SumMerge: An Efficient Algorithm and Implementation for Weight Repetition-Aware DNN Inference

Rohan Baskar Prabhakar*
Princeton University
rohanbp@princeton.edu

Sachit Kuhar*
Indian Institute of Technology, Guwahati
kuhar@iitg.ac.in

Rohit Agrawal
NVIDIA
roagrawal@nvidia.com

Christopher J. Hughes
Intel
christopher.j.hughes@intel.com

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
cwfletch@illinois.edu

* Lead authors.

ABSTRACT

Deep Neural Network (DNN) inference efficiency is a key concern across the myriad of domains now relying on Deep Learning. A recent promising direction to speed-up inference is to exploit *weight repetition*. The key observation is that due to DNN quantization schemes—which attempt to reduce DNN storage requirements by reducing the number of bits needed to represent each weight—the same weight is bound to repeat many times within and across filters. This enables a weight-repetition aware inference kernel to factorize and memoize out common sub-computations, reducing arithmetic per inference while still maintaining the compression benefits of quantization. Yet, significant challenges remain. For instance, weight repetition introduces significant irregularity in the inference operation and hence (up to this point) has required custom hardware accelerators to derive net benefit.

This paper proposes SumMerge: a new algorithm and set of implementation techniques to make weight repetition practical on general-purpose devices such as CPUs. The key idea is to formulate inference as traversing a sequence of data-flow graphs *with weight-dependent structure*. We develop an offline heuristic to select a data-flow graph structure that minimizes arithmetic operations per inference (given trained weight values) and use an efficient online procedure to traverse each data-flow graph and compute the inference result given DNN inputs. We implement the above as an optimized C++ routine that runs on a commercial multicore processor with vector extensions and evaluate performance relative to Intel's optimized library oneDNN and the prior-art weight repetition algorithm (AGR). When applied on top of six different quantization schemes, SumMerge achieves a speedup of between 1.09×-2.05× and 1.04×-1.51× relative to oneDNN and AGR, respectively, while simultaneously compressing the DNN model by 8.7× to 15.4×.

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460375

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; • **Theory of computation** → *Vector / streaming algorithms*; • **Software and its engineering** → *Software performance.*

KEYWORDS

Inference, Deep Neural Networks, Convolutional Neural Networks, Weight Quantization, Weight Repetition

ACM Reference Format:
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 *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460375

1 INTRODUCTION

Deep Neural Networks (DNNs) are used broadly, being the state-of-the-art technique for tasks ranging from autonomous driving to healthcare to entertainment and more [10, 23, 27, 28, 31]. As a result, DNN *inference*, or evaluation, is becoming a dominant task in a range of deployments, ranging from cloud to mobile to IoT [20, 36]. Improving the efficiency of inference in these various contexts can have a large real-world impact.

In this vein, recent work by Hegde et al. [22] proposed to exploit a phenomenon called *weight repetition* to dramatically improve inference efficiency. The key computational primitive for DNNs is a dot product between vectors of weights and inputs/intermediate values. The insight is that when the maximum number of unique values in one vector is smaller than the vector length, some values *will* appear multiple times within the vector. Hegde et al. [22] exploited these repetitions in weight vectors—which are fixed at inference time – hence the name weight repetition. For example, in the dot product between weights $\{x, y, x\}$ and activations $\{a, b, c\}$,

one with knowledge of the repetition pattern can refactor the computation into a sum-of-product-of-sums (i.e., $x(a+c)+yb$) to reduce the number of operations, especially multiplies. (By convention, we will use letters at the beginning of the alphabet to represent activations and letters at the end to represent weights.) Hegde et al. [22] show how more sophisticated algorithms can simultaneously eliminate multiplies, adds and memory accesses, and reduce the DNN model's size.

Importantly, weight repetition complements existing techniques to improve inference efficiency, such as weight sparsity (e.g., [18, 19, 30, 38]) and quantization (e.g., [8, 12–14, 18, 32, 33, 37, 39–42]). That is, more aggressive quantization by definition gives us fewer unique weights which means more repetitions per weight. Likewise, weight sparsity may be viewed as a special case of weight repetition. Sparsity exploits repetitions of the zero-valued weight while weight repetition can exploit repetitions in any (zero or non-zero) weight.

1.1 This Paper

Although the potential benefits of exploiting weight repetition are great, we face several challenges before adopting these techniques in practice. The crux of the problem is that while weight repetition reduces the amount of arithmetic per inference, it adds significant irregularity to the computation. Case in point, the techniques in Hegde et al. [22] call for data-dependent indirection and branching, and require a custom accelerator to derive net benefit. Yet, the bulk of DNN inference in the real world is not performed on accelerators, but rather CPUs and GPUs [2, 7, 20, 36], where irregular computations are difficult to orchestrate for peak performance. Not to mention, most mobile devices use relatively old hardware to save cost [36], making accelerator-based innovations take even longer to reach the market.

This paper addresses these challenges by proposing *SumMerge*: a novel algorithm to improve on the arithmetic savings from weight repetition, and implementation techniques to improve the regularity of weight repetition exploitation. SumMerge enables the effective deployment of weight repetition-aware inference on general purpose devices such as CPUs.

As discussed above, DNN computations are made up of multiple, independent dot products. The key idea in SumMerge's algorithm is to formulate the set of dot products as a shared data-flow graph where repeated sub-computations within and across dot products can be computed once, memoized and reused later. This is related to common subexpression elimination in compilers; here, however, this is not a purely symbolic process, as the "expressions" form according to repetition of weight *values*. Offline (statically), SumMerge performs a search that computes intersections within and across weight vectors involved in different dot products to discover redundant sub-computations, and adds those sub-computations as new unique vertices in the graph. For example, given two dot products $x(a+c)+yb$ and $xw+z(a+c)$, $a+c$ is a common sub-computation and becomes a new vertex with two outgoing edges (indicating reuse). This process is hierarchical. For example, the terms a and c in the example may themselves be the result of many additions which can be further split into additional vertices and shared across even more dot products. Finally, online, we traverse the data-flow graph once to perform the logical work of multiple dot products; reducing the number of additions, multiplications, memory accesses and model size (since the single graph now encodes multiple weight vectors).

On the implementation side, we design the first software-based kernel for weight repetition that runs on general-purpose devices. A key idea in the implementation is that while weight repetition-aware kernels are highly irregular in general, there is significant regularity along certain dimensions. Continuing the above example, $x(a+c)+yb$ results in irregular computation because inputs that are summed in parentheses (e.g., a and c) imply indirections and the positions of parentheses imply branches. Yet, the *pattern* of this irregularity—i.e., $x(?+?)+y?$—repeats in multiple dot products, e.g., due to sliding window filters in convolutions or the multiple weight vectors in fully-connected layers. From a software point of view, DNN inference is a nested series of loops performing sequences of dot products; we pull the irregularity into the outer loops, and capture the regular patterns in the innermost loops. This allows us to leverage performance-oriented features on general-purpose devices such as vector extensions.

Putting it all together, our kernel is written from scratch in C++ and outperforms the state-of-the-art industry library oneDNN [5], for performing dense CNN inference on CPUs, and the prior-art weight repetition algorithm [22] (implemented and optimized in C++ by us). We implement our kernel on top of multiple existing quantization schemes to simultaneously reap performance improvement and model compression from repeated weights.

Contributions. This paper makes the following contributions:

(1) We propose SumMerge, a new algorithm for performing weight repetition-aware inference. To the best of our knowledge, SumMerge is the most efficient weight repetition-aware inference procedure both in terms of arithmetic reduction and model compression.
(2) We design, implement and optimize a software-only kernel for SumMerge which runs on CPUs. To the best of our knowledge, this is the first work to assess the feasibility of exploiting weight repetition on general-purpose devices. We present multiple new techniques to improve efficiency on these devices.
(3) We evaluate our prototype against oneDNN, Intel's state-of-the-art library for DNN inference [5], and the prior-art weight repetition scheme (called AGR [22]) running on six different quantization schemes. SumMerge achieves a speedup of between 1.09×-2.05× and 1.04×-1.51× relative to oneDNN and AGR, respectively, while simultaneously compressing the DNN model by 8.7× to 15.4×.

