TSpace Research Repository utoronto.scholaris.ca

Fusing Adds and Shifts for Efficient Dot Products

Pavel Golikov, Karthik Ganesan, Gennady Pekhimenko, and Mark C. Jeffrey

Version Post-print/Accepted Manuscript

Citation (Published Version) P. Golikov, K. Ganesan, G. Pekhimenko and M. C. Jeffrey, "Fusing Adds and Shifts for Efficient Dot Products," in IEEE Computer Architecture Letters, https://doi.org/10.1109/LCA.2025.3637718.

DOI 10.1109/LCA.2025.3637718

Publisher's Statement © 2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

How to cite TSpace items

Always cite the published version, so the author(s) will receive recognition through services that track citation counts, e.g. Scopus. If you need to cite the page number of the **author manuscript from TSpace** because you cannot access the published version, then cite the TSpace version **in addition to** the published version using the permanent URI (handle) found on the record page.

This article was made openly accessible by U of T Faculty.
Please tell us how this access benefits you. Your story matters.

UNIVERSITY OF TORONTO LIBRARIES

Fusing Adds and Shifts for Efficient Dot Products

Pavel Golikov, Karthik Ganesan, Gennady Pekhimenko, and Mark C. Jeffrey

***Abstract*—Dot products are heavily used in applications like graphics, signal processing, navigation, and artificial intelligence (AI). These AI models in particular impose significant computational demands on modern computers. Current accelerators typically implement dot product hardware as a row of multipliers followed by a tree of adders. However, treating multiplication and summation operations separately leads to sub-optimal hardware. In contrast, we obtain significant area savings by considering the dot product operation as a whole. We propose `FASED`, which fuses components of a Booth multiplier with the adder tree to eliminate a significant portion of full adders from a baseline INT8×INT8,4,2 design. Compared to popular dot product hardware units, `FASED` reduces area by up to $1.9\times$.**

***Index Terms*—AI accelerators, digital arithmetic, vectors.**

I. Introduction

THE *dot product* is the workhorse of tensor-algebra-based AI, including computer vision, recommenders, and large language models (LLMs). This core operation replicates across batches of inputs and across model layers to form large matrix-matrix multiplications, demanding ever increasing throughput from hardware. Dominating model inference time, dot products operate on vectors of input activations ($\mathbf{a} = [\mathbf{a_1} \dots \mathbf{a_n}]$) and the model's trainable weights ($\mathbf{w} = [\mathbf{w_1} \dots \mathbf{w_n}]$): [1]

$$\mathbf{a} \cdot \mathbf{w} = \sum_{i=1}^{n} \mathbf{a_i} \times \mathbf{w_i}. \tag{1}$$

To better utilize constrained resources like area and off-chip bandwidth, *quantization* reduces the bit precision of weights and activations. Quantization improves performance by allowing more dot product computations per area. It increases effective memory capacity and bandwidth by storing and moving more (smaller) units of data per byte. Despite losing precision, quantized models have retained high quality across architectures such as LLMs, vision transformers, convolutional neural networks, and recommendation systems [1], [2]. Quantized dot products are supported in accelerators from NVIDIA [3], Google [4], and Samsung [5], among others.

Many models also exploit *variable bit-width quantization*: squeezing efficiency with 2-bit weights for some layers, while relying on larger (e.g., 4- or 8-bit) weights when required [2], [6]. Specialized variable-width hardware remains important [7], [8] as the alternative is over-provisioned hardware (e.g., 8-bit × 8-bit or $8b{\times}8b$) that simply sign extends low-bit values and misses opportunities for high throughput for lower bitwidths.

Received 14 August 2025; revised 16 October 2025; accepted 19 November 2025. All authors are with the University of Toronto, Canada. Gennady Pekhimenko is also with NVIDIA. (e-mail: golikovp@cs.toronto.edu). Our code is available at github.com/mcj-group/fased-verilog.

[1]Bolded signals represent multi-bit vectors whereas non-bolded signals represent individual bits, e.g., $\mathbf{a_i}$ and $\mathbf{w_i}$ are multi-bit values in Eq. 1.

