB of L2 cache per core, and 12 MiB of shared L3 cache (Detailed in Table 3).

To measure the performance of Q-gym in reducing computations, we use the ratio between the number of reduced FLOPs and the total number of FLOPs ($-ops$) as the evaluation metric. For comparison purposes, we conduct experiments with Q-gym that use the greedy method during extraction (denoted as Q-gym$_{gd}$). Also, Q-gym without the pulsed searching algorithm is denoted as Q-gym$_{-p}$ (i.e., $max_{epoch} = 1$).

Without specification, we set $max_{enode} = 10^7$, epochs max_{epoch} to 10, $max\_explore\_iter = 8$, and set the temporal reuse steps (tpr) to 0 for Q-gym. For ILP extraction time limitation, we set it to $\sqrt{R \cdot S \cdot C \cdot K}$ seconds which is relative to the number of weight kernels in the convolutional layer.

Comparison of Reduced Operations. Figure 5 compares the reduction of FLOPs between Q-gym, Q-gym$_{gd}$, and SumMerge. We evaluate the algorithms on QNNs where the weights are generated synthetically using a uniform distribution. We assume none of the weights are 0 in the synthesized layers.

The result shows that Q-gym achieves significant computation reductions compared to SumMerge on quantized DNN with different Q. On average, Q-gym shows 19.1% / 15.5% more computation

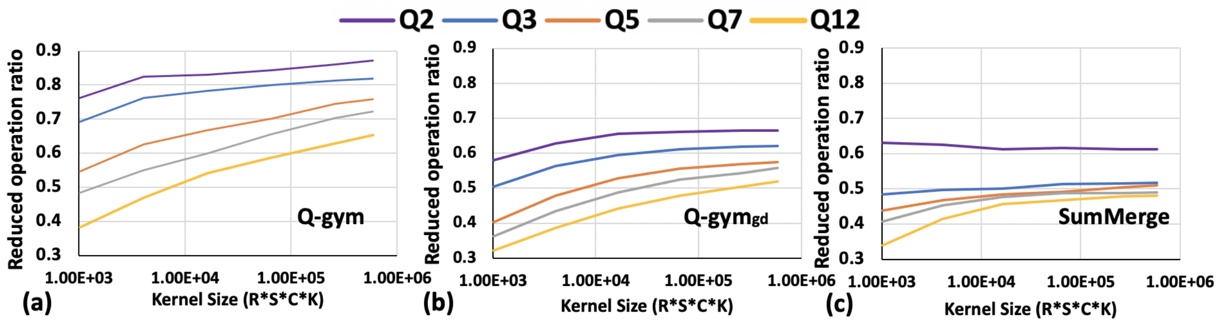

Figure 5: Comparison of reduced operations ($-ops$) between (a) Q-gym (b) Q-gym$_{gd}$ (c) SumMerge over different quantization schemes (Q) and layer sizes ($R \cdot S \cdot C \cdot K$).

Table 3: CPU/GPU configurations.

CPU	Intel i7-8700K, 6 physical Cores , 3.7 GHz, 1 sockets
GPU	NVIDIA 2080, 8GB main memory, 1024 maximum threads
L1 Cache L2 Cache L3 Cache TLB	32KiB 8-way I$, 32KiB 8-way D$, private 256KiB, 16-way, private 12MiB, 11-way, shared L1D 4-way 64 entries, L1I 8-way 128 entries STLB 12-way 1536 entries
DRAM	DDR4, 32GB, 2666MHz, 2 sockets, 6 channels per socket
Kernel	Linux 5.4.0
Software	GCC 7.1, PyTorch 1.4.0+cuDNN, CUDA 11.2, Rust 1.56.1

reduction than SumMerge in Figure 5. Q-gym also achieves a more substantial operation reduction compared to Q-gym$_{gd}$. As discussed in Section 4.4, using an ILP solver can find a better solution with a lower computation cost compared to a greedy heuristic.

Q-gym$_{gd}$ can also outperform SumMerge. On average, Q-gym$_{gd}$ shows 3.6% more computation reduction across all the kernels tested in Figure 5 (2.3% / 8.3% on average for $Q = 2$ / $Q = 3$). This is because e-graph exploration covers a larger search space compared to searching the common expressions between activation groups.

