2025 IEEE International Symposium on Circuits and Systems (ISCAS)

Alternating Greedy Schedules: Enabling Low-Bitwidth Accumulation of Dot Products in Neural Network Computations

Vikas Natesh
Harvard University

H. T. Kung
Harvard University

***Abstract*—We present Alternating Greedy Scheduling (AGS), an algorithm for avoiding overflow, specifically *transient* overflow, during low-bitwidth accumulation of dot products in neural network computations. In conventional quantized (e.g., 8-bit) dot products, partial results are accumulated into wide (e.g., 32-bit) accumulators to avoid overflows when accumulating intermediate partial sums. However, such wide accumulators increase memory bandwidth usage and reduce energy efficiency. We show that iterative N:M pruning in floating point followed by quantization to 8 (or fewer) bits, and accumulation of partial products in an optimal order (via AGS) allows for accurate, compressed models with a large number of partial products that do not require wide accumulators. We design, analyze, and implement the AGS algorithm to eliminate accumulation overflows at inference time for several neural networks. Our method offers a 2.7x reduction in accumulator bitwidth while achieving model accuracy on par with floating-point baselines for multiple image classification tasks.**

I. Introduction

To utilize AI for the edge, low-power Internet of Things (IoT) devices and tinyML applications have been employed to perform a variety of tasks, including eye tracking, gesture detection, motion detection, speech recognition, and head movement detection [1]–[4]. There is a growing demand for efficient implementations of AI with heavily limited memory, bandwidth, and computation power. To this end, model compression is essential for efficient inference on low-power devices. Pruning and quantization are two common approaches to compressing neural networks. Low-power devices for tinyML typically have small local memories [5] and often lack support for efficient floating-point computation [2], [6]. Hence quantization is, by default, a necessity on such systems and most tinyML models are quantized to 8 bits or less.

When performing quantized matrix multiplications, dot products are typically accumulated into 32-bit registers. Reducing accumulator bitwidth can reduce bandwidth and energy usage while increasing inference throughput [7]–[9]. However, if the partial product sum overflows the accumulator, its value may be clipped to a finite range. This introduces numerical errors into the final matrix result that degrade model accuracy and limit how much we can reduce the accumulator bitwidth.

Prior works have attempted to reduce overflow in narrow accumulators through regularization on the loss function [8] or by controlling weight magnitude during training [7], [9]. While such approaches succeed in reducing overflows, they impose restrictive constraints on weights that may reduce model accuracy [7], [8], [10]. Weight magnitude constraints also promote unstructured sparsity in the network [7], which is beneficial for reducing model size but difficult to accelerate.

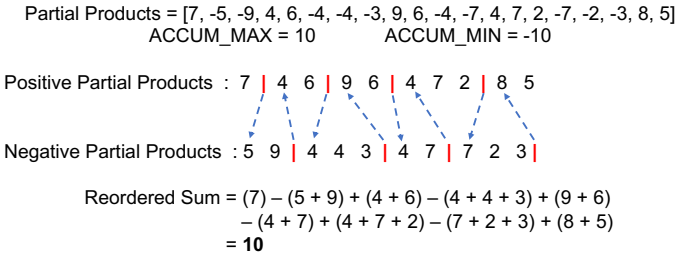

Fig. 1. Example of Alternating Greedy Scheduling (AGS) when summing 20 partial products into an accumulator that holds values in the range [-10, 10]. The red lines mark when the running sum is about to overflow the accumulator. The blue arrows depict how we alternate between adding from positive and negative partial products to avoid overflow. Following the blue arrows through the positive and negative lists reveals the reordered summation where the running sum is always between -10 and +10 (no overflow). The scheduling is *greedy* in the sense that it accumulates as many values of the same sign as possible before switching to values of the opposite sign.

