TangramFP: Energy-Efficient, Bit-Parallel, Multiply-Accumulate for Deep Neural Networks

Yuan Yao
yuan.yao@it.uu.se
Uppsala University
Sweden

Xiaoyue Chen[†]
xiaoyue.chen@it.uu.se
Uppsala University
Sweden

Hannah Atmer
hannah.atmer@it.uu.se
Uppsala University
Sweden

Stefanos Kaxiras
stefanos.kaxiras@it.uu.se
Uppsala University
Sweden

***Abstract*—As energy consumption becomes a primary concern for deep learning acceleration, the need to optimize not only data movement but also compute is becoming important. The basic element of compute, the Multiply-Accumulate (MAC) unit, performs the operation $X \cdot Y + Z$, comprises the compute cores of systolic arrays such as Google's TPU or Nvidia's Tensor Cores, and it is found in practically every deep neural network (DNN) accelerator.**

In this work, we aim to reduce the energy needs of bit-parallel MACs, without perceptible impact on precision, and without affecting the structure of the overall accelerator architecture—in other words, we aim for an energy-efficient drop-in MAC replacement.

Although there is a significant body of work on efficient approximate multipliers and MACs, in this work, we propose a novel approach: a tunable floating-point MAC design, TANGRAMFP, that can deliver the full precision of a standard implementation, yet dynamically adjusts to eliminate ineffectual computation. Different from state-of-the-art approaches that are based on truncated multiplication, TANGRAMFP introduces a new class of multipliers where input operands are split and partial products are selectively generated (by enabling or disabling different areas of the logical multiplier array) and added together. In a hardware implementation, this is achieved by decomposing a large multiplier into four smaller ones, at the same overall hardware cost.

We demonstrate that TANGRAMFP precision can adhere to the same bounds (measured as Unit-in-Last-Place—ULP—error) as standard IEEE FP16 arithmetic and delivers better precision than a state-of-the-art approach based on bit-serial truncated multiplication aimed at eliminating ineffectual computation in DNNs. At the same time, TANGRAMFP is a drop-in replacement for the standard MAC design, having approximately the same mean ULP error, the same area and latency, while achieving up to 36.57% dynamic power savings (27.44% with a mean error close to standard).

I. INTRODUCTION

Accelerators for deep learning perform vast amounts of computation over vast amounts of data, especially for training. This leads to significant energy and power consumption per device [56] (from a minimum of 100W to 20kW for wafer-scale integration [32]). In recent years, the emphasis on optimizing for energy and power efficiency was primarily placed on optimizing data movement. This led to the development of seminal approaches for reducing the cost of data movement, e.g., [10], [11]. With increasing on-chip memory reaching today 100's of MiB [2], [35], and increasing *reuse* of the on-chip data, the relative contribution of compute in energy and power consumption also increases.

In both training and inference, the fundamental compute operation is the Multiply-Accumulate (MAC), typically employed in dot-products. Due to the dominance of the dot-product in deep learning, a MAC operation naturally forms the basic floating point (FP) unit in AI accelerators, such as Google's TPU [35]–[37], [48], or Nvidia's tensor cores [75].

Generally, MAC designs used in DNN acceleration fall in two categories: *bit-parallel* and *bit-serial*. Bit-parallel MAC designs often offer consistent precision, high performance, and are easy to reuse from design to design. They are preferred in most commercial and high-performance (ASIC or FPGA) designs [6], [9], [12], [18], [23], [27], [35], [37], [48], [66], [72]. Typically, bit-parallel MACs either support the highest precision required by a network but are difficult to efficiently adjust to lower precisions (scale down), or, alternatively, support a lower precision but can be grouped for higher precision (scaled up), albeit at a steep performance cost, e.g., [4], [25]. This is a problem since actual precision requirements vary considerably across different networks or even across the layers of the same network [24], [39]. Thus, bit-parallel MACs typically process more bits than needed, leading to inefficiency. In contrast, bit-serial approaches [5], [7], [38], [44], [45], [65] offer the flexibility to adjust precision dynamically at runtime, making them particularly adept in exploiting *ineffectual computation* for energy-efficiency. Other works such as [42] introduces mixed-precision FP operations, allowing the programmer or compiler to statically select the appropriate instruction, which may however overlook optimization opportunities that arise dynamically during runtime.

While there are many bit-serial proposals for exploiting ineffectual *integer* computation (for inference), the state-of-the-art for *floating point computation* is the bit-serial FPRaker [7]. As far as we know, there is no comparable approach for bit-*parallel* MAC designs. The question our work aims to address is whether it is possible to effectively benefit from ineffectual computations in bit-parallel floating point designs for both inference and training.