Conventional wisdom views multiplication separately from accumulation [5], [7], [9]. A common multiply-accumulate architecture is a row of multipliers followed by an adder tree (Fig. 1) and a final accumulator. However, this misses a key opportunity at the intersection of dot products, quantization, and binary arithmetic [10]: *viewing these operations separately leads to wasted area.*

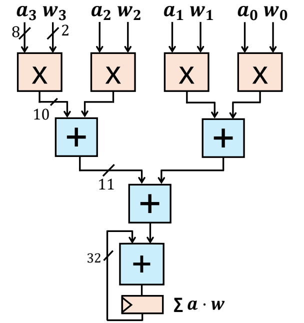


Fig. 1. Dot product hardware.

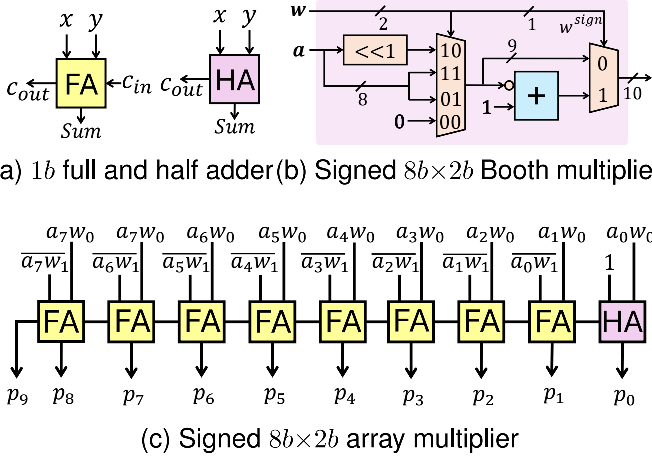
We propose `FASED`, hardware for performing integer dot products that considers multiplication and accumulation holistically. We observe that one area-hungry operation performed during multiplication can be combined with the accumulation step to yield significant area savings. We propose two designs: *(i)* `FASED-FW` performs dot products of $8b{\times}2b$ (Sec. III-A) and *(ii)* `FASED-VW` further supports $8b{\times}4b$ and $8b{\times}8b$ computations (Sec. III-B). We evaluate `FASED` against variable-width dot product hardware units based on the popular Bit Fusion [8] AI accelerator. `FASED` reduces area by $1.41\times$ to $1.91\times$ over the baseline with a $< 9\%$ decrease in F_{max}.

II. Background and Motivation

Fig. 1 shows the hardware unit for a 4-input dot product. The inputs to the *multipliers* are 2-bit model weights ($\mathbf{w}$) and 8-bit activations ($\mathbf{a}$). Each cycle, the unit multiplies four pairs of values and sums them using the *adder* tree to produce the final result. Throughout this paper, we label 1-bit full adders (Fig. 2a) as "FA" and use the term "adder" for a chain of m 1-bit full adders, which is used to add two m-bit numbers.

Array multiplication: Fig. 2c shows the design of the commonly used signed *array multiplier* [11], for multiplying 8-bit and 2-bit values ($8b{\times}2b$). It operates in two stages: *(i)* generation then *(ii)* accumulation of partial products. The figure labels each partial product as $a_i w_k$, indicating the bitwise AND of a_i and w_k. The partial product bits are summed to produce the final 10-bit product ($p_9 \dots p_0$). Fig. 2c shows a single row of adders as we focus on $8b{\times}2b$ multiplication.

Booth multiplication: Booth multiplication generates the partial product for *multiple* bits of the multiplicand per cycle and accumulates them to obtain the final result [10], [11]. Fig. 2b shows a signed $8b{\times}2b$ *radix-4 Booth multiplier*. A radix-2^m Booth multiplier processes m multiplicand bits per cycle, so Fig. 2b takes just one cycle for the 2-bit $\mathbf{w}$. The 2-bit signed $\mathbf{w} \in \{-2, -1, 0, 1\}$ can have three possible magnitudes in the first stage: 0, 1, or 2. The left mux of Fig. 2b uses $|\mathbf{w}|$ to select 0, $\mathbf{a}$, or $2\mathbf{a}$ ($= \mathbf{a} << 1$), respectively. This so-called



(a) $1b$ full and half adder (b) Signed $8b\times2b$ Booth multiplier

(c) Signed $8b\times2b$ array multiplier

Fig. 2. Circuits used frequently in our work. 'FA' and 'HA' indicate 1-bit full and half adders respectively. The blue '+' box is $m-1$ full adders and a single half adder, similar to (c). (b) and (c) output $\mathbf{a}\times\mathbf{w}$.