While we evaluate on CNNs, many of our techniques generalize to other layer types, whose core operations are dot products (such as MLPs). Moreover, while we evaluate using a CPU platform, we expect our techniques to be useful on other platforms, such as GPUs.

Open source. We have open-sourced our prototype kernels of both SumMerge and AGR to the community here: https://github.com/rohan-bp/summerge.

2 BACKGROUND

2.1 Convolutional Neural Network Inference

While our techniques apply to any DNN-based computation that can be formulated as a matrix multiply, we focus on and explain ideas in terms of convolutional layers in Convolutional Neural Networks (CNNs) due to their ubiquity and high computational cost relative to other layer types [11, 21, 26, 34, 35]. A convolutional layer consists of applying K 'sliding-window' filters of dimension $R \times S \times C$ to a $W \times H \times C$ dimension input, generating a $(W-R+1) \times (H-S+1) \times K$ output. Each layer's output becomes the next layer's input. (Throughout the paper, we use $\times$ to refer to dimensions, and $*$ to refer to scalar multiplication.) For each spatial position in the input and each filter, this corresponds to a dot product between a vector of weights (i.e., this layer's filter values) and a vector of input activations (i.e., the previous layer's output values).

For a layer with unit stride and 0 bias, this can be expressed as:

$$\mathsf{O}[(k,x,y)] = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \mathsf{F}[(k,c,r,s)] * \mathsf{I}[(c,x+r,y+s)] \quad (1)$$

for $0 \le k < K, 0 \le x \le W-R, 0 \le y \le H-S$, where O, I and F are output activations, input activations and filter weights, respectively. As is the case with other works targeting inference (e.g., [17, 22]), we assume a batch size of 1, noting that the techniques described extend to larger batch sizes as well.

2.2 Quantization and Activation Group Reuse

There has been much recent work on *quantization schemes* that reduce the number of bits needed to represent the weights (and activations) of DNNs while preserving model accuracy [8, 12, 13, 18, 32, 33, 37, 39–42]. This decreases the memory footprint of DNNs and can also be exploited to increase ALU throughput (e.g., for SIMD implementations, we can fit more elements in a fixed bit-width vector). Importantly, quantization implies that the DNN weights are limited to a smaller number U of distinct values. One effect of decreasing U is that the number of times a given weight value appears in each layer (a.k.a. weight repetition) goes up.

Activation Group Reuse (AGR) is a weight-aware inference algorithm proposed by Hegde et al. [22] that takes advantage of weight repetition to optimize DNN inference. AGR exploits weight repetition to reduce the number of arithmetic operations and memory reads, and also compresses the DNN model size, as described below. Following prior quantization schemes, AGR assumes all filters in a layer share the same U. For inference, the DNN weights are fixed, known well ahead of time, and typically used for a large number of inferences; thus, we can afford to do extensive weight-centric pre-processing if it gives a boost to inference performance and/or efficiency.

2.2.1 Step 1: Exploiting Intra-Filter Weight Repetition. For a filter within a given CNN layer, AGR reduces multiplications by factoring the weights in the filter and computing the dot product as a two-level expression. Figure 1 depicts the factored version of the dot product between Input, a vector of input activations, and Filter, a vector of weights. In this example, the weights in the filter are quantized such that they are either x or y; this would be the case if the network had $U = 2$.

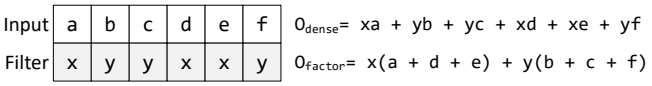

Figure 1: Exploiting intra-filter repetition by factoring weights within each dot product (Section 2.2.1). $\mathrm{O}_{\mathrm{dense}}$ **and** $\mathrm{O}_{\mathrm{factor}}$ **represent the original and factored dot products, respectively.**

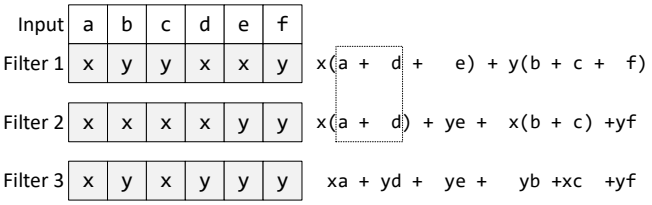

Figure 2: Exploiting inter-filter repetition, i.e., activation group reuse, by recursively factoring across dot products (Section 2.2.2). The terms $a+d$ and $b+c$ can be reused. Note, AGR does not reuse partial products, e.g., ye or yf [22]. The filter expressions have activations aligned to emphasize the recursive structure: AGR looks within activation groups to build sub-activation groups, etc.

Factoring allows us to first sum activations in Input that are to be multiplied by the same weight in Filter, then multiply the sum by the corresponding weight. Repeating this procedure for each unique weight and accumulating these products (that are summed first) permits computing the dot product more efficiently by avoiding superfluous multiplications. Following notation from Hegde et al. [22], input activations that are to be summed locally such as (a, d, e) and (b, c, f) (in Figure 1), are referred to as *activation groups*. Adopting this strategy during inference reduces the number of multiplications in each dot product to at most U – the number of unique weights used by the quantization scheme. The consequent reduction in arithmetic operations is substantial, e.g., using AGR with the TTQ scheme [42] reduces the number of multiplies from hundreds or thousands per dot product to ≤ 3.

2.2.2 Step 2: Exploiting Inter-Filter Weight Repetition. As can be seen from Equation 1, for a given spatial position, the input is invariant across all the dot products for the K different filters. This allows AGR to recursively factor groups of dot products and exploit overlap between activation groups across two or more filters, further reducing the number of arithmetic operations. See Figure 2 for an example, where Filter from Figure 1 now appears as Filter 1. Here, inputs (a, d, e) form an activation group that will be locally summed for Filter 1. Recursively looking at the activations (a, d, e) in Filter 2, we find that (a, d) are multiplied by the same weight in Filter 2, so $(a + d)$ is computed just once and used in the computation for Filter 1 and Filter 2. This is shown with the dashed box in Figure 2. Hegde et al. refers to this as *activation group reuse* (AGR).

AGR applies this idea recursively to find overlap across groups of G filters, where G is chosen offline. Specifically, it looks for "sub-activation groups" for Filter 2 *within* each activation group of

Filter 1, "sub-sub-activation groups" for Filter 3 *within* each sub-activation group Filter 2, so on to Filter G. We call (sub-*)activation groups of Filter G—i.e., in Filter 3 in Figure 2—the *innermost activation groups*. These innermost activation groups are computed first and reused to compute Filter 1 to Filter $G-1$, the groups in Filter $G-1$ are used in Filter 1 to Filter $G-2$, and so on. This strategy enables a simple offline recursive procedure to determine (sub*-)activation group structure and metadata, which is then used during the online step to perform optimized inference.

The computation savings depends on choosing the right G given the number of unique weights U and the filter dimensions $R \times S \times C$. Choosing G to be too small will result in smaller-than-ideal savings because the algorithm could otherwise build (sub-*)activation groups in subsequent filters. On the other hand, if G is too large, (sub-*)activation groups become too small and the cost of transferring results between dot products increases. Suppose, for simplicity, that weight values are uniformly distributed. Then the expected size of the innermost activation group size is $(R * S * C)/U^G$. To maximize efficiency, AGR uses the largest G such that $(R*S*C)/U^G \geq 1$, which indicates that the innermost activation group is non-empty on average. (If the innermost activation group is empty, AGR does not save computation and just adds metadata-related overhead.) If weights follow another distribution (e.g., Laplace), G must be set accordingly, so that innermost activation group size is still ≥ 1 on average.