There are good reasons to aim for bit-parallel designs, as bit-serial approaches bring their own set of constraints

[†]First student author.

in an accelerator architecture: i) they are often multi-cycle designs with a *value-dependent-latency* which may necessitate extensive buffering to smooth out variability and synchronize communicating units, and ii) they often impose constraints on data movement as they must treat data as bit streams. While there are many promising proposals for bit-serial designs, as far as we know, they are not the implementation of choice for the leading high-performance commercial accelerators. On the other hand, bit-parallel designs have the potential to make immediate impacts on energy and power consumption as they can be integrated into existing accelerator architectures with minimum effort. The goal in this case would be to achieve lower energy at the same performance and area.

Our proposal, TANGRAMFP, aims for "one-shot" bit-parallel MACs (pipelined if needed), *avoiding* ***variable*** *multi-cycle timing* that complicates the macro-architecture (e.g., of a systolic array) by requiring interleaving and extensive buffering to absorb timing variations [7]. In other words, we aim for a *drop-in replacement of existing MAC units* found in commercial designs. We aim for the same or better latency, and the same or better area. Finally, it is important to be able to deliver the full accuracy of the baseline MAC as defined by the corresponding FP format (e.g., IEEE-754 FP16 with denormals and rounding).

For floating point computations—the focus of our work—ineffectual computation comes in two kinds: *value sparsity* and *relative sparsity*. Value sparsity refers to the zero operands in a multiplication or addition. Relative sparsity refers to the large relative difference between non-zero operands in an addition that causes the $small + BIG = BIG$ behavior.

Value sparsity is trivially exploited (to increase energy efficiency) in any design, by detecting zero operands [28], [55]. It has been effectively exploited in previous proposals, e.g., [6], [7], [28], [55], and we consider it as a default technique for the baseline. Besides, while value sparsity exists in abundance in Convolutional Neural Networks (CNNs) [33], e.g., due to the extensive use of ReLU activation functions, it seems that in more recent important networks, such as Transformer [71] and Diffusion models [57], it is not so prevalent—see Figure 1.

Relative sparsity is the key inefficiency to exploit in a bit-parallel FP MAC. A MAC operation $X \cdot Y + Z$ multiples two FP operands X and Y by adding their exponents and multiplying their mantissas. To perform the addition with Z, the XY product and the addend Z are *aligned* to have the same exponent. The alignment may cause the product XY to be shifted right (towards the least significant end), effectively pushing a potentially large part of its computed mantissa beyond the representation range. Relative sparsity has not been targeted directly in existing work, resulting in much ineffectual computation in DNN training. Our key insight is that FP *relative sparsity* is prevalent in DNN workloads, and we can exploit it by dynamically detecting such sparsity to reduce ineffectual computation accordingly.

Relative sparsity is prevalent because in both inference and training, the addend Z is generally *larger* than the product XY. Figure 2 shows the distribution of the exponent difference, $e_z - (e_x + e_y)$, between the addend Z and the product XY for ResNet18 [31] (inference and training), Stable Diffusion, and Llama2 (inference only), in FP16 [3]. Positive values correspond to an alignment shift (i.e., dropping significant bits) for the smaller XY product; negative values indicate that the addend Z is *smaller* than the product XY and it should be right-shifted instead. We observe similar distributions to the one shown in Figure 2 in all the networks we examined.

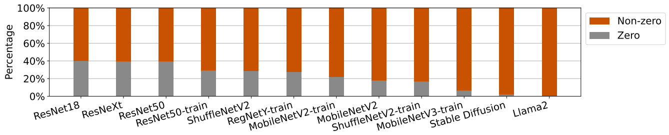


Fig. 1: The diminishing value sparsity in DNNs (FP16 values with denormals not counted as zeroes).

The fact that the distribution of alignment shift is significantly skewed to the right of zero, means that MACs rarely require the full precision of the mantissa product. In fact, the larger the alignment shift, the larger the error tolerance is of the product. Shifts larger than 11 bits in FP16 effectively shift the product mantissa to zero, exhibiting the well known FP behavior $small + BIG = BIG$. The aim of TANGRAMFP is to *dynamically exploit the skew in the exponent difference distribution to optimize the energy-efficiency of the mantissa multiplication*, which represents a large part of the cost of a MAC unit [52].

A well-known approach to discard the least significant part of the mantissa computation is *truncated multiplication* [63]. However, truncated multiplication is plagued by large errors that need to be corrected by adding a *correction factor* to the final result [68]. To compute a correction factor the bits that did not participate in the computation must be used [21], [68]. But this erodes the potential benefit and makes it complex to adjust dynamically [22]. Alternatively, a "buffer zone" of a few bit positions can be used to truncate the mantissa *less* than what is actually desired. Such a truncated multiplication approach is taken in FPRaker which advocates to exploit *term sparsity* [7]. To the best of our knowledge, no approach exists that is solely focused on exploiting *relative sparsity* for a *bit-parallel* MAC. Our work proposes such an approach.