Booth recoding table outputs the partial product $|\mathbf{w}|\times\mathbf{a}$. The right mux of Fig. 2b applies the sign of $\mathbf{w}$ by conditionally negating $|\mathbf{w}|\times\mathbf{a}$, using w^{sign} as the select. Negating a number in two's complement requires *(i)* bit-wise *inverting* all the bits and *(ii) incrementing* the inverted value by 1. Consequently, the Booth multiplier requires an adder and an inverter.

Our work is motivated by two key observations. First, n array multipliers and n Booth multipliers use the same number of 1-bit full adders, but in the Booth multiplier, *the full adders simply increment by 1*. Second, the $n-1$ adders in a dot product adder tree can be easily modified to accept a carry-in to cheaply enable increment by 1. Our technique leverages these observations to save area in quantized dot product hardware.

III. FASED Design

`FASED` is a signed-integer dot product hardware unit that eliminates many of the adders from a Booth-inspired design by performing increments through the carry-in signals in the adder. `FASED` provides native support for variable bit-width dot products to exploit the success of quantized models that use 2-, 4-, or 8-bit integer weights [6], [12], [13]. To maximize quantized performance per area, the hardware unit performs $8b\times2b$ dot products at twice the rate as $8b\times4b$ dot products, which in turn is twice as fast as $8b\times8b$ dot products. The alternative, supported by some prior work, only targets the highest precision (i.e., $8b\times8b$) and therefore sacrifices performance for $8b\times2b$ and $8b\times4b$.

We describe `FASED` in two stages. First, `FASED-FW` only supports fixed-width weights of 2 bits (Sec. III-A). `FASED-FW` builds on the baseline shown in Fig. 1, using $8b\times2b$ radix-4 Booth multipliers (Fig. 3). `FASED-FW` removes adders needed for partial product negation $(+1)$ by absorbing the increments into the adder tree. Second, `FASED-VW` builds on `FASED-FW` to support 4- and 8-bit weights with conditional multiplicand shifting and modifying the magnitude-select signals.

A. Fixed-width FASED

Fig. 4 shows our `FASED-FW` design for a 4-input dot product. `FASED-FW` exploits that, for a n-input dot product (n is a power of 2), each of the $n-1$ adders in the adder tree can be modified to accept a carry-in. We do so by switching the single half adder with a full adder.

We use these carry-in ports to support the $+1$ for conditional negation of partial products. However, to support the case that all n input weights are negative—and requiring n such increments—we use the c_{in} of the accumulator at the bottom of the adder tree. With adders in the Booth multipliers removed, a simple and functional `FASED-FW` would leave the inverters and muxes above the adder tree within each purple region. Instead, our `FASED-FW` in Fig. 4 conditionally negates only *one* of the inputs to each adder, moving two inverters and muxes into the adder tree. This is a step toward `FASED-VW`.

Only the right input to each adder is conditionally negated, but `FASED-FW` supports cases where the left input has a negative weight through term rewriting:

$$l+r=+(l+r) \quad (2) \qquad -l+r=-(l-r) \quad (4)$$

$$l-r=+(l-r) \quad (3) \qquad -l-r=-(l+r) \quad (5)$$

In every rewritten equation, the outer sign matches the original sign of the left input, and is processed in the next adder level in Fig. 4. In the inner term, each rewritten equation leaves the left adder input unchanged, allowing only the right input to be conditionally negated. Specifically, if the left and right inputs have matching signs (Eq. 2 and Eq. 5) neither input is negated in the inner rewritten term. `FASED-FW` achieves this by using the XOR of weight signs as the mux select and adder carry-in (green background in Fig. 4). The process repeats in the next adder level, using the ignored outer signs of the first level (w_3^{sign} and w_1^{sign}) as the inner signs for this level. Eventually this process reaches the accumulator value and the output of the adder tree is either added to or subtracted from the accumulator based on the sign of the most significant weight (w_3^{sign}). Fig. 5 shows a concrete example of `FASED-FW`. Each step labeled ① to ⑤ in Fig. 5 is similarly labeled in Fig. 4.