Compilation Time of Q-gym's Algorithm. We evaluate the time for Q-gym to compile a given weight layer into efficient computation expressions. Figure 6 shows the compilation time of Q-gym. With the increasing weight dimension and unique weights (Q), SumMerge takes a much longer compilation time than Q-gym. That is because SumMerge checks the overlapping of terms between all activation groups for computing the 'maxscore' (Section. 3). Also, the 'maxscore' is computed many times until no overlapping between activation groups is found.

With the increase of Q, the compilation time of Q-gym does not change a lot due to the e-node/explore step limitation (max_{enode}, $max\_explore\_iter$) and max epochs (max_epochs) for pulsed searching. To apply Q-gym in runtime compilation techniques, the users can choose Q-gym$_{gd}$ which is much faster compared to Q-gym and SumMerge.

Sensitivity to the Number of Unique Weights (Q) and Layer Size. With a growing number of weights, the performance of Q-gym also increases (Figure 5). When $Q = 12$, the computation reduction

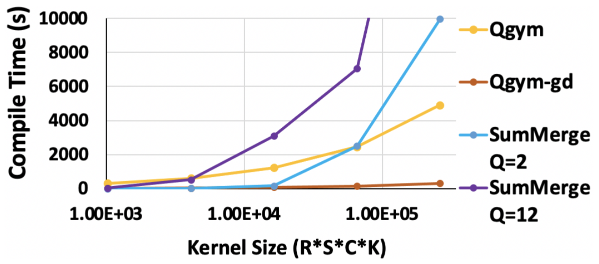

Figure 6: Compilation time for different weight sizes using SumMerge, Q-gym and Q-gym$_{gd}$.

ratio of Q-gym increases from 38.3% to 62.9% with the growing size of weights. For SumMerge, the performance eventually converges at ~50%. This means the greedy heuristic cannot fully exploit the computation reuse.

With the growing number of unique weights (Q), Q-gym always shows more computation reduction compared to SumMerge. Specifically, for $Q = 2$ / $Q = 3$ / $Q = 12$, Q-gym shows 21.3% / 27.6% / 11.2% more computation reduction compared to SumMerge. Note that SumMerge's performance decreases with the increase of Q. This is because the activation groups in SumMerge are factorized into smaller sets when Q is large, offering fewer opportunities for computation reuse.

Sensitivity to Temporal Reuse Steps (tpr). Figure 7 shows a sensitivity analysis of tpr on different QNN layers. With larger R, more inputs are overlapped across time steps and achieve lower computation cost when $tpr \geq 1$. For weight layers where (R, S) is $(3, 3)$, $(5, 5)$, $(7, 7)$, Q-gym where $tpr = 3$ shows 4.7, 6.0%, and 6.2% more computation reduction compares to Q-gym where $tpr = 0$. With the increasing tpr, the overlapping area of input activations also increases (Figure 3).

Comparison between Q-gym and Winograd. We also compare Q-gym with temporal reuse against Winograd, a commonly used technique that leverages fast FFT to reduce computation operations in convolutional layers. The downside of Winograd is that the transformation overhead is not negligible, i.e., the storage overhead of transformation matrices and the computation of input, weight transformations, and inversions. Also, Winograd has a theoretical boundary of 75% computation reduction (factor of 4). In this comparison, we did not take into consideration the extra computation for input, weight transformation, and inversion for Winograd. Thus, the baseline is stronger than what happens in practice.

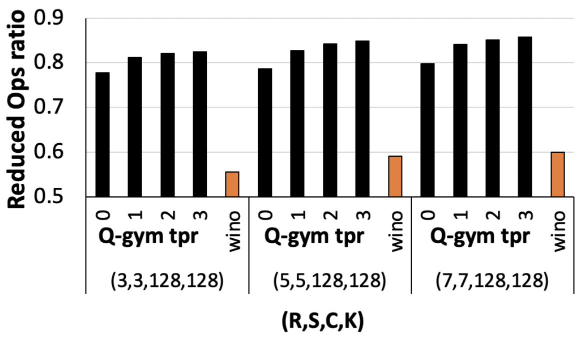

Figure 7: Sensitivity of tpr over the reduction of operations ($-ops$) when $Q = 3$. Comparison between Q-gym with temporal reuse to Winograd (denoted as 'wino'). We assume the input tile size ((H, W)) for Winograd is $(4, 4)$, $(12, 12)$, $(25, 25)$ for weights (R, S) $(3, 3)$,$(5, 5)$,$(7, 7)$, respectively. The corresponding α' are 2.25 / 10.51 / 19.61 in Winograd [31].

