MGS: Markov Greedy Sums for Low-Power DNN Accumulation

Vikas Natesh
Department of Computer Science
Harvard University
Cambridge, MA, USA
vnatesh@g.harvard.edu

H.T. Kung
Department of Computer Science
Harvard University
Cambridge, MA, USA
kung@harvard.edu

David Kong
Department of Electrical Engineering
Harvard University
Cambridge, MA, USA
dkong@g.harvard.edu

***Abstract*—Current deep neural network (DNN) systems avoid overflows in integer (e.g., 8-bit) dot products by performing accumulations on expensive high-precision hardware (e.g., 32-bit). Meanwhile, in floating-point summation, adding values with different exponents may lead to loss of precision in the mantissa of the smaller term, which is right-shifted to align with that of the larger term. To avoid precision loss from such shifting (a.k.a. 'swamping'), low-bitwidth floating-point dot products (e.g., FP8) are also accumulated in higher precision (e.g., FP32). We offer a novel approach for enabling low-power, low-bitwidth accumulation in both integer and floating-point DNN computations. We present Markov Greedy Sums (MGS), an architecture that exploits the distribution of DNN data to use narrow adders for the majority of dot product summations. MGS avoids overflows in integer dot products as well as swamping in floating-point dot products. In contrast to prior works for narrow accumulation, MGS does not require modification of the underlying DNN model, e.g., re-training. We implement MGS in custom multiply-accumulate (MAC) units and integrate them into a systolic array accelerator. Our evaluation across several models and datasets shows that MGS significantly reduces accumulator bitwidth and improves energy efficiency over conventional MAC designs.**

I. Introduction

Quantization has become a ubiquitous optimization for compressing deep neural networks (DNNs) on both low-power edge devices [21], [30], [37], [39] as well as large-scale training and inference systems made up of many GPUs [35]. Low-power devices for tinyML typically have small local memories [6] and often lack support for efficient floating-point computation [3], [39]. Hence, integer quantization is, by default, necessary on such systems, and most tinyML models are quantized to 8 bits or less. Meanwhile, large generative AI workloads push GPU-based training and inference clusters to the limits of available memory, bandwidth, and computation power. To this end, low-bitwidth formats such as brain float-16 (bfloat16) [43], block floating point (BFP) [44], and 8-bit floating-point (FP8) have been implemented in various hardware [1], [2]. Such formats have been successful in reducing memory footprint, memory accesses, computation time, and power consumption [1], [2], [32]

When performing quantized matrix multiplications, dot products are typically accumulated into wider registers. For instance, partial products in FP8 may be accumulated in FP16 or FP32 to ensure numerical accuracy. Modern large language models (LLMs) increasingly rely on high-dimensional dot products, with vector lengths regularly exceeding 10000 elements. Figure 2a demonstrates the trend that over the past 7 years, newer LLMs perform longer dot products to support longer context lengths. For example, Qwen and Llama-3.1 models contain dot products in the feedforward blocks (FFN) that exceed 25000 elements. Reducing accumulator bitwidth for such long dot products can significantly reduce bandwidth and energy usage while increasing inference throughput [11]–

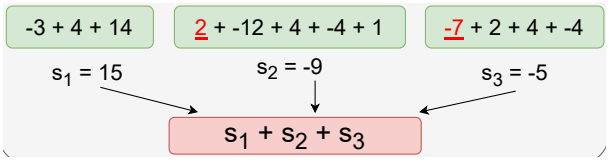

Fig. 1: Example of Markov Greedy Sums (MGS). In (a), we sum 12 ints into a narrow accumulator (green box) until the sum s_i overflows the range [-15, 15]. Then, we accumulate s_i into a wider accumulator (red box). The underlined red values are those that would have caused an overflow of the narrow accumulator, noting that $15 + 2 > 15$ and $-9 - 7 < -15$.

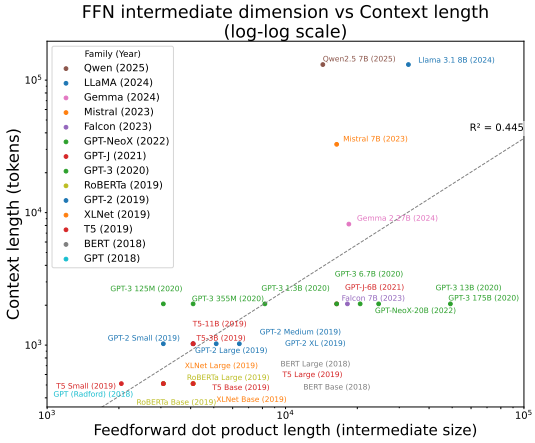

Fig. 2: Context length vs. feedforward layer dot product length for a variety of language models from 2018-present (log-log scale)

[13], [34]. 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 one can reduce the accumulator bitwidth. In addition, there is the swamping problem [24] that causes loss of precision due to right shifting of significant digits when floating point numbers need to align their mantissa before summation.

Prior works enable use of narrow accumulators while avoiding overflow by re-training the network, e.g., loss function regularization [34] or controlling weight magnitude during training [11]–[13]. Re-training LLMs is expensive and may be impossible if access to the original training data is unavailable. In addition, re-training may damage properties of the pre-trained model, such as fairness guarantees [42]. Other works reorder dot product summations to avoid the majority of overflows when using narrow accumulators [33]. However, reordering requires additional sort/permute operations as well as memory for temporary storage and is difficult to optimize on existing hardware.

We propose **M**arkov **G**reedy **S**ummation (MGS), a novel approach to enable low-precision accumulation in neural network dot products without the need for retraining or summation reordering. We analyze overflows during neural network inference and model the value of the partial sum in dot products as a Markov process to derive the expected dot product length without overflow. Our key insight is that based on the statistical properties of weight and activation distributions, we can sum many partial products in reduced precision before overflow occurs. MGS is greedy in the sense that it uses a narrow accumulator to accumulate as many values as possible while falling back on a wider accumulator when the rare overflow occurs. Leveraging this insight, we design dual-multiply-accumulate (dMAC) hardware units that use narrow accumulators for the majority of sums. Our method uses a narrower average accumulator bitwidth compared to prior works when performing DNN computations. Our dMAC units consume up to 64% less energy than conventional integer and floating-point MACs that use wide accumulators for all summations. Figure 1 provides an overview of MGS applied to integer summation. The novel contributions of this paper are:

- Analysis of dot product overflows in integer and floating-point quantized neural networks (Section III).
- Dual-MAC hardware architecture (dMAC) and algorithm for avoiding overflows when using narrow accumulators in quantized integer and floating-point dot products (Sections and IV and V).
- Evaluation of our methods w.r.t. accumulator compression for several data types and DNNs (Section VI).
- Energy consumption and area evaluation of dMAC units compared to conventional integer and floating point MACs when implemented in a 7nm process node. (Section VI)

II. Background

We present background on both integer and floating-point quantization of DNNs and some prior work on avoiding overflow during DNN execution.

A. Integer Quantization

We consider the uniform quantization of weight and activation matrices per-tensor from FP32 to b-bit signed values [27]. The floating-point values in a matrix M have a range $R = max(X) - min(X)$. To map values in M 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, we can map a FP32 activation x to a value x^q in $[0, 2^b - 1]$ using the equation $x^q = round(\frac{x^f}{s_x})$. If the range is asymmetric around zero, 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 $x^* = s_x(x^q - o_x)$. Quantized dot products are then performed using the FP32 approximations.