B. Variable-width FASED

`FASED-VW` supports multiple bit widths using different *modes* of operation, matching prior work [7], [8]. In 2-bit mode, the four weights and activations are the same as in Sec. III-A. In 4-bit mode, `FASED-VW` forms two 4-bit weights by concatenating $[\mathbf{w_3w_2}]$ and $[\mathbf{w_1w_0}]$, while activations $\mathbf{a_3}$ and $\mathbf{a_2}$ are the same, as are $\mathbf{a_1}$ and $\mathbf{a_0}$. In 8-bit mode, all four 2-bit weight signals concatenate into one 8-bit weight and all activation signals are equal.

Hardware changes: Extending `FASED-FW` to support variable-width weights requires two changes to hardware: *(i)* modifying the Booth recoding table (left mux in Fig. 2b) to mimic a multi-cycle Booth multiplier [10], and *(ii)* adding conditional shift operations. Fig. 6 shows the modified Booth recoding table (BRT) for each input pair, with one BRT shown in detail. Like `FASED-FW`, the BRT output remains one of 0, $\mathbf{a}$, or $2\mathbf{a}$. Unlike `FASED-FW`, partial product selection uses 3-bit *Booth groups*, formed by padding the 2-bit weight segments.

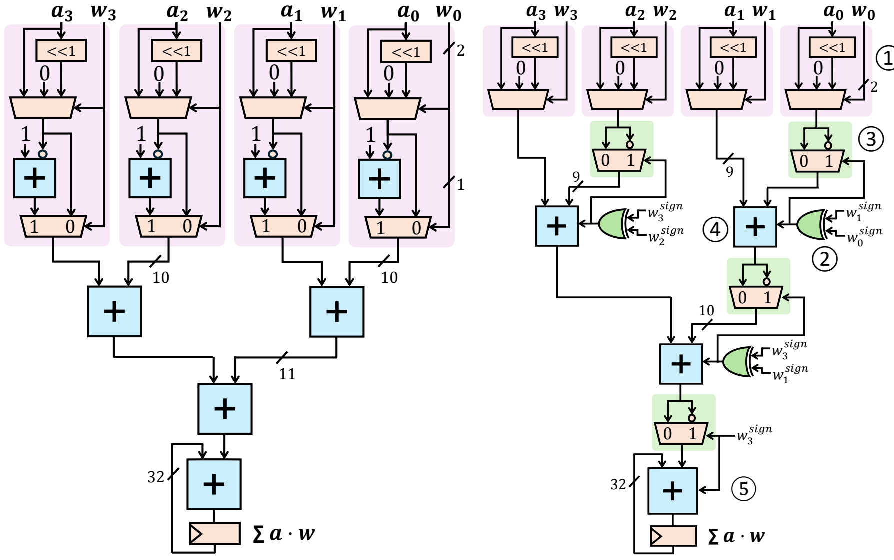


Fig. 3. $8b \times 2b$ baseline with radix-4 Booth multipliers Fig. 4. FASED-FW

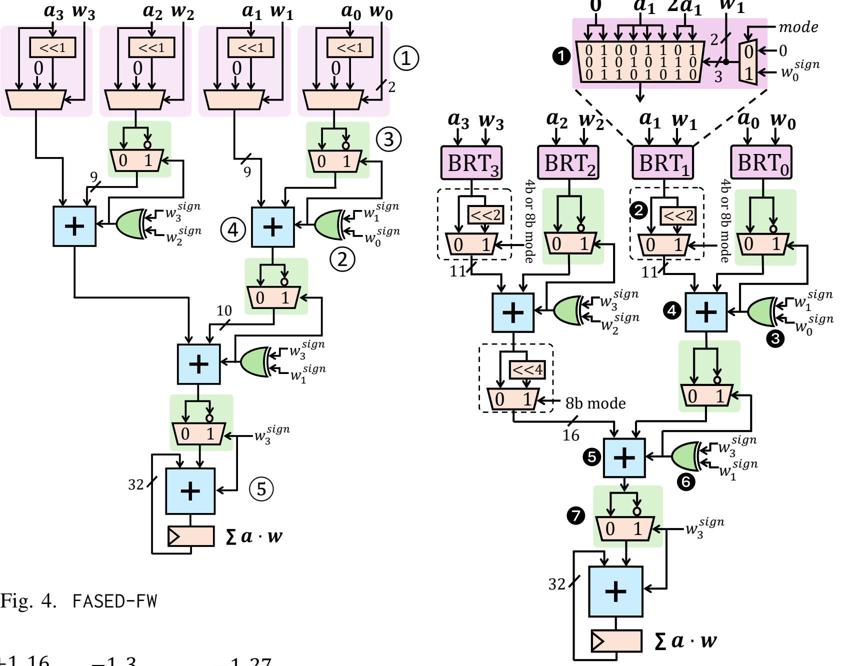


Fig. 6. FASED-VW

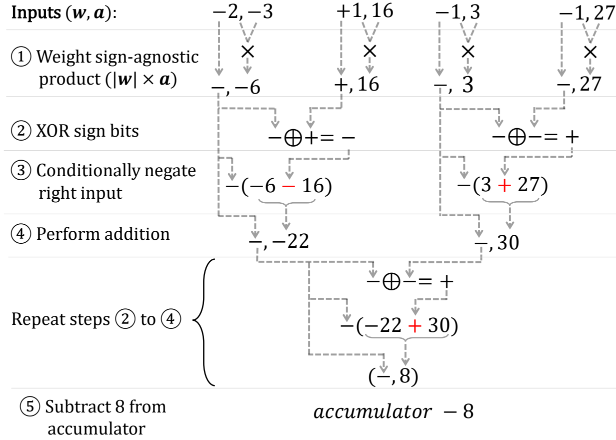


Fig. 5. Example of FASED-FW operation

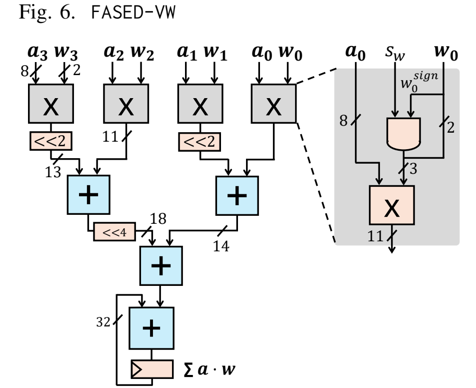


Fig. 7. Bit Fusion architecture baseline

Based on the mode, FASED-VW pads each 2-bit segment $\mathbf{w_i}$ with either 0 or the most significant bit of $\mathbf{w_{i-1}}$ to create 3-bit Booth groups. Based on the sign of a Booth group, FASED-VW then adds or subtracts the corresponding adder input. An advantage of our design is that the XOR of the signs of both Booth groups still produces the correct result in every mode. FASED-VW also introduces conditional shift operations of each adder's *left* input for the first two stages of the adder tree, mirroring FASED-FW's conditional negation of each adder's *right* input. This aligns partial products by shifting values by 2 or 4 bits to support 4- and 8-bit modes, respectively.