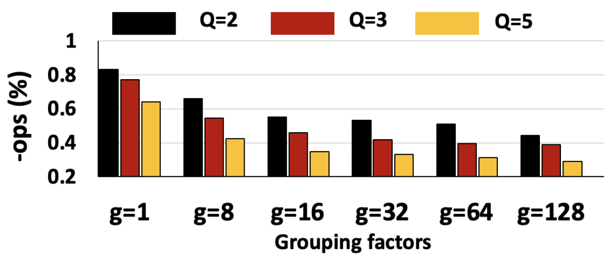

Figure 8: Sensitivity of operation reductions ($-ops$) over grouping factors g. The tested kernel size is $(3, 3, 128, 128)$ and we set $tpr = 0$.

On average, Q-gym shows 24.2% more computation reduction compared to Winograd as shown in Figure 7. Note that Q-gym can be combined with Winograd and we leave it as future work.

Analysis of Q-gym on group convolutions. We also verify if Q-gym is still working in group convolution layers [51, 59, 67]. Traditional convolution layers have a weight dimension (R, S, C, K) as given in Eq.(1). With group convolution where the grouping factor is g, the channel of each weight kernel would be $C_g = C/g$. The K kernels will be separated into g groups and each group has $K_g = K/g$ weight kernels, i.e., each group has a dimension of (R, S, C_g, K_g). When $g = C$, the convolution layer will become a *depth-wise convolution layer*. During group convolution, the input activation can be reshaped into (H, W, C_g, L_g). Each group of kernels (R, S, C_g, K_g) handles different groups of the input activations (R, S, C_g) in the same fashion given in Eq.(1). As such, Q-gym can still handle group convolutions when $R \times S \times C_g \times K_g > 1$. But with the increase of g, the search space decreases, giving Q-gym less opportunity to reuse computations. As shown in Figure 8, when $Q = 3$, the operation reduction drops from 77.1 % of traditional convolution to 38.9 % for depth-wise convolution layers.

Note that most of the depth-wise or group convolutions are implemented together with traditional convolutional layers [51, 59, 67]. Also, these models are typically not quantized due to a large accuracy drop.

Effectiveness of Pulse Searching Algorithm. Figure 9 shows the operation reductions over different epochs during Q-gym's pulse searching. On average, Q-gym with pulsed e-graph searching

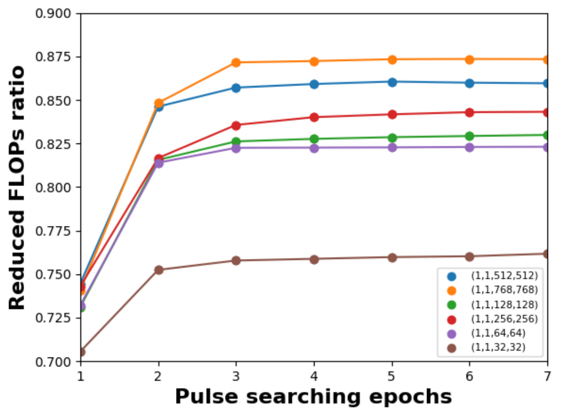

Figure 9: Computation reductions ($-ops$) over the pulsed searching epochs across different convolutional layers (R, S, C, K).

can reduce the computation cost by 9.9% compared to Q-gym$_{-p}$. For small kernels, e.g., $(1, 1, 32, 32)$, the computation reduction of pulse searching is limited due to the relatively small search space. For large kernels, pulse searching shows substantial performance improvement over Q-gym$_{-p}$. Note that from epochs 5 to 7, the number of operations is still reducing across different test cases. However, overall the number of reduced FLOPs becomes negligible compared to the total FLOPs.