We combine pruning and quantization with a novel overflow avoidance algorithm to enable low-precision (i.e., low-bitwidth) accumulation in quantized neural networks (QNNs) with minimal accuracy degradation. We characterize overflows as **persistent** or **transient**, depending on whether the final accumulation result overflows or only an intermediate partial accumulation result overflows, respectively. Instead of reducing weight magnitude during training to reduce accumulator magnitude during inference, we use structured N:M weight pruning [11] to restrict dot product lengths (i.e., the number of partial products) sufficiently to avoid most persistent overflows. We then avoid transient overflows that temporarily arise during inference via a dot product algorithm that reorders the partial products before accumulation (Section III-B). Figure 1 demonstrates our algorithm when accumulating a list of 20 partial products. The novel contributions of this paper are:

- Analysis of persistent and transient dot product overflow in quantized neural networks (Section III-A)
- Alternating Greedy Schedule (AGS) for eliminating transient overflows (Section III-B).
- Evaluation of the resulting methods in terms of model accuracy and accumulator compression for several neural

networks on classification tasks (Section IV).

II. Background

We consider uniform per-tensor quantization of both weights and activations to b-bit signed values [12]. The set of floating-point values in an activation matrix X has a range $R = max(X) - min(X)$. Unlike weights, activation ranges vary greatly during inference so an acceptable range R is typically derived from activation statistics collected during training [13]. To map values in X to integers in $[0, 2^b-1]$, we partition R into 2^b-1 uniform intervals of length $s_x = \frac{R}{2^b-1}$, also called the scale factor. For example, given an FP32 activation x^f, its quantized value $x^q = round(\frac{x^f}{s_x})$ maps x^f into $[0, 2^b - 1]$. Since the FP32 range of activations are asymmetric around 0 after ReLU (all positive), we shift x^q by an offset $o_x = -2^{b-1} - round(\frac{min(X)}{s_x})$ into the range $[-2^{b-1}, 2^{b-1} - 1]$, guaranteeing that the FP32 value for 0 maps to an integer. We can obtain the approximate FP32 representation of a quantized activation x^q by reversing the effect of the scale and offset via the equation $x^{f*} = s_x(x^q - o_x)$. The difference $|x^f - x^{f*}|$ is the quantization error. Analogous to activations, weights are quantized such that $w^{f*} = s_w(w^q - o_w)$.

Multiplication of an $M \times K$ weight matrix and $K \times N$ activation matrix consists of $M \cdot N$ dot products of length K. We perform quantized dot product using the FP32 approximations.

$$s_z(z - o_z) = \sum_{i=1}^{K} s_w(w_i^q - o_w)s_x(x_i^q - o_x) \tag{1}$$

where s_z and o_z represent the quantization parameters of output activations z. The FP32 scale factor terms can be factored out and normalized to an integer representation so the entire computation occurs in integer arithmetic [12]. In practice, neural network weights approximate a normal distribution symmetric about zero and popular neural network libraries fix $o_w = 0$ [12]–[15]. As a result, several terms under the summation disappear and the majority of computation arises from the integer dot product $z = \sum_{i=1}^{K} w_i^q x_i^q$.

III. Accumulating in Low Resolution

Consider the dot product $\sum_{i=1}^{K} w_i^q x_i^q$ that arises when weights and activations are both uniformly quantized to b bits. Assume we accumulate partial results into a p-bit register where each partial product $w_i^q x_i^q$ is $2b$-bits and $p > 2b$. This leaves $p - 2b$ bits leftover for precision during accumulation. Hence, our dot product may overflow when $K \geq 2^{p-2b}$. For 8-bit quantization and a 32-bit accumulator, the threshold $K^* = 2^{(32-2*8)} = 65536$ is high enough to avoid overflow in most popular neural networks. However, if we use a narrow accumulator e.g., $p = 2b$, overflows are possible after summing only 2 partial products. To reliably use low-resolution accumulators, we need a way of reducing such overflow.

A. Characterizing Overflows

We divide dot product overflows into two categories: **persistent** and **transient**. A persistent overflow occurs when the final dot product result overflows regardless of the order in which partial products were added. In other words, a persistent overflow is a true overflow where the final result is simply too large for the accumulator. Transient overflows arise when a partial result overflows but where the final result does not actually overflow the accumulator. They are 'temporary' overflows, a direct consequence of the order of partial products accumulation. Hence, we could potentially eliminate transient overflows by accumulating in some optimal order.

In practice, ML frameworks for quantized networks avoid overflow by either using high-precision accumulators (e.g., 32-64 bits) or clipping partial results into a finite range (saturation arithmetic) as they are accumulated [16]–[18]. We investigate how clipping of persistent and/or transient overflows impacts model accuracy while varying accumulator bit width. To this end, we trained a 1-layer MLP (linear + ReLU) on the MNIST dataset [19] with 8-bit weights and activations using QAT.