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

where s_w, o_w, s_z, and o_z represent the quantization parameters of weights w and output activations z. Floating point scale factors are factored out and normalized to an integer representation, while weights are typically symmetric around zero with $o_w = 0$ [19], [27], [29], [41]. As a result, several terms under the summation disappear, and the majority of computation arises from the integer dot product $z = \sum_{i=1}^{K} w_i^q x_i^q$.

When FP32 weights and activations are quantized to low-precision (e.g., INT8), the computation cost of multiplications $w_i^q x_i^q$ decreases significantly. However, the compute bottleneck transitions to the K dot product summations, as these accumulations are typically executed in higher precision, such as 32-bit, to avoid overflow of the accumulator. For example, assume we accumulate using 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, the dot product overflows when $K \geq 2^{p-2b}$. However, if we use a narrow accumulator $p = 2b$, overflow may occur during any of the K partial sums, leading to inaccurate dot product and poor model accuracy.

Previous works enable the use of narrow accumulators in DNN computations by retraining the network to reduce partial sum magnitude [11], [12], [34] or algorithmically avoiding most overflows [33]. In practice, ML frameworks for quantized DNNs 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 [5], [7], [20]. Clipping is cheap to implement in hardware or software, allowing for a modest reduction in accumulator precision, e.g., from 32 to 16 bits. However, for narrower bitwidths

(< 16), clipping severely degrades numerical accuracy and task performance [11], [33].

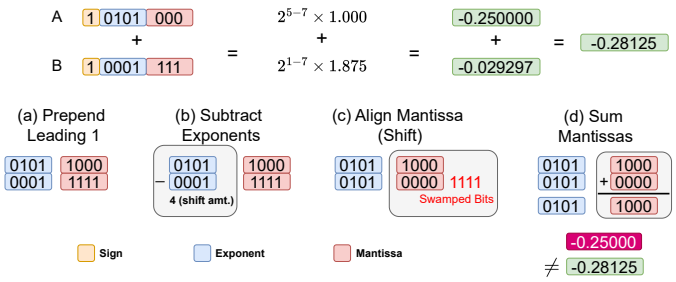

Fig. 3: An example of mantissa bit swamping when adding two E4M3 values with different exponents, $A = -0.25$ and $B = -0.029297$, while using a narrow 4-bit accumulator. The exponent bias in E4M3 is 7. A's exponent of 5 is larger than B's exponent 1 (b), causing B's mantissa to be shifted left by $5 - 1 = 4$ bits (c). Since the entire mantissa shifts out, B is treated as zero, and the final result is -0.25, differing from the closest FP8 result of -0.28125 (d).

B. Floating-Point Quantization

FP32 DNN weights and activations may be quantized to lower-precision floating-point formats such as bfloat16, BFP, FP16, or FP8. In particular, FP8 formats for both inference and training have been developed and implemented on several commercial AI accelerators, such as Nvidia H100 GPUs and Intel Gaudi2 [1], [2], [32]. Such formats are now widely used and can achieve baseline FP32 performance on large AI workloads, such as LLMs [14], [35].

FP8 summation involves several steps as shown in Figure 3. Consider the E4M3 format with one sign bit, four exponent bits, and three mantissa bits [32]. When adding two FP8 values with different exponents, the lower order bits of the smaller value are shifted out ('swamped') due to right-shifting to align exponents with the larger value. This leads to a loss of precision in the final sum. In contrast to floating-point formats with wide mantissas, narrow formats such as E4M3 suffer from a significant loss in numerical accuracy due to swamping. Commercial hardware such as the H100 avoids swamping by accumulating FP8 partial sums in a wider precision such as FP16 or FP32 [2].

There are several classical algorithms for reducing swamping error in floating point summation, including pairwise summation [24] and Kahan summation [28]. Although Kahan summation has higher accuracy, it requires several extra floating point operations to maintain the compensated error term. Meanwhile, pairwise summation is efficient to implement but suffers from large error in narrow floating-point formats.

Figure 4 illustrates the need for high-precision accumulation of FP8 dot products. Using several summation algorithms, we perform dot products between two Gaussian vectors in FP8 precision (4-bit mantissa accumulator) and plot the numerical error relative to the baseline FP32 accumulation (24-bit mantissa accumulator). All algorithms exhibit significant errors due to the swamping of lower order bits when using narrow 4-bit accumulators. Sequential summation loses all accuracy after only 200 sums. Pairwise summation is significantly more accurate than sequential summation but still exhibits up to 50% error for longer dot products. In Section V-B, we discuss how to accumulate FP8 mantissas in low-precision for a majority of sums while attaining numerical accuracy on-par with FP32 accumulation.

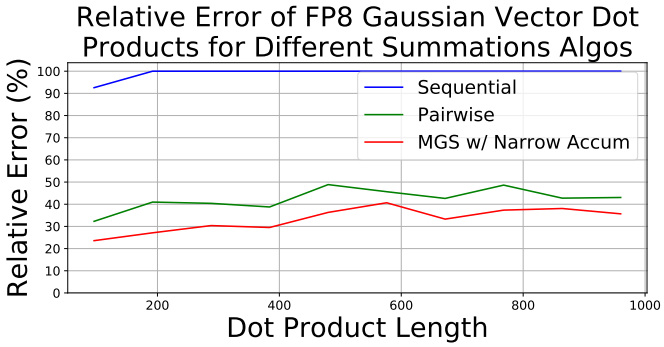

Fig. 4: % Error, relative to FP32 precision, of Gaussian vector dot products performed in FP8 precision. We execute each algorithm using solely a narrow accumulator and clip partial sums upon overflow. All algorithms exhibit significant errors due to the swamping of lower order bits when using reduced-precision accumulators. MGS has lower error than pairwise summation by separating partial product mantissas by exponent and accumulating them in separate narrow accumulators. This means that dot product errors result only from clipping overflows. However, the $\approx 35\%$ error of MGS, when restricted to a narrow accumulator, is unacceptable for DNN applications.

III. Analysis of Dot-Product Overflows

We begin by providing an analytical framework for reasoning about overflows. We define two types of integer overflow and discuss multiple algorithms for avoiding them.

Transient Overflow: Overflow that may occur at any point during the sequential summation of k integers $X = \{x_1, x_2, ..., x_k\}$ when using a b-bit accumulator.

Persistent Overflow: Overflow that occurs when the final sum $y = \sum_{i=1}^{k} x_i$ overflows a b-bit accumulator.

Note that transient overflows may occur even when there is no persistent overflow. We aim to minimize these transient overflows.

A. Avoiding All Overflows

Several prior works aim to avoid both persistent and transient overflows entirely by retraining the neural network such that partial sums are always within the accumulator bounds. A2Q [11] and A2Q+ [12] eliminate the possibility of both transient and persistent overflows by constraining the weight

vector's L1-norm during quantization-aware training (QAT). They first bound the dot product result :

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

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

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

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, network sparsification may reduce model accuracy [31] Meanwhile, retraining a pre-trained DNN to satisfy accumulator constraints may alter properties of the pre-trained model, such as algorithmic fairness guarantees [42]. We find that enforcing strict bounds on weight magnitude is not necessary for using narrow accumulators.