6.2 Evaluation of Q-gym's Downstream Tasks

Experimental Setup. We evaluate the performance of Q-gym on an Intel Core i7-8700K CPU (The same machine as Section 6.1) and an NVIDIA 2080 GPU (1024 max threads), respectively. For CPU and GPU baselines, we choose to compare against the expressions generated from SumMerge. We also compare against the state-of-the-art DNN compiler (OneDNN [2])) for Intel CPU and Pytorch GPU (v1.4.1) [40] which has a carefully designed cuDNN/CUDA (v11.2) back-end for performance purposes. All the weights and input activations are 32-bit floating-point numbers. For all comparisons, we set the batch size to be 1 to accommodate the common configurations used in real-time systems. The implementation methods of Q-gym for CPU/GPU are described in Section 5.

Performance of Q-gym on CPU. Figure 10 shows the per-layer speedup with different Qs on the CPU. All experiments assume an input of dimensions of $H = W = 48$. We enable multithreading (MT) for both Q-gym and OneDNN. For MT implementation in Q-gym, we set $N_{thr} = 12$ and $K_{thr} = 1$ across all layers tested to maximize CPU resource utilization. For OneDNN settings, we scan the number of threads from 1 to 12 for each layer and select the best-performed setting. For SumMerge, we apply the same MT implementation as Q-gym (Section 5) with the computation expressions generated from SumMerge. For vectorization, we unroll the loop along H dimensions (unrolled factor = 8).

When $Q \leq 3$, Q-gym shows 2.56× / 1.83× speedup compared to OneDNN and SumMerge on average across different convolution layers. Note that $Q \leq 3$ are common settings for QNNs (Table 4). We observe that when the Q gets larger, the speedup of Q-gym

Table 4: Comparison between Q-gym and SumMerge/OneDNN on CPU performance. ♭TTQ is a sparse and quantized model (one of the three values is 0. ♮ VGG-small is a derivative architecture based on VGG [56]. ♯ '-Ops Gap' denotes the gap of computation reduction ratio ($-ops$) between Q-gym and SumMerge. The models can be found in the repository [64, 69].

Description				Accuracy		v.s. SumMerge [43]		v.s. OneDNN [2]	v.s. PyTorch [40]
DNN Arch	Q	Dataset	Quantized method	Full	Quantized	♯Ops Gap (%)	CPU MT Speedup	CPU MT Speedup	GPU MT Speedup
AlexNet [30]	2	ImageNet [14]	BWN [47]	59.7	55.7	27.1	1.83	1.76	1.03
AlexNet	3	ImageNet	♭TTQ [70]	59.7	55.2	16.7	1.45	1.84	1.42
ResNet-20 [21]	3	CIFAR-10 [28]	ProxQuant [6]	91.9	91.3	27.0	1.21	2.31	1.86
♮VGG-small [11]	4	CIFAR-10	LQ-Nets	93.8	93.5	21.9	1.40	1.32	0.95
ResNet-18	2	CIFAR-100 [29]	LS-1 [42]	77.8	75.8	27.6	1.55	1.81	1.56
ResNet-18	3	CIFAR-100	LS-T [42]	77.8	76.5	29.8	1.52	1.59	1.20

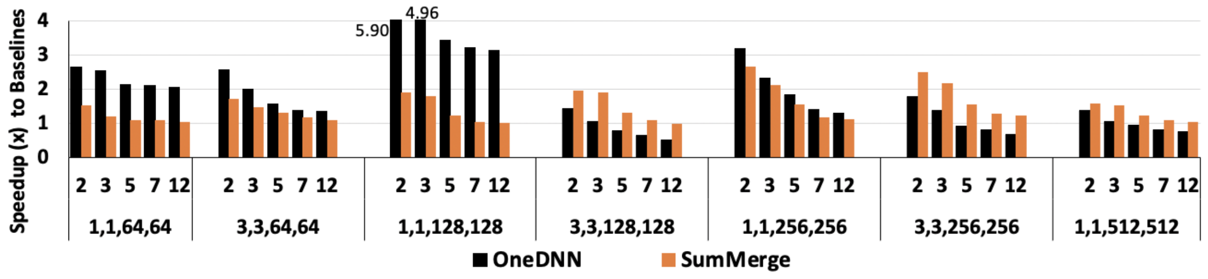

Figure 10: Per-layer speedup (higher is better) comparison on CPU. The tuple (R,S,C,K) denotes the size of different convolution layers. For OneDNN implementation, we use the official performance profiling tool [3].