A. TANGRAMFP *Contributions*

We propose TANGRAMFP, a novel bit-parallel approach for energy-efficient MACs. The novelty of TANGRAMFP is that we disassemble the multiplication into multiple smaller multiplications which we (selectively) re-assemble to create the final result (section III). The key insight is that, due to the properties of multiplication, this takes approximately the same hardware as the baseline MAC. The idea is to split the two N-bit wide X and Y multiplication operands, each into two (or more) parts of p-bit and q-bit widths ($N = p + q$). The resulting (smaller) partial products are selectively assembled to yield the full result. By omitting some of the partial products,

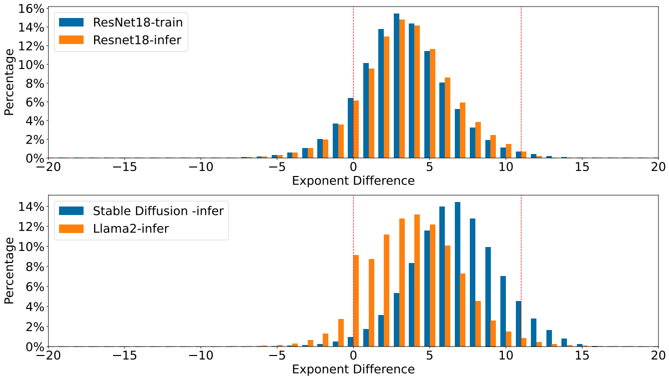


Fig. 2: Exponent difference, $e_z - (e_x + e_y)$, distribution (non-zero operands). Top: ResNet18 training and inference. Bottom: Stable Diffusion (inference) and Llama2 (inference).

a lower energy consumption for the final (reduced-precision) result can be achieved. As in related work [7], we exploit the distribution of exponent differences in MACs to dynamically enable or disable parts of a TANGRAMFP multiplier to yield a desired precision for the product. TANGRAMFP is a true drop-in replacement for a standard MAC as it guarantees: i) full precision (indistinguishable from the baseline)—something that other approximate approaches, in general, fail to do [46], [47], [62], ii) same or lower area, iii) same or lower latency, iv) same or lower power. When TANGRAMFP is used in its full mode, it behaves identically to a standard MAC in terms of error, performance, area, latency, energy/power. When TANGRAMFP is used in one of its lower-power modes, it delivers a reduced-precision result (but always within well-established error bounds that are close to the standard) at the same performance but lower energy consumption.

TANGRAMFP actually comprises a family of possible designs for the multiplier, depending on the chosen split of the operands. Although we present the general design principle of TANGRAMFP, in this work, we select a particular design point (for IEEE-754-2019 FP16 [3]), which we call 1:5:5 (subsection III-A). A full design space exploration for other TANGRAMFP implementations is left for future work.

To support our claims:

- We perform a detailed error characterization (section IV) to derive error bounds for TANGRAMFP and for a bit-serial truncated multiplication MAC similar to FPRaker [7] in terms of error measured in *Unit-in-Last-Place* (*ULP*) [3]. We show that TANGRAMFP: i) can offer full precision when needed, and ii) has tighter error bounds and produces lower mean error than comparable bit-serial truncated multiplication, in its reduced modes.
- We model a hardware implementation (section V) and demonstrate both analytically and with hardware synthesis that a TANGRAMFP MAC requires (slightly) less hardware (area) as the corresponding baseline MAC, has the same latency, and the same power in full mode, while it saves up to 36.57% of power in its reduced-precision modes, and up to 27.44% with a mean error close to standard.
- Evaluation (section VI): We evaluate TANGRAMFP by using a custom execution framework where we replace the native hardware addFP/multiplyFP/MAC operations with the corresponding TANGRAMFP operations. Combined with detailed results of hardware synthesis, our evaluation framework derives power and numerical accuracy results for a TPU-like architecture compared to the same baseline architecture using FP16. As a same-latency drop-in replacement of a standard MAC, TANGRAMFP has no effect on performance.

II. BACKGROUND

In this section, we give a short recap for floating point representations and their error. We also introduce the state-of-art MAC prior to our work — FPRaker [7].

A. FP representations and error

Although DNN inference can be effectively performed with quantization to small integers (e.g., INT8) [24], [29], floating point formats dominate DNN training. There are several formats in use today in GPUs and DNN accelerators, most prominently the IEEE-754 FP32 and FP16 formats [3], Google's bloat16 [1], and Nvidia's Tensor Flow 32 (TF32) [13], summarized in Figure 3. FP Formats with even less precision, e.g., FP8 [43] are actively being developed, but in many cases training requires higher precision. There are also *block floating point* formats [16], [17], [40], [41], [67], [77] that share an exponent among a small group of mantissas that are well fit for group multiply-accumulate operations. TANGRAMFP is not format-specific and applies equally well to all of the above, but to keep this work focused on the basic principles we choose to demonstrate TANGRAMFP for single (or group for evaluation) MAC units in IEEE-754 FP16.

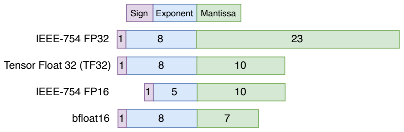


Fig. 3: FP formats for DNN training.