B. Avoiding Transient Overflows

Persistent overflow is a true overflow where the final result is simply too large for the accumulator. Transient overflows are 'temporary' and arise when a partial sum overflows but where the final sum may not actually overflow the accumulator. Hence, in the absence of persistent overflow, we should be able to eliminate transient overflows by reordering the summation.

Theorem 1. *Let $X = \{x_1, x_2, ..., x_k\}$ be a list of k signed integers, where each x_i is represented using n bits. Let $y = \sum_{i=1}^{k} x_i$ be the sum of all elements in X, representable using $m \geq n + 1$ bits without persistent overflow (i.e., $-2^{m-1} \leq y \leq 2^{m-1} - 1$). Then, there exists an ordering of summation for X that avoids transient overflow when using an m-bit accumulator.*

Proof. Suppose $k = 2$. The list $X = \{x_1, x_2\}$ contains n-bit numbers, and its sum $x_1 + x_2$ (or $x_2 + x_1$) can require at most $n+1$ bits. Since $m \geq n+1$, the sum does not overflow m-bits, and the theorem holds for $k = 2$.

Let $l \geq 2$ and assume the theorem holds $\forall k \leq l$, i.e., there exists an ordering of $X = \{x_1, x_2, ..., x_l\}$ such that the sum of elements of X w.r.t. said ordering avoids transient overflow (inductive hypothesis). Denote this ordering by the index set α_l and the so-ordered list by X_{α_l}. Suppose that $k = l + 1$ and $X = X_{\alpha_l} \cup x_k$. Then,

$$y = \sum_{i=1}^{k} x_i = x_k + \sum_{i \in \alpha_l} x_i$$

By the inductive hypothesis, the second term in the sum, denoted by $\hat{y} = \sum_{i \in \alpha_l} x_i$, avoids transient overflow. Since $\hat{y}$ is an m-bit signed integer, and x_k is an n-bit signed integer with $n \leq m - 1$, the sum $y = x_k + \hat{y}$ is represented by at most m bits. Therefore, a feasible ordering to avoid transient overflow is $\alpha_k = \{\alpha_l, k\}$. Thus, by induction, our theorem must hold for any $k \geq 2$. □

The proof shows how to construct a summation sequence without transient overflow by building the 'right' permutation sequence at each step. One example of such an algorithm is first to sort the k values, divide them into a list of negative values and a list of positive values, and repeatedly form the sum of the largest positive and most negative values. We can then take the resulting list, with length at least $k/2$, and apply the algorithm recursively until a single pair of values remains. This method is guaranteed to avoid transient overflow while using the narrowest possible accumulator as the running sum increases monotonically. Performing summations in a sorted order is also beneficial for retaining floating point accuracy since adding pairs of values of similar magnitude reduces the number of bits swamped in the smaller value [16]. However, sorting before adding becomes expensive in DNN applications with long dot products.

AGS is a recent method to avoid transient overflow by reordering in integer-quantized DNNs [33]. AGS first splits the sequence by sign into a positive list and negative list, then alternates summing values from either the negative or positive list depending on whether the accumulator overflows its maximum or minimum value, respectively. This allows AGS to avoid transient overflows while also avoiding sorting and using only an extra bit for overflow detection. However, AGS may require additional registers or memory to buffer partial products. For example, once an overflow is detected, AGS may need to buffer several positive values while waiting for a negative partial product to arrive. The extra memory requirements may overwhelm the benefits of using a narrow accumulator, challenging AGS hardware implementation.

IV. Markov Greedy Summation

In this section, we detail how our proposed MGS avoids all overflows while using narrow accumulators for the majority of summations. We first analyze MGS on dot products in integer-quantized CNNs such as MobileNetV2 (Section IV-A). We show that since weights and activations are normally distributed or half-normally distributed, the chance of overflow during summation is actually low. We then derive the expected number of summations before overflow by modeling the running sum as a random walk. Then in Section IV-C, we apply our analysis to FP8 mantissa accumulation in MobileNetV2, GPT-2, and LLAMA-3.2. Although mantissa distributions vary widely across models and quantization schemes, we are still able to accurately model dot product overflow and show that MGS performs the majority of summations using a narrow mantissa adder. We note that our analysis may be applied to other data types such as FP4 (Section VI).

A. Estimating the Probability of Integer Overflow

We consider b-bit quantized neural network dot products $Z = \sum_{i=1}^{k} w_i x_i$ with weights and activations in the range $[-2^{b-1}, 2^{b-1}]$. Weight and input activation vectors w and x

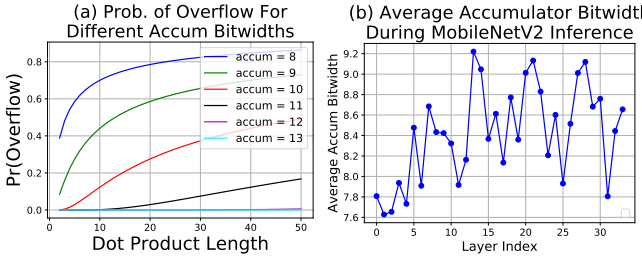

Fig. 5: (a) We estimate the probability of overflow based on the model described in Section IV-A, when performing dot product at different accumulator bitwidths. 5-bit Gaussian weights in the range[-15,15] are multiplied with 7-bit Gaussian activations in [-63,63] to yield partial products $Z \approx N(0, k * \sigma_w \sigma_x)$. We set σ of weights and data such that the extreme values lie 3 σ's away from the mean 0, i.e., $\sigma_w = 15/3 = 5$ and $\sigma_x = 63/3 = 21$. The figure shows that despite 7+5=12-bit partial products, we can use accumulators with ¡ 12 bits for most sums before overflow. For example, there is only a $\approx$ 12% chance of overflow when summing 10 elements in a narrow 10-bit accumulator. In (b), we plot the average accumulator bitwidth when running MobileNetv2 inference with 5-bit weights and 7-bit activations. Although one would expect that at least 5+7=12 bits are required to prevent overflow, the average accumulator bitwidth required varies between 7 and 10 bits.

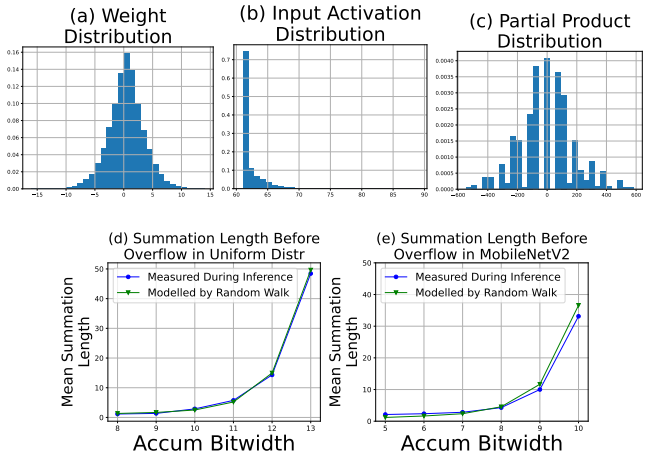

Fig. 6: Plotting the empirical measured average dot product length versus expected dot product length based on our random walk model. 5-bit Weights follow a normal distribution in the range [-15, 15] (a), while 7-bit activations have a half-normal distribution in the range [0,127] after ReLU (b). Note that the plot shows that with the accumulation bitwidth equal to 10, we do not expect overflow at a summation length of about 32 (e). In contrast, a naive analysis would conclude that 17 = 5+7+5 bits are required to avoid overflows, noting that $5 = \log_2 32$. Applying our analysis to the case of dot products between 5-bit and 7-bit uniformly distributed vectors, (d) shows that we can accumulate up to 14 values using 12-bits and 5 values using 11-bits before overflow, on average