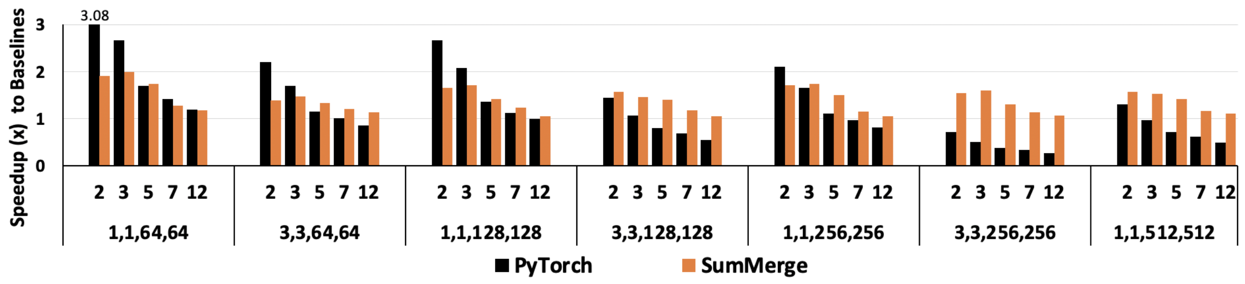

Figure 11: Per-layer speedup (higher is better) comparison on GPU.

over SumMerge decreased due to the relatively lower computation reduction of Q-gym in large Q. On real-world DNN models shown in Table 4, Q-gym achieves 1.49× / 1.77× speedup over SumMerge / OneDNN with an average accuracy loss of 2.12% relative to the full precision models.

We also compare the performance of Q-gym on CPU to its theoretical peak as shown in Figure 12. Specifically, we assume the two VPUs (SIMD width is 8) in each core are fully pipelined running on 3.7GHz. All 6 physical cores are enabled. As shown in Figure 12, when the kernel size is small, the performance ratio is low. That is because the threads' creation and termination overhead are relatively large compared to the computation overhead. For large kernels, the CPU utilization is also decreasing, this is because large convolution kernels yield a large number of computation expressions and the data locality between expressions are getting worse.

Performance of Q-gym on GPU. Figure 11 shows the per-layer speedup on GPU over PyTorch and SumMerge. Regarding the settings of Q-gym's parallelization method (Figure 4), we choose $K_{thr} = 8192/C$ to limit the size of instructions loaded to the GPU cache. We also set $N_{thr} = 1024/K_{thr}$ to maximize the usage of GPU resources. All runs assume an input of dimensions $H = W = 112$.

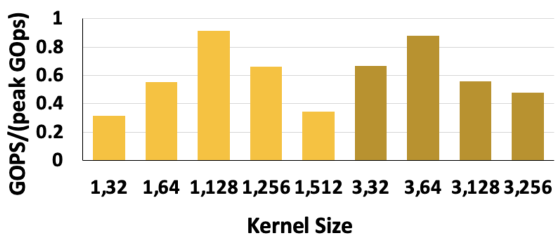

Figure 12: Per-layer GOPS and peak GOPS on CPU across different convolutional layers. The tuple (R, C) denotes the size of the kernel (R, S, C, K) where $R = S$, $C = K$.

Q-gym shows an average speedup of 1.92× / 1.64× compared to PyTorch when $Q = 2$ / $Q = 3$. Also, Q-gym shows an average speedup of 1.63× when $Q \leq 3$ over the computation expression from SumMerge. With the increasing size of weight layers, the performance of Q-gym drops faster than PyTorch due to the larger instruction size and memory footprint. Also, for the convolution layer where H and W are small, Q-gym parallelizes on the K dimension to maximize the usage of GPU resources. This will reduce the search space of Q-gym's compilation phase and reduce the opportunity for computation reuse.

Performance of Q-gym for HE. As mentioned in Section 5.2, we compare DNN inference under HE using the homomorphic operations (HOPs) following previous works [13]. Due to the expensive computation cost of HOP on real hardware, the inference time of HE applications is linear to the number of HOPs regardless of the hardware or software parallelization schemes.

For the baseline comparison, we choose CryptoNet [19], a dense DNN model that is trained on MNIST [32]. FastCryptoNet [13], a follow-up work of CryptoNet that leverages sparsity to bypass HOPs. We use a trained dense QNN and a trained sparse QNN to compare with the baselines separately. The model architectures are the same as FastCryptoNet (a slight variant of CryptoNet). We apply Q-gym to compile the models for HOPs comparison.