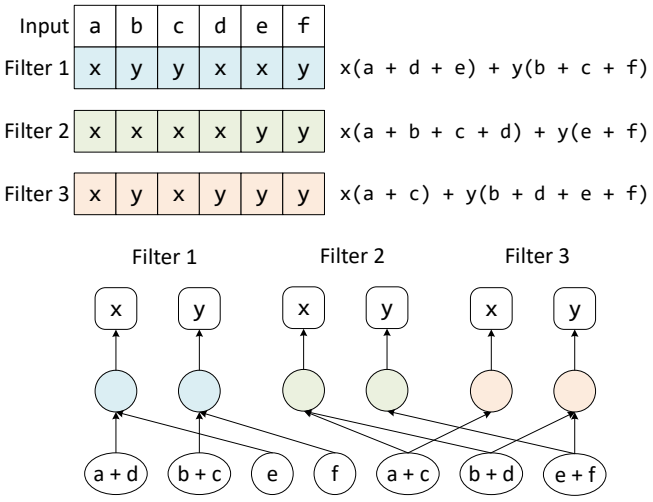

Figure 3: SumMerge graph generation example (using the same filters as were used in Figure 2). SumMerge evaluates the data-flow graph making up the factorized expressions for each of the three filters (shown in top half), while ensuring that shared sub-computations, e.g., $a+c$, $b+d$ and $e+f$ are only computed once. Square boxes with curved edges represent multiplies to the value inside the box. Circles represent graph vertices, which involve a sum accumulation over the values on the incoming edges. Observe, to evaluate 3 dot products for $K = 3$ filters, SumMerge requires the minimum $U * K$ multiplies, as desired.

3 ALGORITHM

3.1 Main Ideas and Overview

We now propose SumMerge, a new method to exploit weight repetition that significantly outperforms AGR. We start by revisiting Figure 2, to illustrate a key limitation in the AGR algorithm. While AGR ensures that sub-activation groups (a, d) and (b, c) are computed only once, it cannot reuse computation related to (e, f) which is a sub-activation group common to both Filter 2 and Filter 3. This is because the recursive-factorization strategy adopted by AGR does not allow sub-activation groups in Filter i to be built *across* activation groups in Filter $i-1$. In other words, when Filter 1 builds activation groups (a, d, e) and (b, c, f), this prevents Filter 2 and Filter 3 from exploiting repetition in (e, f) because e and f are members of different activation groups in Filter 1.

This issue is fundamental to how AGR recursively builds filters, and the missed opportunity increases as a function of the G parameter (Section 2.2.2). Section 3.6 shows this quantitatively by showing arithmetic reduction for AGR compared to SumMerge.

Another consequence of AGR's design is that while the number of multiplies per dot product is equal to U when $G = 1$ (e.g., Figure 1), it grows as a function of $O(U^G)$ in general. For example, in Figure 2 there are 12 multiplies needed to evaluate three dot products: 2 for Filter 1, 4 for Filter 2, 6 for Filter 3. The lower bound in multiplies is significantly lower—$U * G$ or $U * K$ for all filters—corresponding to one multiply for each unique weight in each filter.

Optimization metrics. Throughout this section, our goal will be to perform a convolution in as few arithmetic operations (adds and multiplies) as possible. Note, adds and multiplies have the same performance cost on our implementation platform (Section 4.1). Reducing either also reduces the number of memory reads.

3.2 SumMerge Algorithm

The main idea behind SumMerge is to reformulate the dot products making up the convolution as a traversal over a single data-flow graph, where vertices in the graph represent shared computations within/across the dot products and edges represent data transfers. Figure 3 shows how this idea addresses the missed opportunity from the example in Figure 2. Here, we sum fragments of activation groups individually, and memoize/apply those results to build activation groups for each filter. This allows us to now take advantage of all three groups of shared terms (a, d), (b, c) and (e, f) as well as others. Although the example only shows a two-level graph, in general, a graph will contain more levels as the algorithm discovers additional sub-computations and sub-sub-computations that can be shared across different subsets of filters.

The algorithm consists of two phases: First (Section 3.3), an offline step takes a trained CNN layer and generates data-flow graphs capturing memoization opportunities across groups of filters as described above. Second (Section 3.4), an online step takes a vector of input activations and the graphs from the previous step, and performs the inference operation.

Unlike AGR, whose offline step is trivial computationally (Section 2.2), the offline step for SumMerge requires a non-trivial search

Figure 4: SumMerge offline phase (Section 3.3): generating the data-flow graph. A Vertex **is a set of integers (indices), an** Edge **is a 2-tuple of** Vertex**. Dimensionality for inputs (**Layer **and** Weights**) are shown in brackets.** Filter ID **and** Weight ID **are integers in the range** $[0, K)$ **and** $[0, U)$**, respectively.** { **and** } **represent types of data structures. Loop bound notation assumes open interval upper bounds; i.e.,** $i : 0 \rightarrow K$ **means** $i \in [0, K)$**.**

```
   Input: Layer[K][R * S * C], Weights[U]
   Output: Vertices, EdgeLists, OutputMap
1  List{Vertex} Vertices;
2  List{List{Vertex ID}} EdgeLists;
3  List{(Vertex ID, Filter ID, Weight ID)}OutputMap;
4  // Add sink vertices to the graph
5  for i: 0 → K do
6    for j: 0 → U do
7      vertex = GroupByWeight(Layer[i], Weights[j]);
8      vertexID = Length(Vertices);
9      Vertices.Append(vertex);
10     EdgeLists.Append([ ]);
11     OutputMap.Append((vertexID, i, j));
12   end
13 end
14 // Remove intersections from vertices
15 while True do
16   List{Set{Int}} Intersections;
17   l = Length(Vertices);
18   for i: 0 → l, j: 0 → l do
19     if i ≠ j then
20       Intersections.Append(Vertices[i] ∩ Vertices[j]);
21     end
22   end
23   maxScore, bestIntersection =  MaxScore(Vertices, Intersections);
24   if maxScore == 0 then
25     break;
26   end
27   // Add the new vertex to the graph
28   newVertexID = Length(Vertices);
29   Vertices.Append(bestIntersection);
30   EdgeLists.Append([ ]);
31   for i : 0 → Length(Vertices) do
32     if bestIntersection ⊂ Vertices[i] then
33       Vertices[i] = Vertices[i] - bestIntersection;
34       EdgeLists[i].Append(newVertexID);
35     end
36   end
37 end
38 // Store a topological ordering of the vertices
39 Vertices, EdgeLists, OutputMap =
40   TopologicalSort(Vertices, EdgeLists,OutputMap);
41 return Vertices, EdgeLists, OutputMap;
```

to determine which sub-computations to share across filters. However, this step only needs to be computed once, offline, per trained network and is dwarfed by the time to train the network.

3.3 Step 1 (Offline): Graph Creation

The offline step takes as input a trained CNN layer and generates a graph data structure (edges, vertices) that represents the data-flow for the online step (Section 3.4). By evaluating the data-flow graph once, we will evaluate K dot products, i.e., all filters for a given spatial position in the input. In what follows, we represent

Figure 5: The MaxScore routine, used to select which vertex will be split next. The Index sub-routine returns the index of the first argument within the second. Intersections **and** Vertices **are defined in Figure 4.**

```
   Input: Vertices, Intersections
   Output: maxScore, bestIntersection
1  List{Int} Scores;
2  for i : 0 → Length(Intersections) do
3    score = 0;
4    for j : 0 → Length(Vertices) do
5      if Intersections[i] ⊆ Vertices[j] then
6        score += Length(Intersections[i]);
7      end
8    end
9    Scores.Append(score);
10 end
11 maxScore = Max(Scores);
12 bestIntersection =
13   Intersections[Index(maxScore, Scores)];
14 return maxScore, bestIntersection;
```

network layer Layer as a 2-dimensional array of weight values, of dimensions $K \times (R * S * C)$. As in AGR, we assume there are at most U unique weights in Layer.