are truncated, zero-centered i.i.d normal distributions $N(\mu_w = 0, \sigma_w)$ and $N(\mu_x = 0, \sigma_x)$, respectively. Input activations may also be half-normal distributions due to ReLU operations in the previous layer. The partial products $p_i = w_i x_i$ are i.i.d product-normal distributions with $\mu_p = 0$ and $\sigma_p^2 = (\sigma_w^2 + \mu_w^2)(\sigma_x^2 + \mu_x^2) - \mu_w^2 \mu_x^2 = \sigma_w^2 \sigma_x^2$. The summing of partial products can be represented by the random variable $Z = \sum_{i=1}^{k} p_i$. By the central limit theorem (CLT), for large enough k, $Z \approx N(0, k * \sigma_w^2 \sigma_x^2)$. This enables us to approximate the probability of overflow given a particular dot product length k and accumulator bitwidth a.

$$Pr(|Z| > 2^{a-1}) \approx 2\Phi\left[\frac{-2^{a-1}}{\sigma_w \sigma_x \sqrt{k}}\right]$$

where Φ is the CDF of the standard normal distribution. Figure 5a displays the probability of overflow for different vector lengths and accumulator bitwidths when performing dot product with 5-bit weights and 7-bit activations. The figure shows that for relatively long dot products, such as 10 or 15 elements, the chance of overflow is relatively low, even for narrow accumulators. In Figure 5b, we empirically observe that the average accumulator bitwidth is small across DNN layers, suggesting that wide accumulators may not be needed for a majority of sums.

B. Computing the Expected Number of Overflows

The approximation above provides a loose bound showing that overflow is relatively rare, even with narrow accumulators.

We can derive the expected number of summations before overflow more precisely by modeling summation as a random walk, specifically a Markov chain with a single absorbing state representing overflow.

To illustrate the idea, consider the summation of integers from the range [-2,2] using an accumulator that can only hold values in [-2,2]. In each step, we randomly select an integer from the range [-2,2] and add it to the accumulator. We stop when the accumulator overflows out of range [-2,2], i.e., we enter the absorbing state. Once entered, the process cannot leave the absorbing state. Hence, the random walk will eventually end as the accumulator is permanently absorbed into an overflow state. A 6×6 transition matrix P represents the probabilities of entering different states given the current sum, with each row summing to 1.

$$P = \begin{array}{c} \\ \text{Input State} \end{array} \begin{array}{c} \begin{array}{cccccc} \multicolumn{6}{c}{\text{Output State}} \\ -2 & -1 & 0 & 1 & 2 & \text{Ovfl} \end{array} \\ \begin{bmatrix} 1/5 & 1/5 & 1/5 & 0 & 0 & 2/5 \\ 1/5 & 1/5 & 1/5 & 1/5 & 0 & 1/5 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 & 0 \\ 0 & 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \\ 0 & 0 & 1/5 & 1/5 & 1/5 & 2/5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{array}{c} -2 \\ -1 \\ 0 \\ 1 \\ 2 \\ \text{Ovfl} \end{array}$$

For example, the 5th row represents the probability of different output states given the starting accumulator value of 2. The value 2 may be summed with either 1 or 2 with

probability 2/5 as both are equally likely to be the next state (uniform random draws). Since 2+1=3 and 2+2=4 both overflow the accumulator, we enter the overflow state (ovfl column) with probability 2/5. The last row shows that if we start in an overflow state, we will remain in that state surely.

We can represent the transition matrix P in a blocked form:

$$P = \begin{pmatrix} Q & R \\ 0 & I \end{pmatrix}$$

where 0 is a zero matrix and I is the identity matrix. R represents the transitions from transient states to absorbing states, and Q represents the transitions between transient states. To compute the transition probabilities after k steps, we simply multiply P by itself k times.

$$P^k = \begin{pmatrix} Q^k & R + QR + ... + Q^kR \\ 0 & I \end{pmatrix} = \begin{pmatrix} 0 & (I-Q)^{-1}R \\ 0 & I \end{pmatrix}$$

$Q^k = 0$ reflects the fact that the random walk will eventually end, i.e., eventually there is zero probability of being in a non-absorbed state. The fundamental matrix of the Markov chain $N = (I - Q)^{-1}$ represents the expected number of visits to non-absorbing state $j \in [-2, 2]$ starting from non-absorbing state $i \in [-2, 2]$, before absorption. The accumulator starts with the value 0, varies across different non-absorbing states with each partial sum, and eventually overflows. The expected number of steps to reach overflow is simply the sum of the entries in row 3 of N, corresponding to the state 0. This sum represents the total expected number of visits to all non-absorbing states before absorption, i.e., the total expected number of sums we may perform before overflow.

We apply our random walk model to dot product accumulation when executing quantized MobileNetV2 inference on Imagenet1K with 5-bit weights and 7-bit activations [15], [38]. Since weights and activations may deviate slightly from normal, we compute transition probabilities using their empirical distributions during DNN inference. Figure 6 plots the empirical versus modeled average summation length before overflow in a 1x1 convolution layer in the 13th residual block. When summing partial products derived from multiplying 5-bit weight and 7-bit activations, we expect that 5+7=12 bit accumulation is required. However, Figure 6 shows one may use a narrow 9-bit accumulator to sum 10 values before needing to use a wider accumulator, on average.

Even under weaker distributional assumptions, we can still significantly reduce the average accumulator bitwidth. Consider a worst-case scenario where 5-bit weights and 7-bit activations after ReLU are uniformly distributed in [-16,15] and [0,127], respectively. Despite this, our analysis/experiments demonstrate that we can accumulate 14 values using 12-bits and 5 values using 11-bits before overflow, on average (Figure 6d).

C. FP8 Mantissa Accumulation

Similar to integer-quantized DNNs, weight, activation, and partial product values of FP8 DNNs follow well-defined distributions (see Figure 6). The distribution of partial product mantissas during FP8 dot product accumulation varies depending on these underlying weight and activation distributions.

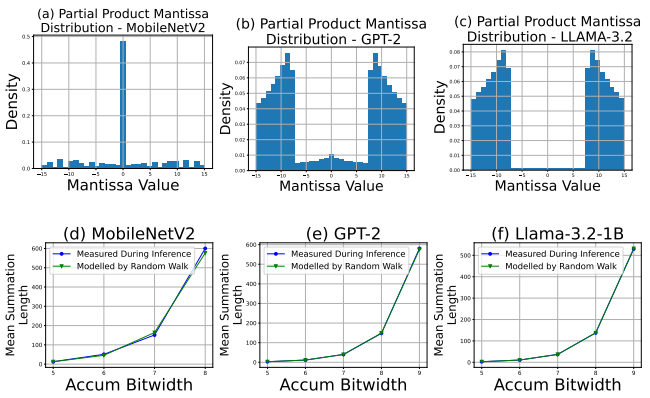

Fig. 7: Distribution of partial product mantissas during dot product in MobileNetV2, GPT-2, and Llama-3.2 (a,b,c). Despite the variability in mantissa distributions across DNNs, MGS precisely models the expected number of summations before overflow within 1% of the measured summation length (d,e,f).

