tuGEMM: Area-Power-Efficient Temporal Unary GEMM Architecture for Low-Precision Edge AI

Harideep Nair, Prabhu Vellaisamy, Albert Chen, Joseph Finn, Anna Li, Manav Trivedi, John Paul Shen
Electrical and Computer Engineering Department, Carnegie Mellon University

***Abstract*—General matrix multiplication (GEMM) is a ubiquitous computing kernel/algorithm for data processing in diverse applications, including artificial intelligence (AI) and deep learning (DL). Recent shift towards edge computing has inspired GEMM architectures based on unary computing, which are predominantly stochastic and rate-coded systems. This paper proposes a novel GEMM architecture based on temporal-coding, called *tuGEMM*, that performs *exact* computation. We introduce two variants of *tuGEMM*, *serial* and *parallel*, with distinct area/power-latency trade-offs. Post-synthesis Power-Performance-Area (PPA) in 45 nm CMOS are reported for 2-bit, 4-bit, and 8-bit computations. The designs illustrate significant advantages in area-power efficiency over state-of-the-art stochastic unary systems especially at low precisions, e.g. incurring just 0.03 mm^2 and 9 mW for 4 bits, and 0.01 mm^2 and 4 mW for 2 bits. This makes tuGEMM ideal for power constrained mobile and edge devices performing always-on real-time sensory processing.**

***Index Terms*—GEMM, unary computing, temporal coding**

I. Introduction and Background

General matrix multiplication (GEMM) performs multiply-and-accumulate operations on matrices and forms the fundamental building block for deep neural networks (DNNs). While the application performance of DNNs has increased steadily over the years, its computational demands have increased exponentially [19]. Traditionally, GEMM was implemented as software libraries for CPUs and GPUs [14], [15]. However, more recently, dedicated GEMM hardware units have been implemented within GPUs and DNN accelerators to improve compute efficiency. The increasing demand for hardware acceleration resulted in companies like Nvidia introducing the tensor cores [22] capable of performing 4x4 matrix multiplication, and Google introducing Tensor Processing Units (TPUs) [6] with Matrix Multiply Units (MXUs). Further, with the latest push towards edge computing and on-device AI [8], [17], focus has shifted towards developing low-footprint GEMM hardware. Towards that goal, Nvidia and Google introduced Jetson Xavier NX [4], and edge TPU [3] respectively, both of which deploy reduced compute on device, albeit at the expense of inference accuracy. In the current landscape, various such lightweight systems have been proposed [10], [16], including deep learning accelerators (DLAs) in modern smartphones [5]. These systems predominantly operate on binary values and trade off inference accuracy to meet the Power-Performance-Area (PPA) constraints for edge devices.

On the other hand, *unary* compute-based implementations offer a promising alternative solution to the increasing parallel computation complexity inherent to binary implementations, delivering low area and low power designs. This emerging paradigm replaces the multiple parallel bits with a single serial bit-stream. Unary computing manifests in two major forms, *rate* and *temporal* coding, with rate-based methods being more prevalent. Recently proposed *uGEMM* [21] is a unified rate-and-temporal encoded GEMM architecture that incorporates unary arithmetic units to perform stochastic GEMM operations. It provides significant PPA improvements compared to previous unary designs, while maintaining high accuracy, making it a promising candidate for edge devices. However, being a stochastic approach, it doesn't perform exact compute and falls under the widely researched domain of approximate computing. In contrast, our work focuses on exact, not approximate, GEMM compute based purely on temporal encoding.

Recent works have emphasized the current trend of AI/DL to move towards lower precision. Authors in [12] performed training with 5-bit weights and 4-bit activations, and in [11] with 4-bit weights and 8-bit activations, both with minimal accuracy degradation. More recently, IBM researchers achieved 8-bit precision for training and 4-bit precision for inference across many deep learning datasets [20], followed by the work in [18] that shows both training and inference can be performed with 4-bits with negligible impact on accuracy. Additionally, Akida NSoCs [2] employ 1, 2, 4-bit computations for their weights and activations targeting edge inference. This growing affinity towards low precision influences this work to explore low bit-width implementations for edge AI.