Graph building is split into two sub-steps. First, we create an initial data-flow graph where no sub-computations are shared across dot products. Second, we progressively split these vertices into *smaller vertices with larger out-degree*. The idea is that each additional outwards edge represents a memoization to a later computation. We now describe each sub-step in detail.

3.3.1 Building Initial Vertices. We first build initial vertices, which will be the starting point for Section 3.3.2. Given Layer, we first group all locations (*indices*, representing input activations) in each filter that share the same weight, i.e., based on the activation groups for each filter (Section 2.2.1). Since there are no more than U unique weights in each filter, and K filters, this results in $U * K$ groups, some of which may be empty.

Each of the above groups is a sink vertex in the graph. For example, Figure 3 shows 6 sink vertices (the colored circles), where $K = 3$ and $U * K = 6$. Each vertex represents a sum over the inwards edges. For example, if a vertex v^* has inwards edges (v_i, v^*), then $v^* = \sum_i v_i$, where each vertex v_i is analogously defined. Thus, in the figure, the final values for the sink vertices of Filter 1 are $a+d+e$ and $b + c + f$.

We next populate a data structure called the OutputMap, which associates each sink vertex with a filter ID from 0 to $K - 1$ and a weight ID from 0 to $U - 1$. In the online step, this tells us to which filter each sink vertex corresponds, and by which weight to multiply the sink vertex.

Together, this sub-step is shown in Lines 5-13 of Figure 4.

3.3.2 Splitting Vertices. Starting with the above initial data-flow graph, we now perform transformations on the graph to reduce the arithmetic required to 'traverse' the graph while preserving

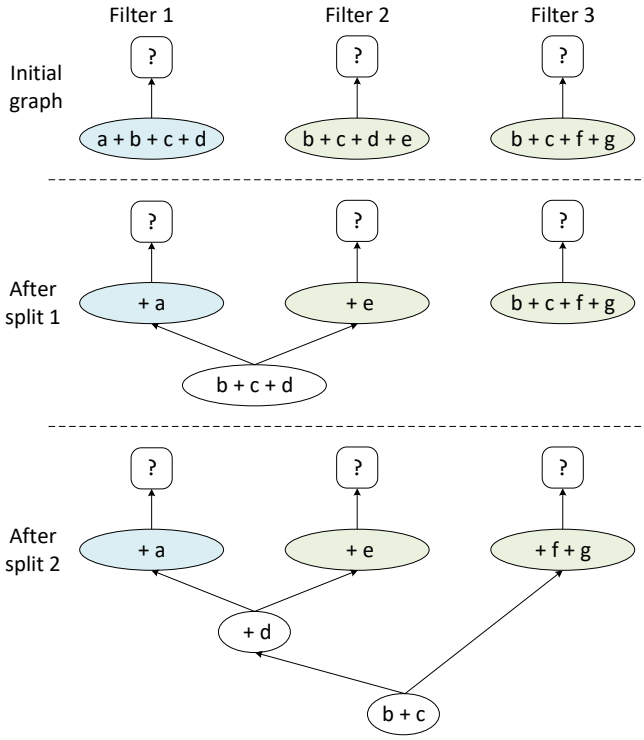

Figure 6: Splitting graph vertices for three filters, showing one activation group per filter for simplicity. The weight multiplied to each activation group is arbitrary, hence marked '?'. "Initial graph" represents the output of Section 3.3.1. The first split reuses the computation b+c+d because that computation is the largest overlapping sub-computation in the first two filters (Section 3.3.2). The notation +a indicates this vertex is the result of summing the values on its inwards edges plus a. Thus, the number of addition operations to evaluate each graph is proportional to the number of + symbols in each graph and we want to minimize this value. The second split further divides the sub-computation from the first split to reuse computation for the third filter. Note: Filter 1-3 shown here are different than those used in previous figures.

semantic equivalence.[1] Our approach is to iteratively *split the vertices* formed in the previous sub-step, to discover opportunities to memoize computations.

Figure 6 shows an example vertex split. The indices (corresponding to input activations) comprising a vertex form a set, and we now perform set intersections across pairs of vertices (sets). The size of the intersection indicates overlap between vertices, and this overlap represents a common sub-computation (sequence of additions) that can be computed once and reused later. We represent this common sub-computation by removing the indices in the intersection from both vertices and building a new vertex containing the common indices, with outward edges pointing to each of the input vertices, as shown in the figure "After split 1."

The challenge is, given the $U * K$ initial vertices, there are $(U * K)^2 - U * K$ pairs of vertices from which we must choose to intersect. Further, our choice of which intersections to turn into vertex splits may impact the quality of results (since multiple intersections will involve the same input activations). In general, deciding which vertex splits to perform, and in what order, is a complex combinatorial search problem. For this paper, we propose a greedy heuristic called MaxScore which finds good solutions reasonably quickly. While asymptotically this procedure has complexity $O((R*S*C*K)^3)$ subset-comparison ($\subseteq$) operations for the entire graph-creation process (all of Figure 4), it takes only seconds to minutes for each DNN layer we evaluate in Section 5.

The heuristic first generates all $(U * K)^2 - U * K$ intersections, i.e., computes $v_i \cap v_j$ for all vertices v_i, v_j where $i \neq j$ (Figure 4, Lines 18-22). Next, we rank these intersections (Figure 5) and pick the highest-ranking intersection (a set denoted bestIntersection). The core operation in MaxScore is Figure 5, Line 6, which encodes two ideas:

- An intersection's score is proportional to the number of vertices that intersection is a subset of (which indicates the degree of reuse).
- An intersection's score is proportional to the size (Length) of the intersection (which indicates a greater savings per reuse).

If the size of bestIntersection is 0, this implies there are no overlaps left to exploit, and that graph generation is complete. Otherwise, we add the bestIntersection as a new vertex to the graph, remove the indices in bestIntersection from every vertex v where bestIntersection $\subset$ v, and add a directed edge that points from bestIntersection to each such v (Figure 4, Lines 30-36). We repeat the above steps (i.e., we re-generate all possible intersections between all pairs of vertices, now also counting the new vertex added by the split, and subsequent steps) until the algorithm terminates with a bestIntersection of 0. Alternatively, an implementation could terminate early when all vertices reach a specified minimum size. Finally, we note that while Figure 4, Line 20 is shown to recompute all intersections during each iteration (for simplicity), our implementation only recomputes intersections for vertices that were split in the previous iteration (to save compute).

To *evaluate* the graph (Section 3.4), we need to perform computation (additions) at each vertex and data transfers for each edge. To make this process more efficient during the online step, we perform a topological sort on Vertices (Figure 4, Line 40). Note, sorting Vertices requires that $vertexID$ and other fields be updated in other data structures. Hence, evaluation involves a single scan of the vertex list representing the graph, from sources to sinks, where each step requires gather-, arithmetic- and scatter-style operations.

Partitioning the filters into groups of G. Generating a single data-flow graph for all K filters typically results in less-than-optimal arithmetic reduction because the graph has a relatively high number of edges (data transfers). Thus, for each layer, we break the K filters into groups of G, similar to AGR. This is conceptually simple: in the above explanations, we replace K with G and only allow intersections to be done within the group of G filters. Then, to

[1]Note, we assume multiplication and addition are associative and commutative. While these properties do not hold for our current evaluation due to our use of floating point (Section 4.1), they typically do hold for inference due to pervasive use and upcoming hardware support for fixed point [6].