Consider the E4M3 FP8 format with 3 mantissa bits. Before summation, we prepend a leading 1 to the partial product mantissa if that FP8 value is not subnormal. After including the sign bit, the mantissas are in the range [-15,15] with subnormal mantissas occupying the sub-range [-7,7]. However, the actual distribution of the mantissa varies significantly between models, layers within the same model, and quantization schemes. For example, since MobileNetV2 activations have high sparsity due to ReLU (Figure 6b), around 40% of FP8 partial product mantissas are also 0 (Figure 7). In the case of LLMs, the presence of outliers and FP8 quantization methods such as per-tensor scaling [4], [32] yield partial product mantissa distributions in Figures 7b and 7c, with a smaller fraction of values in the subnormal range relative to MobileNetV2.

Following the analysis of Section IV, we can model mantissa summation as a Markov chain and compute transition probabilities using the empirical partial product mantissa distributions during DNN inference. Figures 7d, 7e, and 7f compare the empirical versus modeled average summation length before mantissa overflow for MobileNetV2, GPT-2, and Llama-3.2 networks, respectively. Our model estimates the expected summation length within 1% of the true measured length before overflow across all networks.

V. Dual-Accumulator MAC Design

In this section, we describe the hardware for dual-multiply-accumulate (dMAC) units, leveraging our observation that the majority of dot product sums do not overflow when using narrow accumulators. We first introduce the dMAC for integer dot products and then show how this design enables narrow accumulation in FP8.

A. Integer dMAC

The integer dMAC unit uses a narrow adder (green in Figure 8) for most summations and a wide adder (red) to handle partial sums that overflow the narrow adder. It has a slightly higher area overhead than a conventional MAC unit, containing two adders and additional overflow handling logic. However, dMAC consumes significantly less dynamic power by exploiting the low overflow rate in DNN dot products. In addition, we clock-gate the wider accumulator to reduce dynamic power usage further when not performing wide accumulations.

Figure 8 displays our integer dMAC design when multiplying 4-bit weights and activations using 8-bit and 32-bit adders. After multiplication, the product p is accumulated in an 8-bit register a_8. If the 8-bit adder's carry-out overflow flag is set, we accumulate a_8 in the wider 32-bit register a_{32} instead and write p to a_8. Once all the partial products have been accumulated, we add a_8 and a_{32} and return the output.

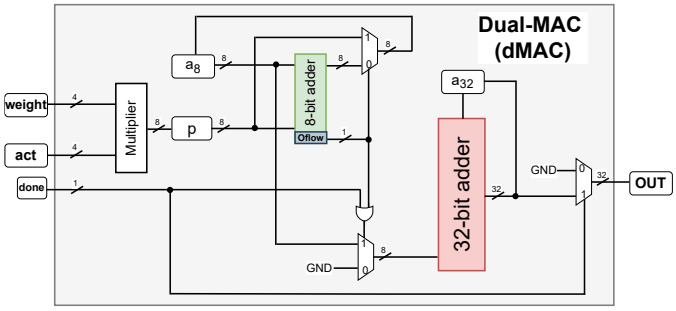

Fig. 8: Dual accumulator MAC hardware unit (dMAC) with output-stationary behavior. In this example, 4-bit weights and data arrive for multiplication and summation with an 8-bit accumulator a_8. If an overflow occurs (oflow = 1), a_8 is summed into the wider 32-bit accumulator a_{32}, and the 8-bit partial product is written to a_8. Upon completing the dot product (done = 1), we return the sum of a_8 and a_{32}.

B. 8-bit Floating-Point dMAC

Existing hardware for FP8 MAC operations accumulate partial products in higher precision such as FP32 [1], [2] This not only requires the use of wide mantissa adders but also FP8-¿FP32 data conversions and FP32 normalizations. Figure 9 provides a high-level view of the difference between our hardware and existing FP8 MAC units. We show that using dMACs for mantissa accumulation can avoid several expensive operations in wide registers while maintaining numerical accuracy.

Figure 10 displays our FP8 dMAC design. A new weight and activation in the E4M3 format arrive each cycle. After multiplication and rounding, the partial product sign bit converts the 4-bit mantissa (with leading 1) to 5-bit signed 2's complement. Using a narrow 5-bit adder, we then accumulate the mantissa into one of 16 5-bit registers based on its 4-bit exponent, which ranges from 0 to 15. By accumulating mantissas of the same exponent in the same register, we avoid

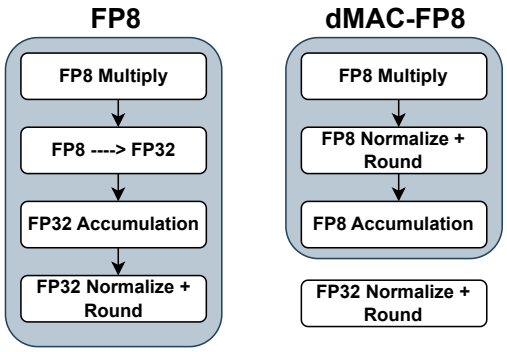

Fig. 9: High-level view of operations in our dMAC-FP8 unit versus conventional FP8 MACs. The gray boxes represent the operations that must occur every time a pair of values arrives. Conventional FP8 incurs overhead from data conversion and wide accumulation and normalization operations. In contrast, dMAC-FP8 performs the majority of computation in narrower precision while amortizing the cost of normalization across multiple partial summations.

the shifting operations required when adding two FP8 values with differing exponents while also avoiding numerical error from swamping (see 3).

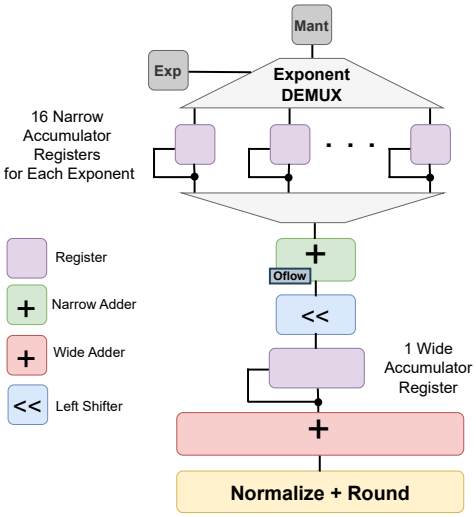

Fig. 10: FP8 dMAC hardware unit. We display the FP8 accumulation step in the computation. A new partial product arrives every cycle. Partial product mantissas are written to one of 16 registers such that they are always accumulated with other values of the same exponent, thereby preventing swamping while enabling the use of the narrow accumulator (green) for most summations. When the narrow accumulator overflows, it is left-shifted and summed with the wide accumulator (red) to prevent precision loss.