We propose a novel GEMM architecture, *tuGEMM*, based on exact temporal compute, targeting area-power efficiency for low precision edge AI. Our key contributions are as follows:

- We propose a novel temporal-coding-based GEMM architecture that performs exact computations in contrast to existing stochastic and rate-based approaches.
- Two architecture variations, *serial* and *parallel*, that offer different area-latency tradeoffs are introduced.
- Gate-level implementations for both designs are presented, and post-synthesis PPA numbers for 2-bit, 4-bit and 8-bit implementations are reported.
- Latency evaluation for tuGEMM compute is performed using a representative DNN workload, ResNet18.
- We illustrate superior area-power efficiency over state-of-the-art unary approach, especially for low precision.

Section II presents the encoding mechanism and the architecture of the two design variants. Section III evaluates tuGEMM against uGEMM based on latency and post-synthesis

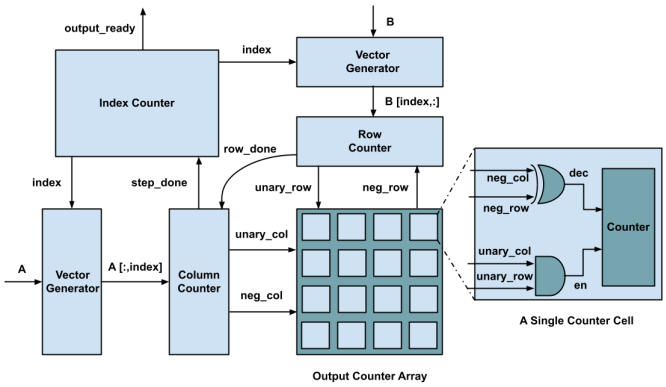

Fig. 1: Serial tuGEMM architecture for 4x4 GEMM compute

area and power. Conclusions and future directions are in Section IV.

II. TUGEMM ARCHITECTURE

General matrix multiplication (GEMM) is central to the compute-intensive operations in DNNs. Its general format is:

$$\mathbf{Y} = \alpha\mathbf{AB} + \beta\mathbf{C} \tag{1}$$

where **A**, **B** and **C** are generic MxN, NxP and MxP input matrices, **Y** is the MxP output matrix, and α and β are scaling factors. This work focuses on *non-scaled* GEMM operation, i.e., α = β = 1. This section describes the input encoding and the micro-architecture of the proposed *tuGEMM* hardware.

A. Input Encoding

The key idea of unary hardware implementation is encoding values as serial bitstreams on a single bitline. Such an input encoding allows the hardware to be repurposed with significantly less area and power. Unary encoding can be accomplished in two ways: rate and temporal coding. Rate-based systems encode values in the frequency of ones randomly distributed across the bitstream, whereas temporal coding encodes values in the time duration for which a signal is asserted. As a result, a temporally encoded bitstream consists of consecutive ones followed by consecutive zeros, resulting in only two transitions. This naturally leads to improved dynamic power consumption, compared to rate coding with multiple signal transitions due to the distributed occurrence of ones.

Rate coding typically implements stochastically-generated bitstreams using expensive random number generators (RNGs) and suffers from the correlation problem, requiring additional hardware to mitigate it [1], [7], [9]. In contrast, temporal encoding uses a single contiguous n-cycle wide logic pulse to represent a value n, analogous to the spike encoding employed in neuromorphic computing [13], and can enable exact deterministic compute in an efficient manner as it does not require RNGs. Our proposed approach distinguishes from previous works by utilizing temporal-unary-encoded exact compute.

B. Serial Architecture

The serial architecture (Fig. 1) consists of an MxP array of *output counter cells* surrounded by peripheral logic that performs unary encoding and co-ordinates the dataflow into the array. The counter array receives unary-encoded input matrices **A**, **B** from the left and top respectively, and implements multiplication in unary fashion. The multiplication compute occurs in N *steps*, where N is the common matrix dimension, which is equal to the number of columns in **A** and number of rows in **B**. Each *step* computes the outer product of i^{th} column from **A** and i^{th} row from **B**. During each *column-row* outer product, the MxP output counters update their counts with the MxP output values, taking as many cycles as the magnitude of the maximum output value (due to unary multiplication which will be described shortly). Thus, these outer products are accumulated over the N *steps*, at the end of which the final counter values reflect the output matrix **Y**. To eliminate a separate adder, the MxP counters are initialized with the binary-encoded input matrix **C** (following sections focus only on **A x B** multiplication). The serial architecture performs the N *steps* serially, and is named so. It has four components:

1) index counter: Each column of **A** is indexed simultaneously with the corresponding row of **B**. This indexing is generated by the *index counter* that counts up from 0 to $N-1$, incrementing each time by one after every *step*, indicated by *step_done* signal. Once its count reaches N, the index counter asserts an *output_ready* signal, implying GEMM has finished.

2) vector generator: Two vector generators receive index i from the *index counter* and use it to index into the input matrices **A** and **B** to generate the i^{th} column of **A** (M-dimensional vector), and i^{th} row of **B** (P-dimensional vector).

3) column/row counters: M *column counters* and P *row counters* convert the binary values from the vector generator to unary signals and co-ordinates the unary multiplication. In every *step*, the column and row values from the *vector generator* are loaded into the counters, which then begin counting towards zero (decrement if the initialized count is positive, increment if negative). The counters operate in a nested fashion such that the row counters are updated by 1 every cycle whereas the column counters only update their values (by 1) once all row counters reach zero. This cycle repeats until all the column counters reach zero thus completing one *step*, eventually triggering the *index counter* for the next *step* via *step_done* signal. During every *step*, the column and row counters assert M *unary_col* and P *unary_row* signals respectively, whenever their corresponding counts are non-zero. These signals represent the converted unary signals derived from the vector generator values, and enable column-row outer product in the *output array* as will be described next. Each counter also asserts a *neg_col/row* signal, if the corresponding initialized count is negative, to determine the direction of update in the output counter cells.

4) output counter array: It consists of MxP counter cells (initialized with matrix **C**), where each counter accumulates the unary *column-row* outer product within a *step*, and is enabled when *unary_col/row* are both asserted. If enabled, it

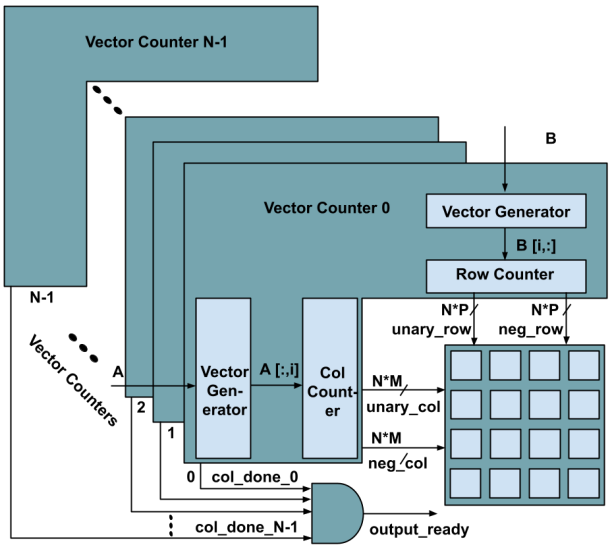

Fig. 2: Parallel tuGEMM architecture for 4x4 GEMM compute

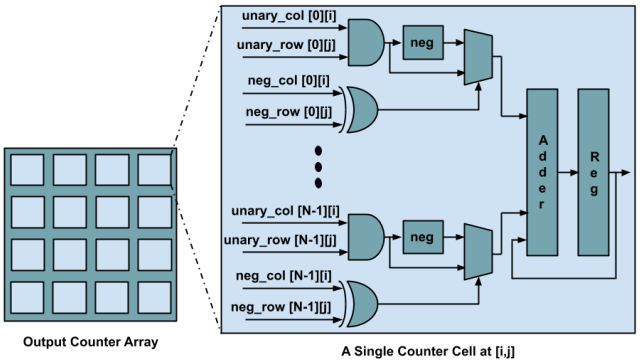

Fig. 3: A single output adder cell in parallel tuGEMM

increments every cycle if the inputs have the same sign, else it decrements. When the *index counter* asserts *output_ready*, the output counter array holds **AB + C**. Note that the final result is binary, which enables direct cascading of multiple tuGEMM units as input values to the vector generators are binary.