As is shown in Table 5 and 6, Q-gym achieves 59.5% and 22.9% HOPs reduction relative to CryptoNet and FastCryptoNet. Meanwhile, the models trained using INQ achieve almost the same accuracy (-0.11%/-0.02%) compared to CryptoNet / FastCryptoNet. Besides, Q-gym reduces the PT-CT multiplication by 10.8× / 1.87× which is the computation bottleneck of DNN inference under HE.

Deploying Q-gym on Small Devices. During code generation, Q-gym unrolls the inner convolution loops and replaces them using efficient computation expressions. As such, the compiled binary size is larger than the one using naive loops. For small architectures (e.g., mobile devices) with a small instruction cache, the large number of instructions may decrease the performance of Q-gym. On those devices, Q-gym can be employed using an alternative option by leaving the computation expressions in the "interpreted form", i.e., we load the expressions during inference and compute the output accordingly, which is the way implemented in SumMerge [43]. Q-gym would then still be more efficient than the "interpreted" SumMerge due to better performance in reducing operations. On our experiment architectures, Q-gym's deploying method is better and we use the same way to evaluate SumMerge in this paper.

7 RELATED WORK

This section discusses previous works related to our study.

Quantized Neural Networks. Neural network quantization [64, 68, 70] is a promising technique to compress and accelerate DNNs. The reduced bit width enables low-bit width multiplication [54] to speed up the inference. Jung et al. introduced parameterized quantization intervals and optimized them to minimize task loss [24]. Han et al. used k-means clustering as a method of quantization to share weights [20]. Zhou et al. proposed a rule-based non-uniform quantization by leveraging a logarithmic distribution [68]. [64, 70] optimize the quantization clipping range during model training. Beyond the computer vision tasks, quantization can also be applied to BERT [55, 65] for NLP tasks. Q-gym can be applied to all these quantization models to reduce the computations of QNNs.

Equality Saturation Applications. The key idea of Q-gym is equality saturation [57, 60]. It has also been widely used in various domains, such as simplifying CAD design, rewriting DNN architectures, improving the accuracy of floating-point expressions and doing semantic code search [37, 39, 44, 63]. In this paper, we contribute to the iterative searching algorithm and elaborated search space to better reduce the computation in DNN. This is also the first work that applies equality saturation to accelerate homomorphic encryption for quantized DNNs inference.

Table 5: A breakdown of HOPs for each layer between Q-gym and CryptoNet. The dense model for Q-gym uses 2-bit INQ [68] where $Q = 4$. The model architecture (same as FastCryptoNet) for Q-gym is a slight variant of CryptoNet. We set $tpr = 0$ for all layers' compilation. Fully-connected (FC) layers can be treated as special convolutional layers (conv) where R, S, H, W are 1. 'Act' denotes activation layers.

CryptoNet					
Layer	Hops	PT-CT Adds	CT-CT Adds	PT-CT Mults	CT-CT Mults
Conv-1	42,757	845	20,956	20,956	-
Act-1	845	-	-	-	845
Pool-1	6,845	-	6,845	-	-
Conv-2	309,905	1,250	154,350	154,350	-
Pool-2	8450	-	8450	-	-
FC-1	241192	100	120546	120546	-
Act-2	100	-	-	-	100
Fc-2	1990	10	990	990	-
Total	612,129	2,205	312,137	296,842	945
Accuracy	99.17				
Q-gym					
Conv-1	25,350	1,690	15,717	7,943	-
Act-1	5,070	845	1,690	1,690	845
Pool-1	6,845	-	6,845	-	-
Conv-2	118,975	1,250	105,225	12,500	-
Pool-2	-	-	8450	-	-
FC-1	82097	100	76997	5000	-
Act-2	600	100	200	200	100
Fc-2	477	10	427	40	-
Total	247,864	3,995	215,551	27,373	945
Accuracy	99.06				

Table 6: Comparison between Q-gym and FastCryptoNet. As FastCryptoNet is a sparse model $sp = 6.63\%$ and applied 2-bit INQ quantization [68], Q-gym uses a sparse and quantized model ($Q = 4$) for evaluation accordingly. Sparse ratio sp is set to be the same as Fast CryptoNet.