Multiplication: Multiplying two floating-point numbers involves multiplying their mantissas and adding their exponents. After the multiplication, the result is normalized by adjusting the exponent and shifting the mantissa if necessary. Rounding is then applied to ensure accuracy within the precision of the floating-point format.

Addition: When adding two floating-point numbers, the first step is to align their exponents by adjusting the mantissas. Once the exponents are aligned, the mantissas are added or subtracted, depending on the signs of the numbers. After the addition, the result is normalized by shifting the mantissa and adjusting the exponent if necessary. Rounding is then applied to ensure that the result is represented accurately within the precision of the floating-point format.

Fused Multiply-Accumulate (FMAC) operations [34] combine a multiplication with an addition, with the benefit

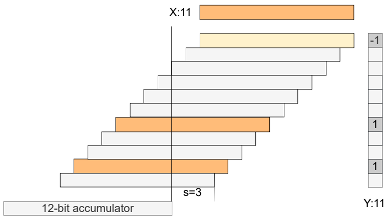


Fig. 4: Diagrammatic representation of an FPRaker MAC.

of not having to normalize and round the result of the multiplication before being used in the addition (as two separate operations would do). FMACs shift the mantissa product (or equivalently, the addend mantissa) to have the same exponent as the addend (or the product). This shift is called *alignment shift*. Not only an FMAC is faster and more efficient than two separate operations [20], but it is also more accurate.

All floating point representations are approximations of real numbers, and any operation that produces a result that does not fit in the representation introduces *rounding error*. The IEEE-754 floating-point standard requires any hardware implementation to produce a result with an error of no more than 0.5 *Unit-in-Last-Place* (*ULP*) when rounding to the nearest value (Round-To-Even or RTE), and less than 1 ULP when rounding up, down, or toward zero (Round-to-Zero, Round-to-Positive-Infinity, and Round-to-Negative-Infinity) [3]. The ULP of a real number x, when represented in a given floating point format, is the distance between the two closest floating point numbers a and b that surround x: $a \leq x \leq b$, $a \neq b$, provided that the number x has a valid exponent in the representation (i.e., the exponent has not exceeded the maximum exponent of the representation). For example, the ULP for an IEEE-754 FP16 number whose exponent is e represents the value 2^{e-10}.

Rounding is why two separate operations always have more error than a fused operation — MACs with 2 roundings have error up to 1 ULP while FMACs with a single rounding have error up to 0.5 ULP. Rounding multiple times is also a weakness in any approximate multiplication algorithm that rounds individual partial products before they are combined to form the final product: rounding error compounds.

B. FPRaker

FPRaker is a processing element for composing DNN accelerators [7]. It is a group MAC that processes 8 multiplications concurrently and then accumulates them together. To focus on FPRaker's core idea, we introduce it as if it only processes one multiplication at a time.

Figure 4 depicts diagrammatically how an FPRaker MAC multiplies two 11-bit mantissas and accumulates them with an alignment shift of 3. FPRaker first encodes Y into Canonical Signed Digit Representation (CSD). CSD can reduce the number of non-zero bits. In this example, $Y = 575 = 0100100000\bar{1}$. Then FPRaker serially (i.e., bit by bit) multiplies every non-zero bit with X, shifts the partial product, and adds to the accumulator. It skips all the zero bits, hence all the uncolored boxes in Figure 4 are never computed. Furthermore, if a partial product is completely out of the range of the accumulator (e.g., the -1 partial product), it is also skipped. If a partial product is partially out of the accumulator range (e.g., the two 1 products), it is rounded so that only the bits within the accumulator range are kept. In this way, FPRaker skips the computation with zero bits and out-of-range bits.

FPRaker [7] claims that it "*... is not an approximation and does not affect numerical accuracy.*" However, this claim is inaccurate, and we will show why in section IV.

III. A Modular Multi-precision Multiplier

In this section, we show that a multiplier can be broken down to several smaller multipliers. We also show how to selectively assemble the results from the smaller multipliers.

We need to perform the multiplication $X \cdot Y$, where X and Y are N-bits wide. We use the $X{:}N$ notation to denote the bit width N of a number X. We split $X{:}N$ into two parts of p and q bits ($N = p + q$), respectively: $A{:}p$ and $B{:}q$; similarly, $Y{:}N$ into $C{:}p$ and $D{:}q$. X and Y and their product are now expressed as:

$$X = A \cdot 2^q + B \tag{1}$$

$$Y = C \cdot 2^q + D \tag{2}$$

$$XY = AC \cdot 2^{2q} + AD \cdot 2^q + CB \cdot 2^q + BD \tag{3}$$

Equation 3 shows that we can split the $N \times N$-bit multiplier to 4 smaller parts — a $p \times p$-bit multiplier (AC), two $p \times q$-bit multipliers (AD and CB), and a $q \times q$ multiplier (BD). The final multiplication result can be assembled from shifting and adding the partial results together. The partial result AC contributes more to the final result than BD due to the different left shifts.