Clipping overflows results in poor model accuracy when using accumulators narrower than 18 bits (Figure 2b green). Assume we can resolve some dot product overflows by temporarily using a high-precision accumulator for those dot products. Figure 2a shows that at low resolutions of 13-16 bits, only 3-24% of overflows are transient while the rest are persistent (97-76%). However, if we resolve only the transient overflows while continuing to clip all persistent overflows, accuracy improves non-trivially from 10% to 40% (Figure 2b red). This suggests that model accuracy becomes more sensitive to clipping transient overflows, as opposed to persistent overflows, when accumulator resolution decreases.

When we decrease accumulator bitwidth, initially both the number of transient and persistent overflows will increase (17-20 bits). However, as we continue decreasing bitwidth, most overflows become persistent as the accumulator is too small to fit most dot product results. As a result, the number of transient overflows decreases (17 bits or fewer). (Note that an overflow is not considered transient if the final result also overflows). Beyond 13 bits, nearly all overflows will be persistent. If we prune the network, the number of terms in the dot product decreases, leading to a decrease in persistent overflows. However, several transient overflows will still remain and degrade accuracy if not resolved (See Figure 3 magenta).

A2Q [7] eliminates the possibility of both transient and persistent overflows by constraining the weight vector's L1-norm during QAT. They first bound the dot product result :

$$|\sum_{i=1}^{K} w_i^q x_i^q| \leq \sum_{i=1}^{K} |w_i^q||x_i^q| \leq 2^{p-1} - 1$$

In the worst case, all activations are maximal $|x_i^q| = 2^{b-1}$ and the weight L1-norm may be bounded such that:

$$\sum_{i=1}^{k} |w_i^q| = \|\mathbf{w^q}\|_1 \leq \frac{2^{p-1} - 1}{2^{b-1}}$$

This bound acts as an L1-regularizer and pulls most weight values toward zero, ensuring that partial sums never grow beyond p bits. L1 regularization promotes unstructured sparsity in the weight matrices, reducing the model size and enabling

acceleration by skipping zero computations. However, models with unstructured sparsity are more difficult to accelerate than structured sparse models on real hardware. Non-zero values may be arbitrarily distributed and must be addressed individually using indexing arrays, incurring computation and memory storage overhead [20], [21].

We find that enforcing strict bounds on weight magnitude is not necessary for using narrow accumulators. Instead, we reduce the number of weights in each dot product via N:M pruning. Then at inference time, we resolve transient overflows by optimally reordering the dot product, as transient overflows are simply a consequence of summation order.

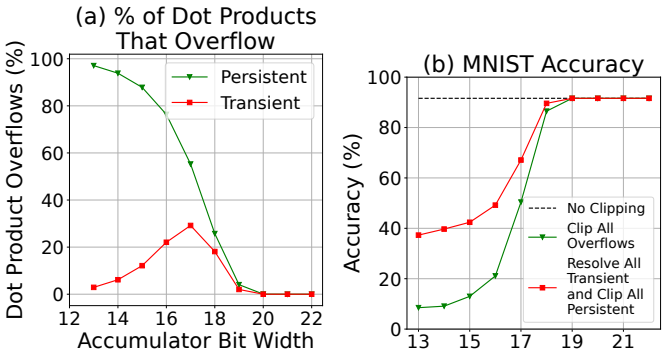

Fig. 2. Profile of overflows during MNIST inference of a 1-layer MLP with 8-bit weight/activations. Even though transient overflows only account for 3% of total overflows when using narrow 13-16 bit accumulators (a), resolving them improves accuracy from 10% to 40% (b) showing that for accumulators with low bitwidths, transient overflows can have a larger impact on accuracy.

B. Alternating Greedy Scheduling (AGS)

AGS optimally reorders dot product summation to avoid transient overflows that arise when adding the sequences of partial products. We first split the sequence into a positive *P* list and negative *N* list. We can then increase the running sum (by adding from the positive list) or decrease the sum (by adding from the negative list) at will. We prove that AGS (Algorithm 1) correctness by induction for the case of no persistent overflow.