FastCryptoNet					
Layer (sp)	Hops	PT-CT Adds	CT-CT Adds	PT-CT Mults	CT-CT Mults
Conv-1	8,619	1,690	3,042	3,887	-
Act-1	5,070	845	1,690	1,690	845
Pool-1	6,845	-	6,845	-	-
Conv-2	22,950	1,250	10,850	10,850	-
Pool-2	8450	-	8450	-	-
FC-1	14,354	100	7,077	7,177	-
Act-2	600	100	200	200	100
Fc-2	306	10	148	148	-
Total	67,194	3,995	38,302	23,932	945
Accuracy	98.73				
Q-gym					
Conv-1	6,760	1,690	2,704	2,366	-
Act-1	5,070	845	1,690	1,690	845
Pool-1	6,845	-	6,845	-	-
Conv-2	18,150	1250	10550	6350	-
Pool-2	8450	-	8450	-	-
FC-1	5,724	100	3383	2241	-
Act-2	600	100	200	200	100
Fc-2	192	10	131	51	-
Total	51,791	3,995	33,953	12,798	945
Accuracy	98.71				

Acceleration of Homomorphic Encryptions for DNNs. Since the first Fully HE scheme was proposed by Gentry et al. [18]. Many acceleration methods for FHE have been proposed, such as Leveled Somewhat HE (SWHE) [7, 9, 10]. Many advances [8, 16, 25] leverage SWHE to do privacy-preserving DNN inference. CryptoNet [19] is the first work to use SWHE for DNN inference. FastCryptoNet [13] leverages DNN model sparsity to accelerate CryptoNet. Many other works have aimed at improving the computation efficiency of non-linear layers [34] or computing SWHE in a SIMD fashion [25]. These methods are orthogonal to the computation reduction generated from Q-gym and can be combined to further improve the efficiency of HE DNN inference. Note that Q-gym does not tamper with the security of HE. The input is kept private all the time and no information is leaked.

8 CONCLUSION

This paper proposes Q-gym, a DNN framework that accelerates quantized DNNs by exploiting the weight repetition characteristic. Q-gym proposes a compiler and a set of acceleration schemes for various DNN applications. For the compiler, Q-gym employs the idea of equality saturation and represents the DNN computation into an e-graph and applies a set of rewrite rules to explore equivalent expressions. After the exploration, we extract the lowest-cost computation expressions from the e-graph by formulating the selection of nodes in the e-graph into an integer linear programming problem. By iteratively conducting exploration and extraction and leveraging a temporal search space, we can further reduce the computation operations compared to previous works.

Leveraging the reduced computation, Q-gym can accelerate a set of DNN applications. We build QNN inference kernels on CPU and GPU with carefully designed parallelization schemes and combine the efficient expressions from Q-gym with multi-threading, vectorization, and loop tiling. Q-gym also proposes to combine the reduced expressions into DNN inference flow under homomorphic encryption. Experiments show significant speedups and operation reductions in those applications compared to the state-of-the-art methods.

REFERENCES

[1] [n.d.]. Gurobi Optimization - The Fastest Solver. https://www.gurobi.com

[2] [n.d.]. oneAPI Deep Neural Network Library (oneDNN). https://github.com/oneapi-src/oneDNN

[3] [n.d.]. OneDNN Performance Profiling Tool. https://github.com/oneapi-src/oneDNN/blob/master/examples/performance_profiling.cpp

[4] 2021. HELib. Online: https://github.com/homenc/HElib. EPFL-LDS.

[5] John O Awoyemi, Adebayo O Adetunmbi, and Samuel A Oluwadare. 2017. Credit card fraud detection using machine learning techniques: A comparative analysis. In *2017 International Conference on Computing Networking and Informatics (ICCNI)*. IEEE, 1–9.

[6] Yu Bai, Yu-Xiang Wang, and Edo Liberty. 2018. Proxquant: Quantized neural networks via proximal operators. *arXiv preprint arXiv:1810.00861* (2018).

[7] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. 2016. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. Cryptology ePrint Archive, Report 2016/510. https://ia.cr/2016/510.