Example: Consider the dot product of $\mathbf{a} \cdot \mathbf{w} = [0 \;\; 3] \cdot [0 \;\; -5]^T$ on FASED-VW using 4-bit weights. We detail the steps with ❶ to ❼ and label the corresponding hardware in Fig. 6. The right subtree (BRT_1 and BRT_0) computes 3×-5. As $-5 = 1011_2$, $\mathbf{w_1} = 10_2$ and $\mathbf{w_0} = 11_2$. The two Booth groups are $10\underline{1}$ and $11\underline{0}$ (padded bits are underlined) and these index into the BRT muxes (❶ in Fig. 6). Thus, the outputs of both BRT_1 and BRT_0 are 3 ($= \mathbf{a_1} = \mathbf{a_0}$).

In 4-bit mode, FASED-VW shifts the output of BRT_1 by 2 bits (❷): $000000011_2 << 2 = 000001100_2 = 12$. Next, FASED-VW XORs w_1^{sign} and w_0^{sign} (❸), finding same-signed partial products (Eq. 5): $1 \bigoplus 1 = 0$. The carry-in to adder ❹ is thus 0 and the output of BRT_0 is *not* inverted. The output of adder ❹ is $12 + 3 = 15$. The output of the corresponding left adder is 0 because $\mathbf{a_3} = \mathbf{a_2} = \mathbf{w_3} = \mathbf{w_2} = 0$.

XOR ❻ computes $0 \bigoplus 1 = 1$, indicating that FASED-VW must subtract the right input from the left input, i.e., invert the right input of adder ❺ and set its carry-in to 1. This results in $0 - 15 = -15$. Finally, ❼ selects the non-negated output of adder ❺ and sets the carry-in to the accumulator as w_3^{sign}=0 and its right input as -15.

IV. Evaluation

We compare the area of FASED against the fixed- and variable-width dot product units of DNN inference accelerators, as the dot product unit is the core component of both 1D and 2D architectures. For example, FASED can be integrated into a weight-stationary or input-stationary systolic array as a drop-in replacement for a column of processing elements.

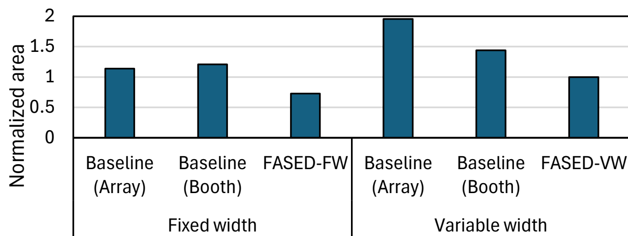


Fig. 8. Area of `FASED` and baselines, normalized to `FASED-VW`. Lower is better.

A. Methodology

We compare `FASED` to four baselines based on Bit Fusion [8]. Bit Fusion was originally proposed for CNN and recurrent models but has since been built upon in many subsequent works targeting CNNs [14], LLMs [7], and video analytics [15]. Bit Fusion supports large-width products via the sum of conditionally shifted small-width multipliers. Bit Fusion originally used $2b \times 2b$ multipliers as their basic building block. However, as this is inefficient for $8b \times 2b$ dot products, our baselines use $8b \times 2b$ multipliers, following the methodology of Bit Fusion. Our baselines use carry-propagate adders throughout all designs. We leave multi-operand adder optimizations such as carry-save adders to future work.

We construct four baselines by varying two design parameters: *(i)* support for *fixed-* or *variable*-width dot products, and *(ii)* use of *array* or *radix-4 Booth* multipliers. The fixed-width baselines serve as an ablation study. Our fixed-width Booth baseline (`base-bfw`) is identical to Fig. 3. Fig. 7 shows our variable-width array baseline (`base-avw`), which uses wider $8b \times 3b$ multipliers to correctly perform unsigned multiplication, matching Bit Fusion [8]. Thus, `base-avw` adds a row of 1-bit full adders in each multiplier to accumulate the third partial sum. Conditional shifters align partial products inside the adder tree, similar to `FASED-VW` and prior work [14], [15]. We implement all designs in Verilog and obtain area/delay with Yosys (v0.43) and OpenRoad/OpenSTA (v2.0-17598) [16].

B. Results

Fig. 8 shows the area of `FASED` and our baselines, separating fixed- and variable-width designs. On the fixed-width side, `FASED-FW` requires $1.52\times$ less area than `base-afw`. `FASED-FW` significantly decreases the area of `base-bfw` by eliminating many adders *because it treats the dot product holistically.* `FASED-FW` more than makes up the overhead of Booth to outperform `base-afw`.