An interesting fact is that the 4 configurations in Table I require the exact same hardware multipliers. This gives TangramFP the ability to modulate precision versus energy consumption (by selectively omitting some of the partial products) in four different ways.

X	Y	XY product
$A{:}p$, $B{:}q$	$C{:}p$, $D{:}q$	$AC \cdot 2^{2q} + AD \cdot 2^q + BC \cdot 2^q + BD$
$A{:}q$, $B{:}p$	$C{:}p$, $D{:}q$	$AC \cdot 2^{p+q} + AD \cdot 2^p + BC \cdot 2^q + BD$
$A{:}p$, $B{:}q$	$C{:}q$, $D{:}p$	$AC \cdot 2^{p+q} + AD \cdot 2^q + BC \cdot 2^p + BD$
$A{:}q$, $B{:}p$	$C{:}q$, $D{:}p$	$AC \cdot 2^{2p} + AD \cdot 2^p + BC \cdot 2^p + BD$

TABLE I: Possible configurations for a $p{:}q$ split TangramFP

For FP16 which has an 11-bit mantissa, there are five possible $p{:}q$ splits (1:10, 2:9, 3:8, 4:7, 5:6). The $p{:}q$ split determines the bit width of the partial multipliers. For a given $p{:}q$ split and configuration, four different modes can be used (discussed below in subsection III-C) to trade precision versus energy consumption. The design space is large — containing 5 different splits × 4 different configurations. A systematic exploration of the design space of all possible splits is left for future work. In this work, we introduce the TangramFP concept and explore one configuration in depth.

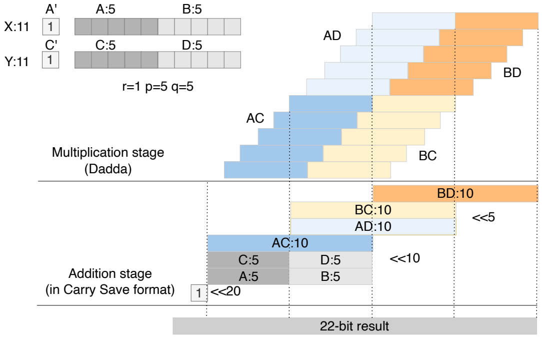


Fig. 5: The 1:5:5 split.

A. *Exploiting the leading bit*

For a normal FP number, the leading bit of its mantissa is implicit and is necessarily 1. We show how to reduce TANGRAMFP multiplier width by exploiting this feature.

If we prepend a leading 1 to X and Y, denoted as X' and Y', we have

$$X' = 2^{p+q} + A \cdot 2^q + B \tag{4}$$

$$Y' = 2^{p+q} + C \cdot 2^q + D \tag{5}$$

$$\begin{aligned} X'Y' &= 2^{2(p+q)} + (X+Y) \cdot 2^{p+q} \\ &+ AC \cdot 2^{2q} + AD \cdot 2^q + BC \cdot 2^q + BD \end{aligned} \tag{6}$$

The first line of Equation 6 contains only additions, and the second line is the same as Equation 3. This method reduces the multiplier width by one bit in both operands: for the 11-bit FP16 mantissas we only need to perform a 10 × 10 multiplication. Furthermore, by picking $p = q = 5$ (i.e., a 1:5:5 split), we get a symmetrical design, shown in Figure 5.

This 1:5:5, split comes with a set of interesting properties:

- Due to symmetry, all of four spatial configurations (Table I) of a 5:5 split are identical (Figure 5). On one hand, we do not have to (dynamically) select which one to use, but on the other we lose the flexibility of choice.
- There is only one type of multiplier, a $5b \times 5b$, resulting in a consistent latency for the partial products.
- Four $5b \times 5b$ multipliers in parallel, have a significant smaller latency than a $11b \times 11b$ multiplier (see section V), allowing room to hide the latency of the additions of the partial products (in Carry-Save format).
- All the partial products are of the same width, 10 bits, and their alignment for the addition to produce the final product is static (no multiplexers are needed as in the case of the four different spatial configurations).

The downside of a 1:5:5 split is that we have to add up to six partial products (seven if one counts the 2^{20} term). However, as we show in section V, the latency of adding six partial products (of an effective 10-bit width) is small using 3:2 Carry Save Adders (CSA), and carrying the partial products in a Carry-Save format. This is the standard practice, for example, in fused MAC designs [20]. For the rest of this paper we focus on studying TANGRAMFP with a 1:5:5 split—a full design space exploration of other possible splits is left for future work.