Figure 7: SumMerge online phase (Section 3.4): Performing inference. NumType refers to the data type for weights and activations. Vertices**,** EdgeLists **and** OutputMap **are output during the offline step (Figure 4). Elements of** OutputMap **are 3-tuples as defined in Figure 4. (Not shown) We assume** Output **is 0-initialized at the start of the operation.**

```
   Input: Inputs[W * H * C], Weights[U], Vertices, EdgeLists, OutputMap
   Output: Output[(W − R + 1) * (H − S + 1) * K]
1  InputTile = List{NumType};
2  Graph = List{NumType};
3  // Iterating over every spatial position
4  for i : 0 → W, j : 0 → H do
5    // Phase 1: Copy the InputTile
6    index = 0;
7    for c : 0 → C, h : j → (j + S), w : i → (i + R) do
8      InputTile[index] = Input[W * H * c + W * h + w];
9      index += 1;
10   end
11   // Phase 2: Accumulate incoming edges
12   for v : 0 → Length(Vertices) do
13     accumulator = 0;
14     for x : 0 → Length(EdgeLists[v]) do
15       accumulator += Graph[EdgeLists[v][x]];
16     end
17     for x in Vertices[v] do
18       accumulator += InputTile[x];
19     end
20     Graph.Append(accumulator);
21   end
22   offset = ((W − R + 1)(H − S + 1) * h + w) * K;
23   // Phase 3: Multiply sink vertices with weights
24   for x : 0 → Length(OutputMap) do
25     mapping = OutputMap[x]
26     index = offset + mapping[0];
27     sum = Graph[mapping[1]];
28     weight = Weights[mapping[2]];
29     Output[index] += sum * weight;
30   end
31 end
32 return Output;
```

evaluate all K dot products, we evaluate $\lceil K/G \rceil$ data-flow graphs—one for each group of G filters.

The best value of G is hard to predict, so we test different values, and choose the best one. The trade-off is: 1) Higher G increases irregularity by increasing the number of vertex splits, i.e., increases the time to evaluate each graph. 2) Higher G implies that more computation is shared across dot products, i.e., reduces the number of graphs that need to be evaluated to perform inference.

3.4 Step 2 (Online): Inference

The online step takes a vector of input activations and the data-flow graphs constructed in the previous section (Section 3.3) and performs the inference operation. Code for the entire operation is shown in Figure 7.

For each spatial position i and j, the computation has three phases. First (Line 5), we copy the input activations being computed on at this spatial position to a data structure called InputTile. Second (Line 11), we traverse the data-flow graph and performs all add operations. Specifically, for each vertex in the graph, we:

- Accumulate values on all incoming edges into that vertex (Line 15).
- Accumulate input activations corresponding to the indices in the vertex itself (Line 18).

Edge values are stored in a temporary structure called Graph and input activations are read directly from InputTile. Third (Line 23), we iterate through the OutputMap to perform multiplications on the partial sums formed during phase 2.

Hardware-level performance considerations. We wrote the kernel in the above fashion to make it easier to performance optimize. There are two points to emphasize now. First, we copy input activations into InputTile to improve data locality in the inner loop. Note that while building InputTile is done online and does take time, it is a relatively simple memory copy operation (that our target platform will be able to accelerate with vector moves) and is amortized over the K dot products computed at that spatial position. Second, while the innermost loops (Lines 15 and 18) require expensive indirections, the indirection pattern depends only on the data-flow graph and *not* the spatial position (i, j). Further, since indirections are *relative* to each spatial position, corresponding input activation indices—i.e., for the same indirection but at different spatial positions—are related by a simple affine function. Putting these together, we will be able to vectorize the inner loop without the use of expensive gather instructions. More details are given in Section 4.

3.5 Other Considerations

Sparsity. Although sparsity can be viewed as a special case of weight repetition, the savings are different because of the absorption property $x * 0 = 0$. We exploit sparsity by forcing GroupByWeight (Figure 4, Line 7) to return an empty vertex whenever the 2nd argument (Weights$[j]$) is 0. This means the vertex will have no incoming edges/partial sum accumulation and that the kernel will skip all computation needed by only the 0-valued weight.

Per-layer vs. per-filter weights. Depending on the quantization scheme, the same U weights may be shared across all filters (e.g., TTQ [42]) or each filter may have a distinct U weights (e.g., LQ [39] and LS [32]). Up to this point, the description has been consistent with the former case. For the latter case, OutputMap is modified to store weights, not weight IDs, and Weights (input to Figure 4) becomes $U \times K$-dimensional. Correspondingly, Line 28 of Figure 7 indirects into the mapping, not Weights.

3.6 Improvement over AGR

Figures 9-10 compare the arithmetic reduction possible for SumMerge and AGR (Section 2.2), for different synthetic weight distributions, different numbers of unique weights (U) and filter sizes ($R * S * C$). We study both uniformly- and Laplacian-distributed weights as we find these to represent opposite extremes in real DNN layers. For uniform distributions, one of the U weights is the 0 weight. We show the Laplacian used to generate the data in Figure 8. To sample Laplacian-distributed weights, we create U bins of uniform width from -1 to 1 (Figure 8) and assign each bin a weight value given by the bin midpoint. We evaluate U that are odd; hence, one weight is always the 0-valued weight and 0 is the most common weight.

We define the number of arithmetic operations as the sum of two-input additions and multiplications. As expected, both schemes reduce arithmetic relative to a conventional dot product, and benefit from fewer unique weights (smaller U) and larger filters (larger $R * S * C$). There are two additional takeaways. First, given both uniformly- and Laplacian-distributed weights, SumMerge improves arithmetic reduction relative to AGR, for a given U and $R * S * C$. Second, while AGR sees roughly the same improvement for both weight distributions, SumMerge sees a significant improvement given a Laplacian distribution.

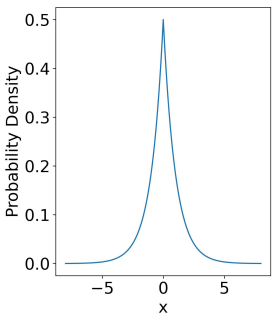

Figure 8: Probability density function of a Laplace distribution with mean = 0.0 **and** scale = 1.0**.**

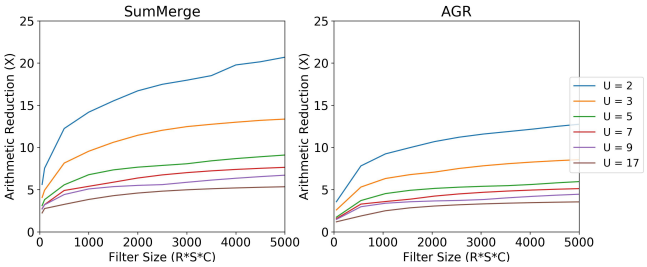

Figure 9: Arithmetic reduction (higher is better) for SumMerge and AGR given uniformly-distributed weights for different numbers of unique weights (U) and filter sizes ($R*S*C$). Results are relative to a dense computation.

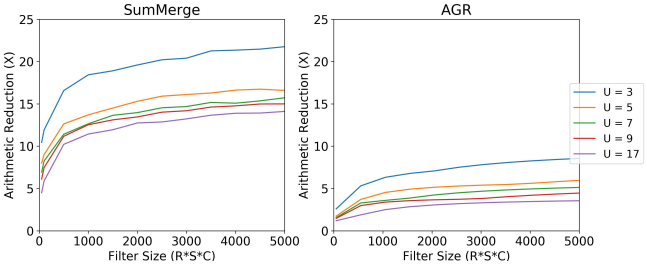

Figure 10: Same as Figure 9 but given Laplacian-distributed weights (with mean = 0 **and** scale = 1**). We do not show data for $U = 2$ because splitting the Laplacian into two bins results in a uniform distribution.**

To understand why, note from Figure 8 that weight distribution is skewed given a Laplacian. Call the most frequent weight the *common weight*. Due to its high frequency, activation groups related to the common weight (for both AGR and SumMerge) contribute a relatively large percentage of the total arithmetic. Yet, with AGR the activation group formed for the common weight becomes (sub-*)activation groups of rapidly shrinking size as a function of G because in AGR, (sub-*)activation groups must be subsets of their parent activation groups (Section 2.2.2). SumMerge does not have this restriction (Section 3.1), enabling it to save more computation.