Inductive Hypothesis: Initially, $z = 0$ and z satisfies the invariant MIN_VALUE $\leq$ z $\leq$ MAX_VALUE. Assume that after k iterations of the main loop, z satisfies the invariant. We must show that after iteration $k+1$, the invariant still holds. During an iteration, we may accumulate into z from either the positive or negative list.

Adding Positives: Assume we are accumulating into z from the positive list P in the current iteration $k+1$. Let $P[i_P]$ be the next positive number. We check if $\text{z} + P[i_P] >$ MAX_VALUE. If true, we stop adding and switch to the N list where additional sums are guaranteed to decrease z away from MAX_VALUE. Otherwise, we update z with $\text{z} = \text{z} + P[i_P]$. In both cases, z satisfies MIN_VALUE $\leq$ z $\leq$ MAX_VALUE at the end of iteration $k+1$.

Adding Negatives: Now assume instead we are accumulating into z from the negative list N in the current iteration $k+1$. Let $N[i_N]$ be the next negative number. We check if $\text{z} + N[i_N] <$ MIN_VALUE. If true, we stop accumulating and switch to the P list where additional sums are guaranteed to increase z away from MIN_VALUE. Otherwise, we update z with $\text{z} = \text{z} + N[i_N]$. In both cases, z satisfies the invariant at the end of iteration $k+1$.

By induction, MIN_VALUE $\leq z \leq$ MAX_VALUE for all iterations and AGS correctly avoids transient overflows.

Algorithm 1: Alternating Greedy Schedule

Input: List X containing K m-bit signed integers for which the accumulated sum does not overflow, i.e., no persistent overflow
Output: n-bit sum z without transient overflow
MAX_VALUE $= 2^{n-1} - 1$; MIN_VALUE $= -2^{n-1}$;
// Split X into positive (P) and negative (N) lists
for *each* $x \in X$ **do**
  if $x > 0$ **then**
    Append x to P;
  if $x < 0$ **then**
    Append x to N;
$z = 0$; $i_P = 0$; $i_N = 0$;
// Alternating summation b/w P and N
while $i_P < |P|$ *or* $i_N < |N|$ **do**
  while $i_P < |P|$ **do**
    if $z + P[i_P] >$ *MAX_VALUE* **then**
      break;
    z $\leftarrow$ z $+ P[i_P]$;
    $i_P \leftarrow i_P + 1$;
  while $i_N < |N|$ **do**
    if $z + N[i_N] <$ *MIN_VALUE* **then**
      break;
    z $\leftarrow$ z $+ N[i_N]$;
    $i_N \leftarrow i_N + 1$;
return z;

IV. Evaluation

In this section, we evaluate our framework in terms of accumulator compression and model accuracy for several neural networks. We then perform RTL simulations of the AGS algorithm to evaluate its impact on dot product throughput and latency when performing neural network inference in the context of practical CPU systems.

A. Software Library and Training Setup

Prior works have addressed the difficulty of analyzing transient overflows due to lack of support in standard deep learning frameworks [7], [8]. We extend PyTorch's quantization framework with custom linear and convolution layers implementing AGS to measure its impact on model accuracy. We unroll dot product computations allowing the user to vary weight, activation, and accumulator bitwidths and evaluate overflow solutions such as AGS, clipping, wraparound arithmetic. To our knowledge, our library is the first to enable fine-grained analysis of quantized dot products in neural networks.

We evaluate AGS using MobileNetV2 [22] and ResNet-18 [23] on CIFAR10 dataset [24]. We train these networks with N:M semi-structured sparsity followed by post-training uniform quantization via QAT. We iteratively prune all 2D-convolution and linear layers except the first 2D convolution and final linear classifier head. Every 10 epochs, we prune the smallest 10% of values within each consecutive group of $M = 16$ weights. For example in epoch 10, we set the smallest 2 out of 16 ($\approx 10\%$) values to 0 while in epoch 20, we prune such that 20% of weights are set to 0 (3 out of every 16 values). We uniformly prune every layer to the same sparsity (%) and enforce the same weight, activation, and accumulator bitwidths across each layer (e.g., 5/7/12 weight/act/accum bitwidths for all layers). Once the desired sparsity is reached, we continue training the network until a total of 200 epochs have elapsed.