C. Parallel Architecture

A key observation is that the N computation *steps* are independent of each other. Hence, unlike serial, the parallel architecture (Fig. 2) computes all the N *steps* in parallel giving it its name, and is designed for reduced latency at the cost of increased area and power. To achieve this, it integrates the two *vector generators*, and the column and row counters into a single *vector counter* that is replicated N times. It also houses an MxP array of *output adder cells* instead of the output counter array as in serial architecture, where each cell is now capable of adding the counts from all N steps in parallel. Note that there is no need for an index counter used earlier to serialize the N steps. The two main components here are:

1) vector counters: The i^{th} *vector counter* generates unary signals for i^{th} column from **A** and i^{th} row from **B**. Once all the M column counters within a vector counter reach zero, it asserts its *col_done* signal. The GEMM computation finishes when all the vector counters assert this signal, generating *output_ready* signal via an AND gate. The vector counters output N sets of M-dimensional *unary* and *neg* signals from the left and N sets of P-dimensional *unary* and *neg* signals from the top, to be used by the output adder array.

2) output adder array: This component (Fig. 3) holds majority of the hardware modifications with respect to the serial design. Firstly, the counter in the output counter cell of serial architecture is replaced by an adder and a register for accumulation, as necessitated by the requirement for computing all *steps* in parallel. Secondly, each *output adder cell* is capable of processing N different pairs of *unary* and *neg* signals in parallel. Each of the N pairs generates '1', '-1' (*neg* block is used to generate '-1' in two's complement) or '0' based on the *unary_col/row* and *neg_col/row* signals controlled using a simple multiplexer, that are fed into a binary adder which accumulates into a register which holds final value for that cell in the GEMM output, after *output_ready* is asserted.

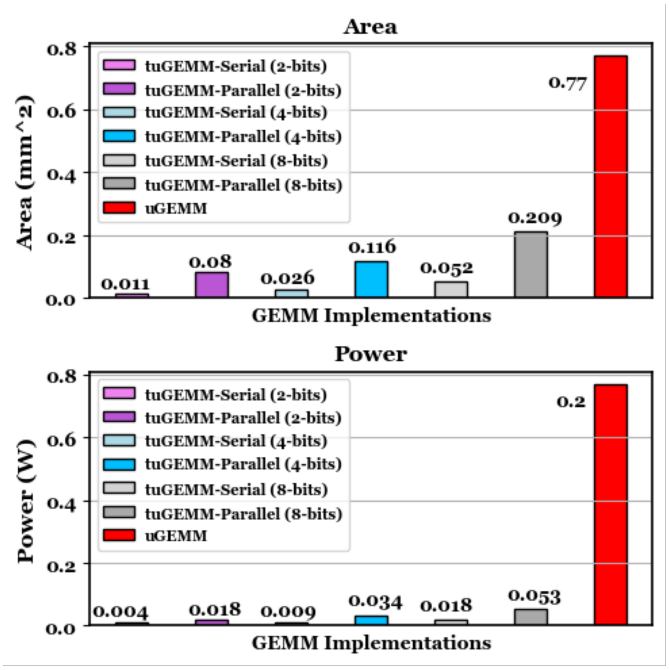

Fig. 4: PPA comparison for 16x16 GEMM implementations (serial/parallel tuGEMM vs. uGEMM) across 2, 4, 8 bitwidths.

III. Evaluation and Results

A. Post-Synthesis Area-Power Evaluation

Serial and parallel tuGEMM designs are implemented in System Verilog, and synthesized with Nangate45 Open Cell Library using Synopsys Design Compiler. Post-synthesis area and power are compared against uGEMM [21] for 8-bit 16x16 matrices (M=N=P=16) at 400 MHz (uGEMM uses this configuration). We further extend the design space to larger 32x32 matrices and lower precision (4 bits and 2 bits).

All post-synthesis tuGEMM numbers are reported in Table I, with 16x16 values illustrated in Fig. 4, alongside uGEMM.