4 KEY OPTIMIZATION TECHNIQUES

We developed a C++ implementation of SumMerge and compared performance with oneDNN, Intel's state-of-the-art library for deep learning. While SumMerge is efficient with regard to the number of arithmetic operations, it also introduces irregularity in the form of branches and more frequent, irregular memory accesses when compared to oneDNN which instead performs a dense version of convolution. To mitigate these overheads and improve performance we employ several optimizations discussed below.

4.1 Evaluation Platform

To provide context for some of our design decisions, we briefly describe our evaluation platform's hardware while noting that our techniques can be easily extended to other platforms with SIMD support. We consider a single node system comprising a general-purpose processor with 6 Intel Skylake cores and support for SIMD. In each cycle, each core can execute two AVX-512 arithmetic instructions (e.g., vector fused multiply-add, or VFMA), read two cache lines (64B) and write one cache line from/to the L1 data cache, and retire four instructions. In addition, FP vector add and vector multiply have the same latency and throughput; FP scalar add and multiply have the same throughput, but FP scalar multiply has two cycles longer latency [16]. Further, each core has 32 vector registers, a 32KB L1 data cache, a 1MB L2 cache, and a 1.375MB slice of a shared, non-inclusive L3 cache.

4.2 Exposing Parallelism and Vectorization

Each iteration of the spatial traversal (Figure 7, Line 4) is independent in both the W and H dimensions, i.e., there are no data dependencies between different iterations. Further, there is no control-flow divergence between iterations, i.e., we use the same Vertices, EdgeLists, and OutputMap in each iteration and all the control logic in phases 2 and 3 of the online step (Line 11 and Line 23 of Figure 7) depend on values from these data structures.

The above implies that we can parallelize and vectorize the spatial traversal. Specifically, we use vector instructions throughout the inner loop and process $VectorWidth$ iterations of the W loop in a single iteration. Further, since each core can execute two vector instructions in a single cycle, and since our hardware's vector arithmetic latency is multiple cycles, we unroll the H loop by a factor $VerticalUnrolling$. This ensures we have enough instruction level parallelism (ILP) in the inner loop to fully utilize the vector functional units in a core.

4.3 Tiling

If the implementation uses vector instructions and loop unrolling as detailed above, the size of InputTile increases from $R*S*C$ to $R*(S+VerticalUnrolling-1)*C*VectorWidth$. Similarly, if the number of vertices in the original graph is V, the memory footprint of Graph increases from V entries to $V*VectorWidth*VerticalUnrolling$ entries. This could mean that both Graph and Inputs no longer fit even in the L3 cache, resulting in long-latency memory accesses.

To mitigate this effect while still utilizing unrolling and vector instructions, we reduce the working set size from $R*S*C$ indices to $R*S*C_t$ indices where C_t describes the tile size in the C dimension. A consequence of this scheme is that we now only take advantage of repetitions in a volume of $R*S*C_t$ as opposed to $R*S*C$, leading to reduced arithmetic reduction. Also, inference evaluates $(K/G)*(C/C_t)$ graphs instead of the (K/G) graphs per spatial position.

4.4 Multi-threading

While we have described a single-threaded algorithm meant to run on one physical processor core, there are multiple ways to multi-thread the computation to take advantage of multiple cores. We opt for a simple multi-threading strategy whereby inference is performed on multiple independent inputs, and each physical core computes on one input. This enables different threads to share model parameters (e.g., Vertices, EdgeLists, etc.). As each thread is designed to have sufficient ILP to saturate VFMA bandwidth per core (Section 4.2), we only run one thread per physical core, i.e., disable hyperthreading.

Alternatively, in a scenario involving single-input inference, there are multiple ways to multi-thread work within a single input due to the abundant coarse-grain parallelism present in DNN computations. For example, we can parallelize across spatial positions (W, H) or graphs (of which there are $\lceil K/G \rceil$ where $K \gg G$ is typical). Each strategy enables different data sharing. For example, parallelizing across graphs increases the working set size related to the Graph data-structure but shares InputTile across threads. Parallelizing across spatial positions has the opposite characteristics.

5 EVALUATING PERFORMANCE

We now evaluate SumMerge on multiple real quantized DNNs and synthetic configurations.

5.1 Methodology

Test environment. All runs for all configurations are performed on an Intel Core i7-7800X CPU (matching the description in Section 4.1). This machine has 6 physical cores, 32/32 KB of L1 instruction/data cache per core, 1 MB of L2 cache per core, and 8.25 MB of shared non-inclusive L3 cache. We disable simultaneous multithreading as well as dynamic frequency scaling and enable 2MB pages.

Test methodology. All speedup numbers correspond to the online step (Section 3.4). The offline step time (Section 3.3) is not shown but takes a few seconds to several minutes, depending on R, S, C, C_t, K and G. Overall speedup numbers include the time to compute all convolutional layers that each quantization scheme elects to quantize (others are run dense). The fully-connected layer is always run as dense; further, some schemes do not quantize the first convolutional layer [42]. All weights and inputs are represented as 32-bit floating point numbers (FP32) and all arithmetic operations are computed in terms of FP32 values. Finally, each experiment is run 50 times when the machine is unloaded; reported values are from the run with lowest execution time.

Test configurations. We evaluate SumMerge relative to oneDNN [5], a state-of-the-art CPU library for DNNs on Intel processors, and a single-threaded software implementation of the AGR algorithm (Section 2.2). We constructed the AGR kernel by hand and carefully applied a number of best-practices optimizations (e.g., vectorization/data parallelism and tiling similar to that described in Section 4, factoring work out of inner loops) to make it as high performance as possible.

Quantization schemes. To show SumMerge robustness, we evaluate performance on six quantization schemes (with different U values), across several DNN architectures and datasets [8, 12, 32, 33, 39, 42]—shown in Table 1. For each entry in the table, we used code from each work's public repository to train and quantize each DNN. (We only evaluated DNN-dataset combinations that were publicly available/evaluated.) The goal in populating the table was to have configurations covering several values of U, DNN architectures and datasets. Finally, we only show $U = 2, 3$ configurations as our current implementation does not see overall speedup for $U \geq 4$ on the quantization schemes studied in the table.

Quantization scheme details. For each quantization scheme, we quantize weights but not activations. While each scheme quantizes to a different set of weights, the primary concern for SumMerge is the effective U value, the resulting weight distribution (Section 3.6) and whether one of the weights is 0 (Section 3.5). For example, TTQ [42] and LS-T [32] quantize to $U = 3$ weights: $\{Wn, Wp, 0\}$ and $\{-S_k, S_k, 0\}$, respectively, where Wn and Wp are positive and negative per-layer weights and S_k is a per-filter weight (Section 3.5). The difference in values between W_n, W_p and S_k does not impact performance, but the fact that each scheme includes the 0-valued weight, and the weight distribution of each layer, does impact performance.

5.2 Main Result

The main result—Overall Speedup vs. oneDNN vs. AGR and DNN model storage for each quantization scheme configuration—is shown in Table 1.

5.2.1 Performance results. As we can see, there is a large speedup range (from 1.09× to 2.05×, single-threaded), relative to oneDNN, depending on parameters. We also see significant speedup (from 1.04×-1.51×) relative to AGR, consistent with our findings in Section 3.6. We now breakdown sources of speedup along several dimensions (referencing single-threaded speedup numbers for consistency).