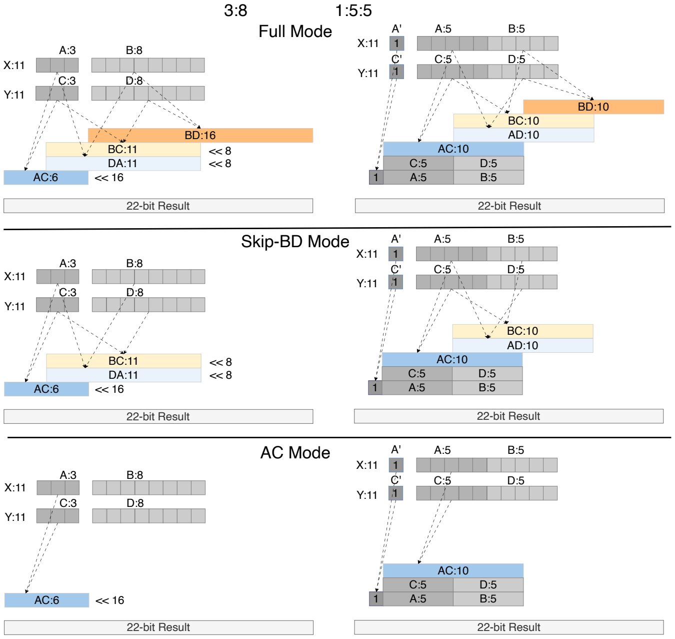


Fig. 6: 3:8 and 1:5:5 splits featuring three modes of operation: i) Full Mode; ii) Skip-BD Mode; iii) AC Mode.

B. *Denormals*

Denormal numbers are a way to extend the exponent range to smaller numbers. When a denormal number is detected as an input operand to the MAC, TANGRAMFP reverts to full precision. In the 1:5:5 split, some of the partial products become zero (depending on whether A', or B', or both, are zero), and the multiplier produces a precise result. Handling normalization and rounding with denormals is delegated to the mechanisms of a standard implementation, depending on the FP format. However, handling of denormals is generally slow and expensive [64] and, for DNN acceleration, they can be avoided by having a larger exponent range (e.g., TF32 or bfloat16) [50]. In FP16, TANGRAMFP produces the precise result (referred to as `Full Mode` in the next section) for denormals and employs the mechanisms of the standard implementation to handle them.

C. *Modes*

TANGRAMFP determines the required precision of the multiplication by considering the alignment shift, i.e., $e_z - (e_x + e_y)$. The three exponents are routinely used in existing fused MAC designs for comparing XY and Z to determine the alignment shift for the addition and perform it in parallel with the mantissa multiplication [20], [51], [52]. Similarly to other work [7], we use the exponents to determine the configuration of the TANGRAMFP multiplier for energy savings while delivering the needed precision.

The advantage of a TANGRAMFP split over the monolithic multiplier is that we can modulate the precision of the result versus energy consumption by enabling or disabling individual parts of the multiplier. In particular, there are fourmodes of operation, three of which are shown in Figure 6 for a 3:8 split and a 1:5:5 split, respectively. The four modes are:

- `Full Mode` : In full mode, all the partial products are taken into account producing a precise FP16 result (Figure 6 top).
- `Skip-BD Mode`: In this mode, the *tail* partial product, BD, responsible for the low-order bits of the result is skipped (Figure 6 middle). This can lead to modest energy savings while preserving accuracy in the high-order result bits.
- `AC Mode`: By keeping only the *head* partial product, AC, we maintain some high-order bit accuracy while saving considerable energy (Figure 6 bottom). In this mode, A and C are rounded representations of the full $\{$A.B$\}$ and $\{$C.D$\}$, respectively.
- `Null Mode`: The `Null Mode` is used in two situations: i) when at least one of the operands of the multiplication is zero, and ii) when the alignment shift s is greater than the mantissa width: $s > 11$. `Null Mode` discards both the multiplication and the addition.

The question is how do we select which mode to use? We answer this by means of a detailed error characterization.

IV. Error Characterization

IEEE 754 states that a standard FMAC shall only have rounding error because it shall compute $X \cdot Y + Z$ "*as if with unbounded range and precision, rounding only once to the destination format*" [3]. Approximate MACs, on the other hand, have both rounding error and error from the approximate computation of $X \cdot Y + Z$. They often reduce the amount of computation by approximating the multiplication because it is the most costly operation in a MAC.

We focus on the multiplication approximation error of approximate MACs. It should be clear that any multiplication algorithm that omits any term, no matter how small, from the full multiplication is by necessity approximate. It is not possible to get precisely the same result as the full multiplication unless all the terms are taken into account. This holds, of course, for all forms of *truncated multiplication* [21], [22], [63], [68], as well as for our approach.

The orange-shaded area in the bottom row of Figure 7 corresponds to the omitted computation of FPRaker. The orange-shaded areas in the first two rows correspond to the omitted computation in TangramFP's `AC` and `Skip-BD` modes.

To assess the error of the two approaches with respect to a standard (full) FMAC, we present a detailed error characterization including an error bound analysis and mean error characterization. We use error measured in *Unit-in-Last-Place* (ULP). Error in ULP is a commonly used metric in numerical analysis of error bounds for floating-point computations [14], [19], [58].