B. Reducing Accumulator Bitwidth

In this section, we evaluate the ability of AGS to enable low-resolution accumulation while maintaining FP32 model accuracy in ResNet-18 and MobileNetV2. We sweep the design space by training several models with varying sparsity and weight/ activation/accumulator bitwidths. We vary weight and activations from 5 to 8 bits while varying the accumulator from 11 to 24 bits. We select the best performing models with the lowest required accumulator bitwidth to generate a pareto frontier. For models on the frontier, we use our software library to evaluate the accuracy impact when we clip dot product overflows instead of avoiding them via AGS.

Figure 3 shows that AGS can push the accumulator bit width lower than A2Q while also maintaining task performance. Weights are roughly 80-95% sparse. While pruning can reduce the length of individual dot products, transient overflows still arise during inference. The magenta lines show that clipping transient overflows within sparse dot products can limit how much we may reduce accumulator bitwidth. AGS allows us to avoid transient overflows and reduce accumulator resolution by ≈ 4 bits while maintaining accuracy.

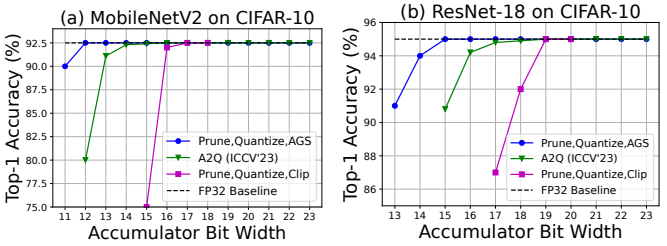

Fig. 3. We visualize the trade-off between accumulator bitwidth and accuracy. AGS (blue) can make use of accumulators with lower bitwidth than A2Q, without sacrificing significant accuracy.

C. AGS Hardware Evaluation

We estimate AGS's performance when running inference on a single CPU core via behavioral simulation of Algorithm I in RTL. Modern CPU's are equipped with wide data buses in the registers and cache (e.g., 512 bits) as well as SIMD units that can perform multiple operations in parallel. We assume the core's SIMD unit can multiply two 8-element vectors pairwise and write out 8 partial products over a wide bus in a single cycle (8 ops/cycle). To match the SIMD unit's processing rate, the AGS unit must be able to accumulate 8 partial products every cycle. When implementing AGS in RTL, we can separate 8 partial products into positive and negative lists in a single cycle. However, the main loop of the algorithm alternating summation between positive and negative lists runs sequentially and performs only a single add per cycle. Performing dot products using AGS alone would result in a reduced throughput of 1 op/cycle.

We mitigate sequential computation overhead by employing AGS only once the accumulator overflows. Specifically, we start by accumulating partial products using SIMD instructions. If the running sum overflows the accumulator, we invoke the AGS unit to sum the remaining partial products. We perform MobileNetV2 inference (5-bit weight, 7-bit activation) on a single batch of 100 CIFAR10 images and measure latency when using AGS with different accumulator bitwidths (Figure 4). The baseline represents inference latency when traditional SIMD instructions are used to perform dot products at a rate of 8 multiply-accumulate (MAC) ops/cycle. When bitwidth is 14 or more, AGS introduces minimal overhead and latency is similar to the baseline. However for narrower accumulators of 12-13 bits, many dot products overflow early in the partial product summation, triggering the sequential AGS algorithm and slowing down the computation up to 2.5x. We plan on improving performance of AGS hardware in future work.

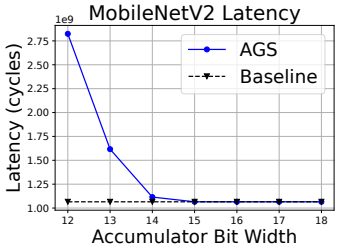

Fig. 4. We perform MobileNetV2 inference when using AGS with different accumulator bitwidths. At narrow bitwidths (12-13 bits), many dot products overflow early during partial product summation, triggering the sequential AGS algorithm and slowing down the computation as much as 2.5x.

V. Conclusion

In this work, we use structured N:M pruning [11] to reduce dot product length and avoid persistent overflows of the accumulator during quantized inference. Then, we show that by reordering dot product accumulations using AGS, we can avoid transient overflows as well. These techniques together form the proposed AGS method of this paper in achieving low-bitwidth accumulation of dot products with minimal accuracy degradation. Our evaluation results show that AGS reduces accumulator bitwidth from 32 bits to 12 bits and outperforms prior work by 2-5 bits. To the best of our knowledge, the AGS method and our analysis are novel in the literature.