On the variable-width side, `FASED-VW` requires $1.41\times$ less area than `base-bvw`and $1.91\times$ less area than `base-avw`. Interestingly, the switch from array to Booth *decreases* area by $1.36\times$, due to the overhead of the extra row of full adders in `base-avw`. Booth multipliers obviate this problem because they were designed to accumulate one partial product per cycle with only one row of 1-bit full adders—accumulating multiple partial products across different bit width modes comes naturally. `FASED-VW` area improvements come from its holistic treatment of multiplication and accumulation, on top of a more efficient variable-width multiplier design. `FASED` scales to larger vector sizes by increasing the number of multipliers and the size of the adder tree. Our experiments show that `FASED` exhibits similar scaling behavior to the baselines.

We also compared `FASED` against MANT [7], a recent LLM accelerator supporting $8b \times 2b$ as the basic operation, that uses the Bit Fusion approach to support 2-, 4- and 8-bit weights. The MANT dot product unit area far exceeds that of `FASED` or our baselines as it requires additional hardware to accumulate a second set of partial products. The novelty of MANT lies in a quantization technique that is orthogonal to our own; we exclude MANT from our results as it would be unfair to compare directly with `FASED`.

The hardware changes for `FASED` have only a minor impact on F_{max}. Compared to `base-afw`, `FASED-FW` sees a $< 9\%$ drop in F_{max}, while `FASED-VW` sees a $< 7\%$ drop compared to `base-avw`. We believe that pipelining can close this gap but we leave this exploration to future work.

V. Conclusion

Dot products are the essential operation underpinning many important AI models. We show that the typical hardware for computing signed integer dot products is sub-optimal. We propose `FASED`, which treats the multiplication and summation steps in dot products holistically. By modifying a radix-4 Booth multiplier, `FASED` significantly reduces area of dot product computation. Compared to a popular variable bit-width dot product unit, `FASED-VW` reduces area by up to $1.9\times$.

Acknowledgements

We thank reviewers for feedback. We gratefully acknowledge support from NSERC, Fujitsu, and the Vector Institute.

References

[1] K. Liu *et al.*, "Low-bit model quantization for deep neural networks: A survey," *arXiv preprint*, 2025.
[2] M. Li *et al.*, "Contemporary advances in neural network quantization: A survey," in *IJCNN*, 2024.
[3] D. Salvator *et al.*, "Int4 precision for AI inference," NVIDIA Developer Technical Blog, 2019.
[4] N. Jouppi *et al.*, "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *ISCA*, 2023.
[5] J.-W. Jang *et al.*, "Sparsity-aware and re-configurable NPU architecture for Samsung flagship mobile SoC," in *ISCA*, 2021.
[6] R. Gong *et al.*, "A survey of low-bit large language models: Basics, systems, and algorithms," *arXiv preprint*, 2024.
[7] W. Hu *et al.*, "M-ANT: Efficient low-bit group quantization for LLMs via mathematically adaptive numerical type," in *HPCA*, 2025.
[8] H. Sharma *et al.*, "Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ISCA*, 2018.
[9] T. Chen *et al.*, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.
[10] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, 1951.
[11] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2010.
[12] C. Zeng *et al.*, "ABQ-LLM: Arbitrary-bit quantized inference acceleration for large language models," in *AAAI*, 2025.
[13] G. Xiao *et al.*, "SmoothQuant: Accurate and efficient post-training quantization for large language models," in *ICML*, 2023.
[14] S.-F. Hsiao *et al.*, "Flexible multi-precision accelerator design for deep convolutional neural networks considering both data computation and communication," in *VLSI-DAT*, 2020.
[15] Y. Kim *et al.*, "DACAPO: Accelerating continuous learning in autonomous systems for video analytics," in *ISCA*, 2024.
[16] T. Ajayi *et al.*, "Toward an open-source digital flow: First learnings from the OpenROAD project," in *DAC*, 2019.