TABLE I: 45nm post-synthesis tuGEMM area-power (16x16 and 32x32 for 2, 4, 8 bits). 16x16 uGEMM baseline included.

GEMM Hardware	Bit-Width	Area (mm^2)	Power (W)	Area (mm^2)	Power (W)
		16x16		32x32	
tuGEMM (serial)	2	0.011	0.004	0.044	0.016
tuGEMM (parallel)	2	0.080	0.018	0.347	0.083
tuGEMM (serial)	4	0.026	0.009	0.099	0.034
tuGEMM (parallel)	4	0.116	0.034	0.506	0.145
tuGEMM (serial)	8	0.052	0.018	0.198	0.068
tuGEMM (parallel)	8	0.209	0.053	0.794	0.202
uGEMM (baseline)	8	0.770	0.200	-	-

It can be observed that both serial and parallel tuGEMM are significantly more area-power efficient with respect to uGEMM. The parallel design consumes 3.7x and 3.8x less area and power, respectively, while the serial design reduces them by 14.8x and 11.1x, respectively. Also, serial design incurs 5.2x and 3.7x less area and power than parallel design. Works in [11], [12], [18], [20] suggest very low bit-widths are sufficient for DNNs to perform training and inference without significantly affecting the accuracy. tuGEMM shows significant PPA benefits in transitioning to very low bit-widths. On average, for every 2x reduction in bit-width, the area, power and delay are reduced by 2.1x, 2x, and 1.2x for serial, and 1.6x, 1.7x, and 1.1x for parallel designs, respectively.

Scaling matrix sizes, area and power for 32x32 tuGEMM increase by 4x compared to 16x16, as expected. An interesting observation here is that 32x32 parallel tuGEMM incurs similar area and power as 16x16 uGEMM (both 8-bit); 32x32 serial tuGEMM is more than 3x area-power efficient than 16x16 uGEMM. This superiority in area and power for tuGEMM arises by trading off latency, which is discussed in detail next.

B. Latency Evaluation

In this section, latencies for a complete GEMM compute are assessed. Assume w is the input bit-width for this discussion.

1) Worst-Case Latency: With two's complement numbers, the largest representable value is 2^{w-1}. As the column/row counters perform unary encoding, it can take up to 2^{w-1} cycles for the row counter to reach zero and $(2^{w-1}) * (2^{w-1}) = (2^{w-1})^2$ cycles for the column counter to reach zero and generate the maximum value in any *step*. Since serial tuGEMM operates through N such steps serially, it can take up to a maximum of $N * (2^{w-1})^2$ cycles in the worst case. As can be seen from Table I, the increase in area and power for parallel design compared to serial design is less than N-fold, potentially resulting in an overall boost in energy efficiency.

Worst-case latency scales exponentially with bitwidth, hence tuGEMM is best suited for low precision. However, average-case latencies for real workloads can be much lower depending on the frequency of large values.

2) Average-Case Latency for Edge AI/DL: Given the input matrices may not hold the largest absolute value, the actual latency can be significantly lower in most cases. In order to profile maximum values, we use a representative edge DNN workload, INT8 quantized ResNet18. During inference, in PyTorch, we keep track of the maximum values within each intermediate feature map and calculate the total number of times each value between 0 and 128 (maximum magnitude for 8-bit signed values) manifests as the maximum value within a feature map. This frequency of occurrence is plotted as percentage in Fig. 5. It illustrates that close to 8% of the operations have 0 as the maximum value, about 50% have maximum values less than 50 and 90% have values less than 80. The average-case maximum value for ResNet18 can be calculated as area under the blue curve, which gives 41 (3x lower than 128). As a result, tuGEMM's average-case latency is significantly (10x) lower. This demonstrates the efficacy of tuGEMM in typical edge AI scenarios where much of the latency can be hidden due to sparsity of data values. An accuracy evaluation on the same multi-layer perceptron from [21] yields 96.08% for tuGEMM (exact) as opposed to 94.7% for uGEMM (approximate). Exact compute becomes very important for lower precisions as any approximations can further exacerbate the quantization penalty on accuracy.