When the 5-bit adder overflows, we left-shift the accumulator by its exponent and accumulate into a wide 32-bit accumulator. Left-shifting by the exponent forces the 5-bit accumulator value to have exponent zero, allowing partial

sums with different exponents to be added into the same wide register without error. Since overflows are rare, dMAC amortizes the shifting cost of mantissa alignment over several summations instead of between each pair of elements as in conventional FP32. Once the dot product is complete, the values in each accumulator register are left shifted by their respective exponent and summed into the 32-bit accumulator. This $16\times$ shift+add operation is only performed once per dot product. Finally, the result is normalized, rounded, and returned.

VI. Evaluation

We evaluate MGS in terms of accumulator compression and power consumption on several vision and language models. In Section VI-B, we compare MGS to state-of-the-art (SOTA) works in accumulator compression when performing inference on various DNNs. We show that MGS can significantly reduce accumulator bitwidth while achieving accuracy on par with FP32 baselines, without retraining. We include the accuracy metric only for comparison with prior works noting that the numerical result of dot products performed with dMACs is the exact same as that on conventional MACs. In Section VI-C, we implement dMAC units for INT8, FP8, and FP4 in a 7nm node and measure power consumption relative to conventional MACs. Then, in Section VI-D, we integrate dMAC units into an output-stationary systolic array accelerator and estimate the energy efficiency when running inference on several LLMs. Our dMACs can reduce inference power consumption by up to 64%. While our evaluation focuses on INT8, FP8, and FP4 inference, MGS may also be applied to other data formats to reduce accumulator bitwidth, e.g., during training with the E5M2 FP8 datatype.

A. dMAC Emulation Library

Prior works have addressed the difficulty of efficiently profiling overflows due to lack of support in standard deep learning frameworks [11], [34]. We have built a C++/CUDA library to emulate dMACs on both CPUs and GPUs to run experiments quickly. We extend PyTorch's quantization framework with custom linear and convolution layers implementing MGS for INT8, FP8, and FP4 quantization to measure the impact on model accuracy. We unroll dot product computations, allowing users to vary weight, activation, and accumulator bitwidths and evaluate overflow solutions such as MGS, clipping, or wraparound arithmetic.

B. Reducing Accumulator Bitwidth

In this section, we evaluate the ability of MGS to enable low-resolution accumulation while maintaining FP32 model accuracy in MobileNetV2 [25], ResNet-18 [22], and ViT [17] on ImageNet [15].

1) 8-Bit Integer Quantized DNNs: We sweep the design space by varying weight and activations from 5 to 8 bits while varying the accumulator bitwidth from 8 to 20. 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 accuracy compared to SOTA methods A2Q [11], A2Q+ [12], AGS [33], and overflow clipping [5], [7], [20].

TABLE I: Average FP8 Mantissa Accumulator Bitwidth

MAC Unit	MobileNetV2	ResNet-18	ViT-Large	GPT-2	Llama-3.2	Phi-3-mini-4k
FP32	24	24	24	24	24	24
FP16	11	11	11	11	11	11
Bfloat16	16 or 24	16 or 24	16 or 24	16 or 24	16 or 24	16 or 24
FP8-MAC	11 or 24	11 or 24	11 or 24	11 or 24	11 or 24	11 or 24
FP4-MAC	11 or 24	11 or 24	11 or 24	11 or 24	11 or 24	11 or 24
FP8-dMAC	**7**	**7**	**8**	**8**	**8**	**8**
FP4-dMAC	**4**	**4**	**4**	**4**	**4**	**4**

Figure 12 shows that MGS can push the accumulator bit width lower than A2Q+ while also maintaining task performance. The magenta lines show that clipping transient overflows within dot products can limit how much we may reduce accumulator bitwidth. AGS accurately avoids transient overflows but clips persistent overflows, leading to accuracy drops at lower bitwidth where clipping becomes more prevalent.

2) Floating-Point Quantized DNNs: We evaluate MGS when performing floating-point inference using our target models. For FP8, we employ the E4M3 datatype and vary the narrow mantissa accumulator bitwidth from 5 to 10 bits while for FP4 we use the E2M1 datatype and vary the narrow accumulator bitwidth from 3 to 6 bits. Typically, E4M3 and E2M1 dot products are accumulated in FP16 or FP32 precision using a 11-bit or 24-bit mantissa, respectively. Table I shows that MGS is able to use a narrower mantissa accumulator than conventional FP4, FP8, FP16, and FP32 MAC units. On average, MGS enables use of the narrow accumulator for $\approx 90\%$ of summations.

C. dMAC ASIC Physical Implementation

We compare the ASIC implementations of various MAC designs for accurate power, performance and area characterization. We perform the full physical implementation of the designs at the 7 nm node using the ASAP7 PDK [9] with a 0.7 V supply voltage. To balance switching speed and power consumption, we implement the design using standard threshold voltage transistors targeting a clock frequency of 500 MHz. We synthesize designs using Cadence Genus [8], perform implementation using Cadence Innovus, run gate-level simulations using Synopsys VCS [40], and characterize power consumption based on transient behavior using Cadence Voltus.

D. Energy Efficiency on Accelerator

We integrate our dMAC units into a 128×128 output-stationary systolic array accelerator with a single 2MB level of local SRAM memory (128 16KB banks) and a 1GB external DRAM memory with 1KB page size (Figure 13) We use FN-CACTI [36] to estimate energy consumption of the SRAM and DRAM memories at the 7nm node. All MAC units are fully pipelined with 1 MAC/cycle throughput. Table II summarizes the area and power breakdown of the various components and possible MAC units.

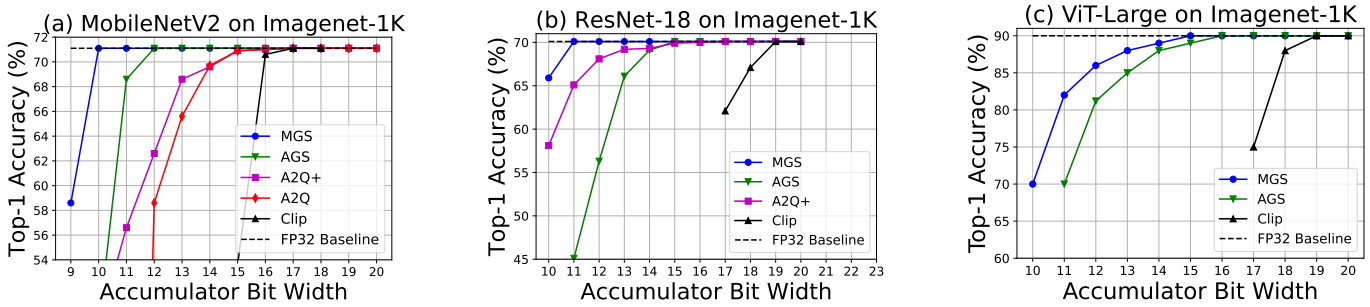

Fig. 11: Comparing INT8 MGS to SOTA methods for low-precision accumulation during quantized inference using several models. We sweep weight and activation bitwidths from 5 to 8 bits while varying the accumulator from 8 to 20 bits. We then plot the best-performing models with the lowest required accumulator bitwidth. Since MGS uses both narrow and wide accumulators during the dot product, we plot the average accumulator bitwidth when running MGS. In principle, MGS can indefinitely reduce the narrow accumulator bitwidth as it always falls back on the wide accumulator. However, we stop reducing accumulator bitwidth for MGS when additional reduction increases the average bitwidth (using the wide accumulator more often) and instead start clipping those overflows. By using narrow accumulators for the majority of sums, MGS can reduce accumulator bitwidth beyond the SOTA.