[8] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. 2013. Improved security for a ring-based fully homomorphic encryption scheme. In *IMA International Conference on Cryptography and Coding*. Springer, 45–64.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Report 2011/277. https://ia.cr/2011/277.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[11] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5918–5926.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[13] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. 2018. Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference. *CoRR* abs/1811.09953 (2018). arXiv:1811.09953 http://arxiv.org/abs/1811.09953

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

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

[16] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[17] Agner Fog et al. 2011. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering* 93 (2011), 110.

[18] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[19] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*. PMLR, 201–210.

[20] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[22] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 674–687. https://doi.org/10.1109/ISCA.2018.00062

[23] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures. *CoRR* abs/1607.03250 (2016). arXiv:1607.03250 http://arxiv.org/abs/1607.03250

[24] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. 2019. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4350–4359.

[25] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1651–1669.

[26] Georgios A Kaissis, Marcus R Makowski, Daniel Rückert, and Rickmer F Braren. 2020. Secure, privacy-preserving and federated machine learning in medical imaging. *Nature Machine Intelligence* 2, 6 (2020), 305–311.

[27] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. 2022. Caviar: an e-graph based TRS for automatic code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 54–64.

[28] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n.d.]. CIFAR-10 (Canadian Institute for Advanced Research). ([n. d.]). http://www.cs.toronto.edu/~kriz/cifar.html

[29] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n.d.]. CIFAR-100 (Canadian Institute for Advanced Research). ([n. d.]). http://www.cs.toronto.edu/~kriz/cifar.html

[30] imagenet Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[31] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4013–4021.

[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[33] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.

[34] Qian Lou and Lei Jiang. 2019. She: A fast and accurate deep neural network for encrypted data. *Neural Information Processing Systems* (2019).

[35] Christos Louizos, Max Welling, and Diederik P. Kingma. 2018. Learning Sparse Neural Networks through L_0 Regularization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1Y8hhg0b

[36] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*. 5058–5066.

[37] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[38] Charles Gregory Nelson. 1980. *Techniques for program verification.* Stanford University.

[39] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. https://doi.org/10.1145/2813885.2737959

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[41] Johan Perols. 2011. Financial statement fraud detection: An analysis of statistical and machine learning algorithms. *Auditing: A Journal of Practice & Theory* 30, 2 (2011), 19–50.

[42] Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. 2020. Least squares binary quantization of neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 698–699.

[43] Rohan Baskar Prabhakar, Sachit Kuhar, Rohit Agrawal, Christopher J. Hughes, and Christopher W. Fletcher. 2021. SumMerge: An Efficient Algorithm and Implementation for Weight Repetition-Aware DNN Inference. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/3447818.3460375

[44] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

[45] W Nicholson Price and I Glenn Cohen. 2019. Privacy in the age of medical big data. *Nature medicine* 25, 1 (2019), 37–43.

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[47] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542.

[48] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[49] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015), 91–99.

[50] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[52] Alexander Schrijver. 1998. *Theory of linear and integer programming.* John Wiley & Sons.

[53] SEAL 2018. Microsoft SEAL (release 3.0). http://sealcrypto.org. Microsoft Research, Redmond, WA.

[54] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 764–775.

[55] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8815–8821.

[56] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *ICLR* (2015).

[57] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based translation validator for LLVM. In *International Conference on Computer Aided Verification*. Springer, 737–742.

[58] Mengshu Sun, Zhengang Li, Alec Lu, Yanyu Li, Sung-En Chang, Xiaolong Ma, Xue Lin, and Zhenman Fang. 2022. FILM-QNN: Efficient FPGA Acceleration of Deep Neural Networks with Intra-Layer, Mixed-Precision Quantization. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 134–145. https://doi.org/10.1145/3490422.3502364

[59] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

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

[61] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. https://doi.org/10.14778/3407790.3407799

[62] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

[63] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021).

[64] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*. 365–382.

[65] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020. TernaryBERT: Distillation-aware Ultra-low Bit BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 509–521. https://doi.org/10.18653/v1/2020.emnlp-main.37

[66] Wentai Zhang, Hanxian Huang, Jiaxi Zhang, Ming Jiang, and Guojie Luo. 2018. Adaptive-precision framework for SGD using deep Q-learning. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[67] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 6848–6856.

[68] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044* (2017).

[69] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* abs/1606.06160 (2016). http://arxiv.org/abs/1606.06160

[70] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained ternary quantization. *arXiv preprint arXiv:1612.01064* (2016).