Number of unique weights (U): Keeping all else constant, decreasing U enables larger speedup, and changing U in general enables speedup/accuracy trade-offs. Consider the LS-1 and LS-T configurations, which differ only in U. The former ($U = 2$) loses 2% accuracy relative to a full precision DNN and achieves 2.05× speedup; the latter ($U = 3$) loses 1.3% accuracy and sees 1.52× speedup. Which configuration is best depends on the deployment scenario.

Smaller U enables higher speedup because it implies larger repeated weight overlaps across filters, and consequently higher "best" G values (Section 3.3.2). For example, in the LS-1 and LS-T configurations, we found that G ranges from 6 to 8 (depending on the layer) for LS-1 and ranges from 4 to 6 for LS-T.

Table 1: Configurations evaluated and main results. Top-1 "Full" gives validation set accuracy reported from the literature; "Quantized" accuracies were measured from our trained models after we performed quantization. (Note that while our quantized accuracy shows a several % drop relative to full precision for ImageNet, this is because we did not have the compute to perform sufficient fine tuning. With fine tuning, prior work has reported significantly higher quantized accuracy [24, 42], e.g., to within 1% of the full-precision model [24].) BWN configurations were obtained from the DoReFa-Net repository [41]. VGG*, AlexNet* and AlexNet are derivative architectures based on VGG [34] and AlexNet [26], respectively. DNN architectures marked with ‡ have smaller values of C and K than the original architecture to prevent over fitting on smaller datasets [3, 9]. ST stands for single-threaded, MT stands for multi-threaded. G range denotes the range of G values (Section 3.3.2) SumMerge selects across all layers.**

				Top-1 accuracy (%)		Overall speedup (X) vs. oneDNN		Overall speedup (X) vs. AGR	Storage reduction (X) relative to Dense FP32		G
Quantization	U	**DNN arch**	**Dataset**	**Full**	**Quantized**	**ST**	**MT**	**ST**	**SumMerge**	**Codebook**	range
BWN [33]	2	AlexNet* [3]	ImageNet [15]	59.7	55.7	1.76	1.47	1.19	14.9	31.9	5-8
BWN	2	AlexNet**‡ [4]	SVHN [1]	97.7	97.3	1.61	1.33	1.32	13.9	31.9	4-8
BC [12]	2	ResNet20‡ [21]	CIFAR10 [25]	91.9	90.2	1.36	1.39	1.08	11.6	31.8	4-7
ProxQuant [8]	2	ResNet20‡	CIFAR10	91.9	90.3	1.30	1.30	1.04	10.0	31.8	4-6
ProxQuant	3	ResNet20‡	CIFAR10	91.9	91.3	1.09	1.02	1.17	8.7	15.9	3-4
LQ [39]	2	VGG*‡ [9]	CIFAR10	93.8	93.7	1.69	1.29	1.22	14.8	31.2	7-8
LQ	2	GoogleNet [35]	ImageNet	72.9	69.1	1.60	1.34	1.43	14.3	30.5	3-8
LS-1 [32]	2	ResNet18 [21]	CIFAR100 [25]	77.8	75.8	2.05	1.50	1.46	15.4	31.3	6-8
LS-T [32]	3	ResNet18	CIFAR100	77.8	76.5	1.52	1.21	1.43	10.5	15.7	4-6
TTQ [42]	3	AlexNet*	ImageNet	59.7	55.2	1.86	1.58	1.51	13.9	15.6	3-9

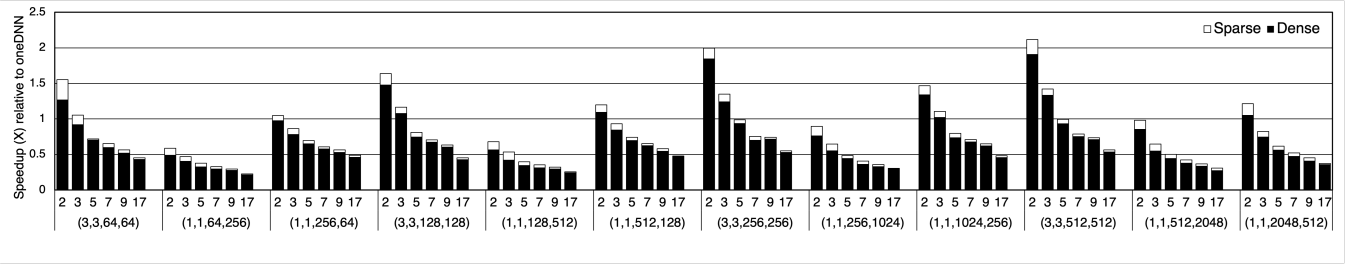

Figure 11: Per-layer performance analysis (uniformly-distributed weights). Each group/4-tuple indicates layer shape (R, S, C, K). Each number within each group (2,3,5,7,9,17) indicates a different U value used for that layer. Note that Sparse speedup is always greater than Dense speedup because SumMerge saves additional computation in the presence of the 0-valued weight.

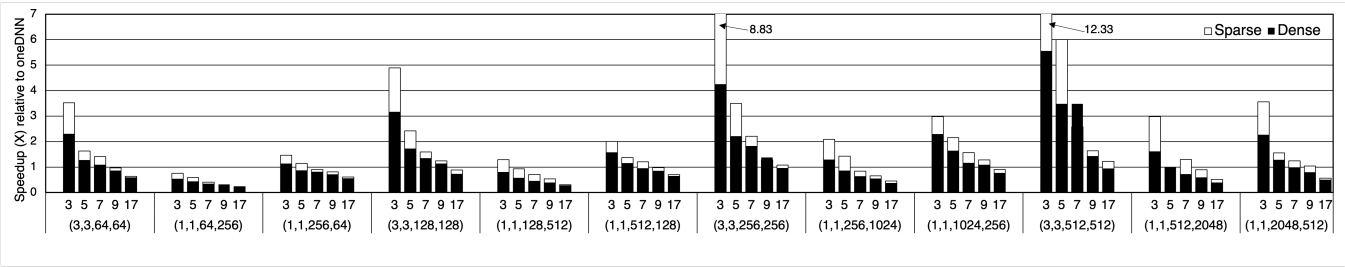

Figure 12: Per-layer performance analysis (Laplacian-distributed weights). Conventions are the same as in Figure 11. As in Figure 10, we do not show data for $U = 2$ because splitting the Laplacian into two bins results in a uniform distribution.

Average filter size: Generally, DNN architectures with larger filters (larger $R*S*C$) see larger speedup (Section 3.6). Consider the BC configuration on CIFAR10 relative to the LS-1 configuration on CIFAR100, which are both $U = 2$ but run on ResNet20 and ResNet18, respectively. ResNet20 has a similar architecture to ResNet18, but with smaller C values, i.e., by a factor of 4 to 8 depending on the layer. Hence, we should expect higher speedup from ResNet18, since larger C results in a larger window through which to search for repetitions (when $C > C_t$, which is not always the case with ResNet20).

Dataset (weight distribution): Finally, keeping U and DNN architecture constant, we still expect performance differences because

changing dataset (or even re-training on the same dataset) will result in a different distribution of weight values, which will impact data-flow graph structure and therefore speedup (Section 3.6).

Finally, we note that our single-threaded speedups are larger than their multi-threaded counterparts. We believe this is because single-threaded SumMerge better exploits the shared parts of the memory hierarchy than the single-threaded oneDNN.