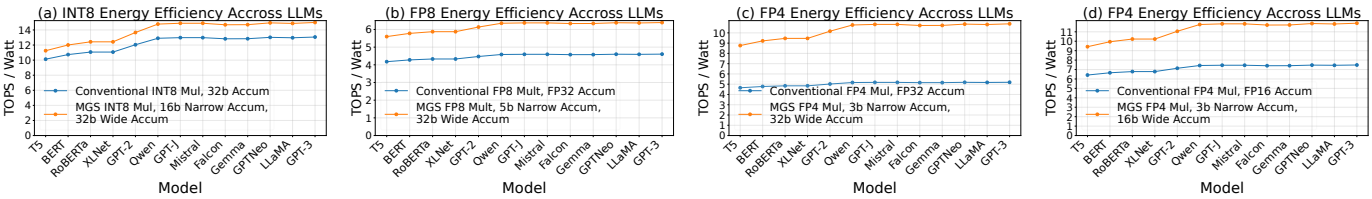

Fig. 12: Energy Efficiency (TOPs/Watt) when running various LLMs on our systolic array accelerator. Models are sorted by their embedding dimension (k).

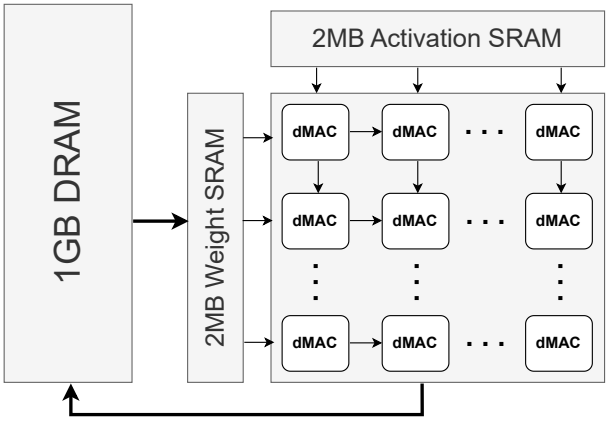

Fig. 13: 128×128 Output-stationary systolic array accelerator with integrated dMAC units.

We compare dMACs to conventional MACs when running inference computations from the MLP layer of several LLMs on GSM8K [10], Hellaswag [46], MMLU [23], and ANLI [45] datasets. We measure the energy Efficiency (TOPs/Watt) when running various LLMs on our systolic array accelerator.

Component	Area (um^2)	Power (W)
INT8-MAC	1336	0.424
FP8 mul, FP32 Accum	4262	1.558
FP4 mul, FP32 Accum	5774	1.398
FP4 mul, FP16 Accum	3671	0.914
INT8-dMAC	1510	0.394
FP8 dMAC, 32b Wide Accum	4207	1.068
FP4 dMAC, 32b Wide Accum	3927	0.572
FP4 dMAC, 16b Wide Accum	2841	0.507
SRAM	-	0.102
DRAM	-	0.083

TABLE II: Power consumption when using different MAC units in the systolic array

When implemented with dMAC units, the accelerator has up to $1.6\times$ higher energy efficiency compared to conventional MACs across data types.

E. Worst-Case Hardware Costs

We clarify (1) when MGS offers little or no benefit and (2) the worst-case hardware costs of dMAC. MGS requires the wide accumulator to be clock gated on the overflow flag for energy reduction benefits. In addition to the cases mentioned in the paper, we provide results below for: (a) "always-on" fallback logic where there is no gating of the

METRIC	INT8 MAC	INT8 dMAC	INT8 dMAC (always_on_fallback)
narrow accum bitwidth	32	16	16
critical path (ps)	1047	953	1028
critical path overhead	–	-9%	-2%
logic cell area (um^2)	1336	1510	1619
logic cell count overhead	–	13%	21%
chip area (um^2)	2114	2372	2532
chip area overhead	–	12%	20%
total power mobilenetv2 (uW)	25.9	24.1	33.4
total power adversarial (uW)	28.4	24.9	33.4

TABLE III: Worst-case results for INT8-dMAC with 16b narrow and 32b wide accumulators

METRIC	FP8-MAC	FP8-dMAC	FP8-dMAC (always_on_fallback)
mantissa bitwidth	32	5	5
critical path (ps)	1321	635	1260
critical path overhead	–	-52%	-5%
logic cell area (um^2)	4262	4207	4350
logic cell count overhead	–	-1%	2%
chip area (um^2)	6438	6363	6570
chip area overhead	–	-1%	2%
total power mobilenetv2_(uW)	95.1	65.2	83.3
total power adversarial_(uW)	95.3	68.9	85.5

TABLE IV: Worst-case results for FP8-dMAC with 5b narrow and 32b wide mantissa accumulators

wide accumulator, and (b) adversarial inputs with frequently toggling bits and overflows. Tables III and IV provide results for power/performance/area of the maximum and minimum narrow accumulator bitwidths for INT-dMAC and FP8-dMACs. In summary, although dMACs incur logic and area overheads, employing clock-gating of the expensive wide accumulator minimizes switching activity and reduces overall power consumption, even when adversarial inputs cause significant switching and overflow.

We analyze dMAC's critical-path delay where the delay is defined as the maximum combinational logic delay from the output of one register to the input of another register. MGS uses two alternative data-paths following the proposed gating: (1) narrow accumulation and (2) dMAC's "reduced" 32b-accumulation. The critical path determines the maximum clock frequency of the designs. Since dMAC splits multiply and accumulate operations in different stages, the critical path is shorter than in conventional MACs as there is less logic between flip-flops. For the "always-on" dMACs without clock-gating, latency is similar to that of conventional-MACs, while also achieving power-saving benefits. The dMAC unit cycle times are $<$ 2ns, and we may clock our designs at 500MHz.

For power, we compared dMACs to conventional-MACs when running inputs that maximize bit toggles (labeled adversarial). For example, FP8 adversarial weights/activations cause partial-products to utilize and switch all 16 exponent registers while incurring frequent mantissa overflow. As expected (last row of tables), using an adversarial input increases power consumption compared to mobilenetv2 for all designs due to increased switching activity. Even under such adversarial inputs, both int and float DMACs use less power than conventional-MACs running the same workload. This is because we clock-gate the dMAC wide accumulator based on the overflow flag. When we remove clock-gating, for a worst case "always-on" fallback logic running adversarial inputs, INT8-dMAC consumes 14% to 17% more power than conventional-MACs while FP8-dMAC consumes 10% less power than conventional-MAC at narrow bitwidth by avoiding FP8 to FP32 conversion and normalization (Figure 9).

F. Estimating the Expected Overflow Rate

For a target accumulator bitwidth, transition probabilities associated with each matrix-multiplication in a pre-trained DNN can be efficiently estimated using a single inference pass over the training and/or test dataset. Given the fixed dataset and fixed weights of the pre-trained DNN, we can obtain partial-product statistics via a single inference pass with the dataset. For every matmul, we record the empirical frequency of each distinct quantized partial-product. These frequencies are then used to construct the state transition matrix P, from which the expected summation length is derived via the fundamental matrix $N = (I - Q)^{-1}$ (Analysis Section). This estimation procedure introduces minimal computational overhead relative to that of quantization-aware-training or post-training-quantization techniques for FP8 or integer-quantized networks, which typically require multiple additional epochs of training. Moreover, despite variability in the input data, we observe that the distribution of partial-products remains highly consistent across inputs within a given matrix-multiplication layer due to normalization operations such as batchnorm. Hence, the construction of P doesn't require full dataset coverage and can be further accelerated by sampling a fraction of training and/or test data without significant loss in estimation quality.