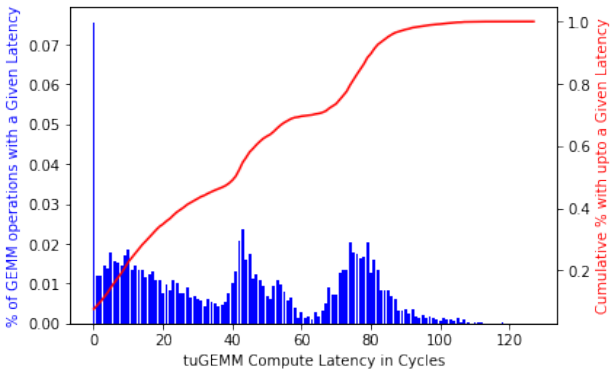

Fig. 5: Percentage of GEMM operations that involve the corresponding X-axis values as the maximum magnitude during inference of INT8 quantized ResNet18. Right Y-axis plots the cumulative percentage of operations with maximum values less than or equal to the corresponding X-axis value.

IV. Conclusion and Future Work

This work introduces a novel temporal unary GEMM design, tuGEMM, capable of exact compute with very high area-power efficiency. In 45nm CMOS, 8-bit 16x16 serial tuGEMM consumes just 0.05 mm^2 area and 18 mW power. The parallel design reduces serial latency by 16x while incurring only an increase of 5x/4x in area/power. Compared to state-of-the-art unary stochastic uGEMM, serial and parallel tuGEMM are about 15x/11x and 3.7x/3.8x more efficient in area/power respectively. This does incur a latency penalty which can be partially mitigated in tuGEMM by exploiting data sparsity and frequently occurring small values. For 4-bit and 2-bit precisions, tuGEMM consumes minimal area and power with very reasonable latency, and thus can be excellent candidates for low-precision always-on edge-AI devices. Future research plans include exploring different input encodings targeting latency optimization, and incorporating tuGEMM in DLAs.

REFERENCES

[1] A. Alaghi and J. P. Hayes, "Exploiting correlation in stochastic circuit design," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 39–46.

[2] Brainchip Holding Ltd., "Akida NSoC," https://brainchipinc.com/akida-neural-processor-soc/.

[3] S. Cass, "Taking ai to the edge: Google's tpu now comes in a maker-friendly package," *IEEE Spectrum*, vol. 56, no. 5, pp. 16–17, 2019.

[4] M. Ditty, A. Karandikar, and D. Reed, "Nvidia's xavier soc," in *Hot chips: a symposium on high performance chips*, 2018.

[5] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[7] V. T. Lee, A. Alaghi, and L. Ceze, "Correlation manipulating circuits for stochastic computing," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1417–1422.

[8] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2019.

[9] S. Liu and J. Han, "Energy efficient stochastic computing with sobol sequences," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 650–653.

[10] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 34–37, 2020.

[11] N. Mellempudi, A. Kundu, D. Das, D. Mudigere, and B. Kaul, "Mixed low-precision deep learning inference using dynamic fixed point," *arXiv preprint arXiv:1701.08978*, 2017.

[12] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[13] H. Nair, J. P. Shen, and J. E. Smith, "A microarchitecture implementation framework for online learning with temporal neural networks," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2021.

[14] C. Nugteren, "CLBlast," https://github.com/CNugteren/CLBlast.

[15] Nvidia, "cuBLAS," https://docs.nvidia.com/cuda/cublas/index.html.

[16] R. Pilipović, V. Risojević, J. Božič, P. Bulić, and U. Lotrič, "An approximate gemm unit for energy-efficient object detection," *Sensors*, vol. 21, no. 12, p. 4195, 2021.

[17] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, 2016.

[18] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1796–1807, 2020.

[19] N. Thompson, K. Greenewald, K. Lee, and G. Manso, "The computational limits of deep learning," *arXiv preprint arXiv:2007.05558*, 2020.

[20] N. Wang, J. Choi, and K. Gopalakrishnan, "8-bit precision for training deep learning systems," 2018.

[21] D. Wu, J. Li, R. Yin, H. Hsiao, Y. Kim, and J. San Miguel, "Ugemm: Unary computing architecture for gemm applications," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 377–390.

[22] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 634–643.