5.2.2 Storage results. Although SumMerge requires auxiliary data structures to represent each DNN layer, e.g., Vertices, a key observation is that this metadata is shared across groups of G filters. This means Bytes-per-filter storage is effectively divided by a factor of G which enables a substantial storage compression, relative to a dense representation. We calculate model storage as the sum of the storage required for all data-flow graphs, where a graph with E edges and V vertices (representing G filters) requires $U * 32$ bits for Weights, $E * log2(V) + V * \log2(V)$ bits for EdgeLists, as well as $K * U * \log2(V)$ bits for OutputMap (stored in sorted order). The Vertices data structure can be inferred from EdgeLists if we store the edge lists in the order specified by the topological sort. Storage for the dense baseline is calculated as $R * S * C * K * 32$ bits per layer i.e. we assume the weights are stored as FP32 values.

Storage savings and G ranges are shown in Table 1. For comparison, we also show the compression factor given a standard codebook scheme—i.e., that represents each weight in $\log2(U)$ bits—which is commonly used by quantization schemes [42]. Importantly, codebook-based schemes do not enable faster inference (in fact, they would likely slow inference due to indirection logic). Thus, the takeaway is that SumMerge enables competitive compression relative to a codebook while simultaneously enabling significant speedup.

5.3 Per-Layer Analysis

To provide additional insight, we now perform a more comprehensive study using synthetically generated weights, varying U, layer parameters (R, S, C, K) and weight distributions. We analyze performance on a range of layer shapes that appear in modern DNNs, e.g., ResNets [21]. We show results for the two distributions used in Section 3.6: uniform and Laplacian with 0 mean and 1 scale. To distinguish the speedup due to weight repetition vs. the speedup due to sparsity (repetition of the 0-valued weight), we show "Dense" and "Sparse" data. The former means all U weights are non-zero. The latter means the 0-valued weight is present. All weight values are assigned using the same methodology as in Section 3.6. All runs assume an input of dimensions $W = H = 112$.

The main takeaways echo what we saw in the main result and Section 3.6. First, speedup increases as U decreases. Second, speedup increases as layer size $(R * S * C)$ increases. Note, speedup does not depend on K. Thus we expect larger speedup given a layer with parameters, e.g., (1, 1, 256, 64), relative to a layer with parameters (1, 1, 64, 256). Third, speedup given Laplacian-distributed weights is larger than with uniformly-distributed weights. Finally, importantly, both repetition of non-zero-valued weights and the 0-valued weight independently contribute significantly to overall speedup.

6 RELATED WORK

We describe related work on quantization and activation group reuse (AGR) in Section 2 as well as compatibility with SumMerge.

Deep Reuse [29] proposes an algorithm to speedup convolution by exploiting similarity between vectors of input activations. This work is complementary to SumMerge in that SumMerge exploits similarity in weights while Deep Reuse exploits similarity in activations. In more detail, Deep Reuse looks for vectors of similar activations both within and across activation maps, creating vectors of neurons that closely represent several vectors of consecutive activations in the input. Next, instead of multiplying entire activation maps with filters as is the case with dense convolution, Deep Reuse multiplies these select neuron vectors with filters and then reconstructs the output activation map to arrive at a similar result to what is attained by doing the dense version of the operation. Here, SumMerge can be used as an efficient alternative to doing the dense operation between a neuron vector and the filters, further reducing the number of arithmetic operations required.

Cowan et al. [14] discusses a complementary, automated approach for implementing quantized inference that relies on the scheduling phase of a compiler and program synthesis techniques. Specifically, [14] takes as input a quantized inference kernel which we believe can be replaced by a SumMerge-based kernel. The result would be an optimized version of SumMerge that takes into account bitplane scheduling and other complementary optimizations discussed in [14].

7 CONCLUSION

This paper proposed SumMerge, the first weight repetition-aware DNN inference kernel that speeds up execution on a general-purpose platform (as opposed to a custom accelerator). The key idea is that weight repetition implies a data-flow graph where computations can be simplified within a filter, and memoized across filters. To prove out the idea, we wrote an optimized C++ kernel that represents DNN inner products as partially shared data-flow graphs, evaluates computation in data-flow order, and speeds up multiple existing quantization schemes relative to an optimized CPU baseline.

Acknowledgements. We thank Nandeeka Nayak, Joel Emer, Michael Pellauer and Kartik Hegde for insightful discussions, as well as the anonymous reviewers for their very helpful feedback. This work was partially funded by DARPA SDH contract #HR0011-18-3-0007 and by NSF grants #1909999 and #1942888.

REFERENCES

[1] The Street View House Numbers (SVHN) Dataset. http://ufldl.stanford.edu/housenumbers/, 2011.
[2] GPU-Based Deep Learning Inference: A Performance and Power Analysis. In *NVIDIA Whitepaper*, 2015.
[3] AlexNet Derivative 1. https://github.com/tensorpack/tensorpack/blob/master/examples/DoReFa-Net/alexnet-dorefa.py, 2016.
[4] AlexNet Derivative 2. https://github.com/tensorpack/tensorpack/blob/master/examples/DoReFa-Net/svhn-digit-dorefa.py, 2016.
[5] Intel oneDNN. https://github.com/oneapi-src/oneDNN, 2016.
[6] Introduction to Intel Deep Learning Boost on Second Generation Intel Xeon Scalable Processors. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-deep-learning-boost-on-second-generation-intel-xeon-scalable.html, 2019.
[7] Amazon sagemaker ml instance types. https://aws.amazon.com/sagemaker/pricing/instance-types/, 2021.

[8] Y. Bai, Y. X. Wang, and E. Liberty. Proxquant: Quantized neural networks via proximal operators. arXiv (2018).
[9] Z. Cai, X. He, J. Sun, and N. Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *CVPR'17*.
[10] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR'12*.
[11] J. Cong and B. Xiao. Minimizing computation in convolutional neural networks. In *ICANN'14*.
[12] M. Courbariaux, Y. Bengio, and J. P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS'15*.
[13] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. In *NIPS'16*.
[14] M. Cowan, T. Moreau, T. Chen, J. Bornholt, and L. Ceze. Automatic generation of high-performance quantized machine learning kernels. In *CGO'20*.
[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR'09*.
[16] Agner Fog et al. Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.
[17] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: efficient inference engine on compressed deep neural network. In *ISCA'16*.
[18] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR'16*.
[19] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *NIPS'15*.
[20] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA'18*.
[21] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR'16*.
[22] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *ISCA'18*.
[23] G. Hinton, L. Deng, D. Yu, G. Dahl, A. R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, and T. Sainath. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29:82–97, November 2012.
[24] Q. Hu, P. Wang, and J. Cheng. From hashing to cnns: Training binaryweight networks via hashing. In *AAAI'18*.
[25] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.
[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS'12*.
[27] H. Malmgren, M. Borga, and L. Niklasson. *Artificial Neural Networks in Medicine and Biology: Proceedings of the ANNIMAB-1 Conference, Göteborg, Sweden, 13–16 May 2000*. Springer Science & Business Media, 2012.
[28] J. Morajda. Neural networks and their economic applications. In *Artificial intelligence and security in computing systems*, pages 53–62. Springer, 2003.
[29] L. Ning and X. Shen. Deep reuse: Streamline cnn inference on the fly via coarse-grained computation reuse. In *ICS'19*.
[30] J. Park, S. R. Li, W. Wen, H. Li, Y. Chen, and P. Dubey. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. arXiv (2016).
[31] J. L. Patel and R. K. Goyal. Applications of artificial neural networks in medical science. *Current clinical pharmacology*, 2(3):217–226, 2007.
[32] H. Pouransari, Z. Tu, and O. Tuzel. Least squares binary quantization of neural networks. In *CVPRW'20*.
[33] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV'16*.
[34] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR'15*.
[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR'15*.
[36] C. J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. M. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. In *HPCA'19*.
[37] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, and K. Keutzer. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas. In *FPGA'19*.
[38] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *ISCA'17*.
[39] D. Zhang, J. Yang, D. Ye, and G. Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *ECCV'18*.
[40] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. In *ICLR'17*.
[41] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv (2016).
[42] C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. In *ICLR'17*.