We use exhaustive search to find the upper and lower bound of errors in a standard FMAC, FPRaker, and TangramFP under various alignment shifts. The bound analysis reveals the MAC result's error range. Those findings are discussed in subsection IV-A. We randomly sample actual MAC operations on DNN workloads [31], [54], [57], [61], [69], [76] and characterize the average numerical accuracy of different MAC algorithms. The results are presented in subsection IV-B.

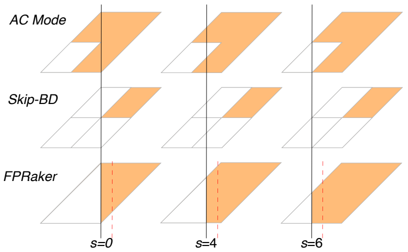


Fig. 7: A diagrammatical explanation for the behavior of TangramFP `AC Mode`, `Skip-BD Mode`, and FPRaker as a function of the alignment shift s. The shaded areas represent the omitted part of the multiplication that causes the error.

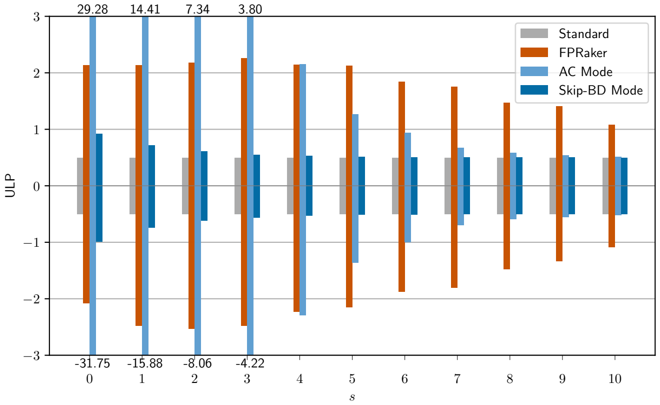


Fig. 8: Error bounds in ULP.

A. Error Bounds

Figure 8 shows how the MAC error ranges under different alignment shifts. We compare TangramFP to *Standard* [3] and FPRaker [7] and show that our approach, for the appropriate s, has comparable error bound range to the standard multiplication and significantly lower than that of the FPRaker approach.

Standard refers to the standard FP16 fused multiply-add algorithm with a single rounding (into an FP16 result) in the end [3] (we use Round-To-Even, RTE, rounding). Therefore, its error results purely from rounding. Note, also, that our `Full Mode` it is equivalent to *Standard*.

Besides `Full Mode`, TangramFP uses either `AC Mode` or `Skip-BD Mode` to produce a 22-bit approximate mantissa. In `AC Mode`, A and C are rounded (RTE) versions of the $\{$A.B$\}$ and $\{$C.D$\}$. This rounding takes place before the multiplication.[1] In `Skip-BD Mode` neither A nor C are rounded (i.e., they are simply the truncated part of $\{$A.B$\}$ and $\{$C.D$\}$ respectively).

[1] RTE rounding of A or C may overflow them to 6 bits. However this is a special case that is easily detected and handled. For example, an overflow on RTE(A) is only triggered when A is '11111' and the MSB of B is non-zero. In such a case, RTE(A) becomes '100000', simplifying the multiplication process to a shift of the multiplicand by six positions.

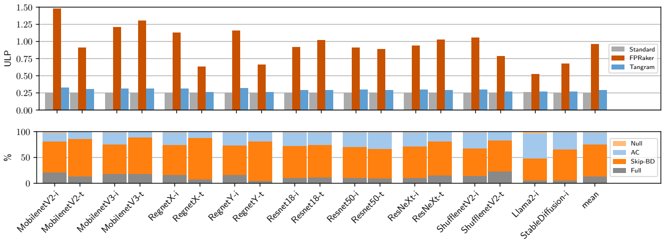


Fig. 9: Top: Comparison of ULP Error of the MAC. Bottom: TANGRAMFP Dynamic: Mode Usage Breakdown

FPRaker's [7] precision claim is inaccurate because it rounds every partial product that crosses the accumulator width boundary and skips the partial products that it deems *out-of-bounds*, causing a significantly larger error than a standard FMAC. This phenomenon is exacerbated as the alignment shift s increases.

Figure 8 confirms the ±0.5 ULP error bound for *Standard* (RTE) under any alignment shifts. The error bounds of our `Skip-BD Mode` are consistently close to the standard multiplication rounding error and much smaller than that of FPRaker (e.g., ±0.51 ULP with `Skip-BD Mode` v.s. ±2.15 ULP with FPRaker when $s = 5$). The more power-efficient AC mode starts to have tighter error bounds than FPRaker when $s \geq 5$. Based on those observations, we can dynamically choose operating mode based on the alignment shift s to reduce computation while retaining almost-standard error bounds.