G. Applying MGS to New Hardware and Applications

MGS's methodology can be applied to any application involving summation of integer or floating-point numbers. There is no loss of accuracy when using MGS as the dMAC defaults to using a wide, high-precision accumulator upon overflow. Consider, for example, dot-products for two applications in bfloat16 precision, (1) matched filtering (1D-convolution) for radar chirp signals [26] and (2) solving a linear ODE via the Forward Euler method. The ODE is an energy transport master equation $y'(t) = Ay(t)$ modeling thermal energy flow inside a protein [18] where A is a 500×500 dense matrix representing interactions (rate constants) between 500 heavy atoms in hemoglobin and $y(t)$ is kinetic energy of each atom at time t. In both applications, the 7-bit mantissas of bfloat16 partial-product values have a roughly uniform distribution. Based on our Markov analysis and empirical measurements of dot-products in these workloads, MGS can use a narrow accumulator for 5 out of every 6 sums. We expect MGS to yield power savings for several non-ML applications.

VII. Conclusion

This paper introduced MGS to reduce the required accumulation bitwidth in performing dot products that form the bulk of DNN computations. Based on the statistical properties

of weight and activation distributions, we can sum many partial products in reduced precision before overflow occurs. Specifically, MGS uses a narrow accumulator to accumulate as many values as possible while falling back on a wider accumulator when the rare overflow occurs. We have designed dual-multiply-accumulate (dMAC) hardware units that use narrow accumulators for most sums, resulting in a narrower average accumulator bitwidth compared to prior works. Using MGS, we can compute dot products with significantly reduced energy usage without accuracy loss and without retraining the model. Since dot products are one of the most frequently performed operations in DNNs, the MGS approach proposed in this paper is expected to be fundamental to efficient DNN computations.

REFERENCES

[1] Habana gaudi2 white paper, 2022.

[2] Nvidia h100 tensor core gpu architecture white paper, 2022.

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

[4] Nvidia transformer engine, 2025.

[5] Arm Limited. *ARM CMSIS Library*, January 2022.

[6] Arm Limited. *Cortex-M4 Technical Reference Manual*, January 2022.

[7] Arm Limited. *Arm Neon technology, the Advanced SIMD (Single Instruction Multiple Data) architecture extension for implementation of the Armv8-A or Armv8-R architecture profiles*, June 2024.

[8] Cadence Design Systems. Cadence innovus implementation system, 2024.

[9] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[10] Karl Cobbe, Vineet Kosaraju, Mo Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168, 2021.

[11] Ian Colbert, Alessandro Pappalardo, and Jakoba Petri-Koenig. A2q: Accumulator-aware quantization with guaranteed overflow avoidance. *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 16943–16952, 2023.

[12] Ian Colbert, Alessandro Pappalardo, Jakoba Petri-Koenig, and Yaman Umuroglu. A2q+: improving accumulator-aware weight quantization. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[13] Barry de Bruin, Zoran Zivkovic, and Henk Corporaal. Quantization of deep neural networks for accumulator-constrained processors. *Microprocess. Microsystems*, 72, 2020.

[14] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[16] L. C. W. Dixon and D. J. Mills. Effect of rounding errors on the variable metric method. *J. Optim. Theory Appl.*, 80(1):175–179, January 1994.

[17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *ArXiv*, abs/2010.11929, 2020.

[18] Erhan Deniz et al. Through bonds or contacts? mapping protein vibrational energy transfer using non-canonical amino acids. In *Nature Communications*, 2021.

[19] PyTorch Foundation. Pytorch quantization. https://pytorch.org/docs/stable/quantization.html.

[20] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca 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*, 378(2164):20190155, December 2019.

[21] Jorge Gomez, Saavan Patel, Syed Shakib Sarwar, Ziyun Li, Raffaele Capoccia, Zhao Wang, Reid Pinkham, Andrew Berkovich, Tsung-Hsun Tsai, Barbara 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.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[23] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021.

[24] Nicholas J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993.

[25] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[26] Zi Huang, Akila Pemasiri, Simon Denman, Clinton Fookes, and Terrence Martin. Multi-task learning for radar signal characterisation. In *Proceedings of the 2023 IEEE International Conference on Acoustics, Speech, and Signal Processing Workshops (ICASSPW)*, pages 1–5, 2023.

[27] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry 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.

[28] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, January 1965.

[29] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. Term quantization: furthering quantization at run time. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[30] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. MCUNet: Tiny deep learning on IoT devices. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, pages 11711–11722, Red Hook, NY, USA, December 2020. Curran Associates Inc.

[31] Eran Malach, Gilad Yehudai, Shai Shalev-Shwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. *ArXiv*, abs/2002.00585, 2020.

[32] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022.

[33] Vikas Natesh and H. T. Kung. Alternating greedy schedules: Enabling low-bitwidth accumulation of dot products in neural network computations. In *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2025.

[34] Renkun Ni, Hong-Min Chu, Oscar Castañeda, Ping-yeh Chiang, Christoph Studer, and Tom 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.

[35] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, Ruihang Li, Miaosen Zhang, Chen Li, Jia Ning, Ruizhe Wang, Zheng Zhang, Shuguang Liu, Joe Chau, Han Hu, and Peng Cheng. Fp8-lm: Training fp8 large language models, 2023.

[36] Divya Praneetha Ravipati, Rajesh Kedia, Victor M. Van Santen, Jörg Henkel, Preeti Ranjan Panda, and Hussam Amrouch. Fn-cacti: Advanced cacti for finfet and nc-finfet technologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(3):339–352, 2022.

[37] Andrew Sabot, Vikas Natesh, H. T. Kung, and Wei-Te Ting. MEMA runtime framework: Minimizing external memory accesses for tinyml on microcontrollers. *TinyML Research Symposium 2023*, abs/2304.05544, 2023.

[38] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4510–4520. Computer Vision Foundation / IEEE Computer Society, 2018.

[39] Moritz Scherer, Manuel Eggimann, Alfio Di Mauro, Arpan Suravi Prasad, Francesco Conti, Davide Rossi, Jorge Tomás Gómez, Ziyun Li, Syed Shakib Sarwar, Zhao Wang, Barbara De Salvo, and Luca 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)*, pages 217–220, 2023.
[40] Synopsys. Synopsys vcs simulation solution, 2021.
[41] Tensorflow. Tensorflow lite quantization.
[42] Cuong Tran, Ferdinando Fioretto, Jung-Eun Kim, and Rakshit Naidu. Pruning has a disparate impact on model accuracy. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.
[43] Shibo Wang and Pankaj Kanwar. Bfloat16: The secret to high performance on cloud tpus. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, 2019. Accessed: 2025-02-02.
[44] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. January 1964.
[45] Adina Williams, Tristan Thrush, and Douwe Kiela. Anlizing the adversarial natural language inference dataset, 2020.
[46] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.