Acknowledgments

This research was supported in part by the Air Force Research Laboratory under award number FA8750-22-1-0500, and in part by Meta Platforms Technologies under award number A51540.

REFERENCES

[1] J. Gomez, S. Patel, S. S. Sarwar, Z. Li, R. Capoccia, Z. Wang, R. Pinkham, A. Berkovich, T.-H. Tsai, B. De Salvo *et al.*, "Distributed on-sensor compute system for ar/vr devices: A semi-analytical simulation framework for power estimation," *arXiv preprint arXiv:2203.07474*, 2022.

[2] M. Scherer, M. Eggimann, A. D. Mauro, A. S. Prasad, F. Conti, D. Rossi, J. T. Gómez, Z. Li, S. S. Sarwar, Z. Wang, B. D. Salvo, and L. Benini, "Siracusa: A low-power on-sensor risc-v soc for extended reality visual processing in 16nm cmos," in *ESSCIRC 2023- IEEE 49th European Solid State Circuits Conference (ESSCIRC)*, 2023, pp. 217–220.

[3] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, pp. 11 711–11 722.

[4] A. Sabot, V. Natesh, H. T. Kung, and W. Ting, "MEMA runtime framework: Minimizing external memory accesses for tinyml on microcontrollers," *TinyML Research Symposium 2023*, vol. abs/2304.05544, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.05544

[5] *Cortex-M4 Technical Reference Manual*, Arm Limited, January 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/

[6] "Gap8 iot application processor," https://greenwaves-technologies.com/gap8_mcu_ai/, Nov 2023.

[7] I. Colbert, A. Pappalardo, and J. Petri-Koenig, "A2q: Accumulator-aware quantization with guaranteed overflow avoidance," *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 16 943–16 952, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261214350

[8] R. Ni, H. Chu, O. Castañeda, P. Chiang, C. Studer, and T. Goldstein, "Wrapnet: Neural net inference with ultra-low-precision arithmetic," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=3SqrRe8FWQ-

[9] B. de Bruin, Z. Zivkovic, and H. Corporaal, "Quantization of deep neural networks for accumulator-constrained processors," *Microprocess. Microsystems*, vol. 72, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:201895129

[10] I. Colbert, A. Pappalardo, J. Petri-Koenig, and Y. Umuroglu, "A2q+: Improving accumulator-aware weight quantization," 2024. [Online]. Available: https://arxiv.org/abs/2401.10432

[11] A. Zhou, Y. Ma, J. Zhu, J. Liu, Z. Zhang, K. Yuan, W. Sun, and H. Li, "Learning n: M fine-grained structured sparse neural networks from scratch," *ArXiv*, vol. abs/2102.04010, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231847094

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[13] P. Foundation, "Pytorch quantization," https://pytorch.org/docs/stable/quantization.html.

[14] Tensorflow, "Tensorflow lite quantization." [Online]. Available: https://arxiv.org/abs/2401.10432

[15] H. T. Kung, B. McDanel, and S. Q. Zhang, "Term quantization: furthering quantization at run time," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[16] *Arm Neon technology, the Advanced SIMD (Single Instruction Multiple Data) architecture extension for implementation of the Armv8-A or Armv8-R architecture profiles*, Arm Limited, June 2024. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/intrinsics/

[17] *ARM CMSIS Library*, Arm Limited, January 2022. [Online]. Available: https://developer.arm.com/tools-and-software/embedded/cmsis

[18] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2164, p. 20190155, Dec. 2019. [Online]. Available: http://dx.doi.org/10.1098/rsta.2019.0155

[19] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[20] E. Trommer, B. Waschneck, and A. Kumar, "dcsr: A memory-efficient sparse matrix representation for parallel neural network inference," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE Press, 2021, p. 1–9. [Online]. Available: https://doi.org/10.1109/ICCAD51958.2021.9643506

[21] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 548–560.

[22] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[24] A. Krizhevsky, "CIFAR-10 and CIFAR-100 datasets," https://www.cs.toronto.edu/~kriz/cifar.html.