Figure 7 explains why TANGRAMFP `AC Mode` and `Skip-BD Mode` have much tighter bounds when alignment shift increases, and why it is not the case for FPRaker. The columns in Figure 7 shows how the omitted computation (the orange-shaded areas) changes as alignment shift increases. The areas of omitted computation are constant for `AC Mode` and `Skip-BD Mode`, while it increases by a trapezoidal area whose height is the alignment shift for FPRaker.

For the average error characterization in subsection IV-B, we use 6 as a threshold to use `AC Mode`, i.e., choose `Full Mode` when $s = 0$, choose the more precise `Skip-BD Mode` when $s \in [1, 5]$ (±0.51 ULP to ±0.74 ULP), and the more power-efficient `AC Mode` when $s \geq 6$ (±1.00 ULP to ±0.52 ULP). How different thresholds to use `AC Mode` affect errors is closely examined in subsection VI-C.

B. Mean Error

In addition to the error bound analysis that only provides guarantees for the numerical fidelity, we also need the mean error to characterize what is actually observed in practice. For this, we randomly sample MAC operations using reservoir sampling [73] in the training and inference of several models: MobilenetV2, MobilenetV3, RegnetX, RegnetY, Resnet18, Resnet50, ResNeXt, ShufflenetV2, Llama2, and Stable Diffusion [31], [54], [57], [61], [69], [76]. More details of those models are presented in subsection VI-B.

The top graph of Figure 9 (inference is denoted by a trailing "-i" and training by a trailing "-t" in the network's name) shows the mean (absolute) error of a standard FMAC, FPRaker, and TANGRAMFP which dynamically choose operating modes according to the alignment shift, as described in subsection IV-A. The mean errors across the workloads are 0.25, 0.96, and 0.29 ULP for Standard FMAC, FPRaker, and TANGRAMFP, respectively. TANGRAMFP displays a mean error very close to the standard, and is far smaller than FPRaker. Obviously, such mean error is meaningless if TANGRAMFP uses `Full Mode` 99% of the time. Therefore, we show the bottom graph in Figure 9 that gives a breakdown of the mode usage. Across all workloads, TANGRAMFP uses 13.07% of `Full Mode`, 61.83% of `Skip-BD Mode`, 24.06% of `AC Mode`, and 1.04% of `Null Mode`.

C. Error Tuning

Once a TANGRAMFP MAC is designed with a specific split, the error bounds for its various modes are fixed (from the full precision of the `Full Mode` to the reduced precision of the `AC Mode`). What gives TANGRAMFP its flexibility is that we can tune its behavior by *setting the shift thresholds* where TANGRAMFP changes from one mode to the next, according to the run-time exponent difference of MAC operations. The thresholds are given as parameters to the hardware. Importantly, the thresholds define the guaranteed error bound, so one can always control the numerical stability of TANGRAMFP. On the other hand, one can change the thresholds more aggressively for more power savings at the expense of increased error, depending on how insensitive the target application is to error [30], [53] (see section VI).

V. HARDWARE MODELING

A major benefit of the MAC, compared to two separate FP-Multiply and FP-Add units, is that no rounding (or normalization) is performed to the product XY before being added to the accumulator Z [20], [34]. Instead, only a final normalization and rounding is performed for the result of the addition. TANGRAMFP is a cascade fused-MAC design that performs the multiplication first and aligns the product XY and the addend Z by shifting to the right the *smaller of the two*. This avoids the extra-wide (3× the mantissa width) shifter and datapaths of RS/6000-types designs [34] and leads to area- and energy-efficient implementations [20].

In the TANGRAMFP MAC, the product is *not* rounded (RTE) or normalized before the addition but only once, after the addition. This leads to smaller overall MAC error (than rounding twice) and saves hardware. In contrast, FPRaker cannot replicate this behavior as it has to round each term of the product before adding it to the addend—otherwise FPRaker would not be able to exploit the *term sparsity* of the multiplicand [7]. This incremental—per term—rounding, contributes to FPRaker's increased error compared to TANGRAMFP and standard MAC (section IV).

A. High-performance multipliers

The TANGRAMFP approach is orthogonal to the type of multiplier chosen for its core. To demonstrate its broad applicability, we focus on high-performance, parallel multiplier designs. High-performance multipliers are typically built as reduction tree multipliers, specifically, as Wallace [74] or Dadda [15] multipliers. Reduction tree multipliers do not require complex hardware for the input encoding (e.g., Booth or CSD) before the multiplication, which may introduce an extra pipeline stage in some cases. The Dadda design is both smaller and faster than a corresponding Wallace design [70] but also optimal in minimizing the required hardware [26].

A Dadda multiplier consists of a parallel bit-wise multiplication stage (AND gates), followed by a number or reduction stages that consist of a combination of full adders (*3:2 Counters*) and half adders (*2:2 Counters*). Table II gives the number of reduction stages as function of the *multiplier* width, denoted by N [8]. The last reduction stage produces two rows of results that need to be added together (e.g., with a Ripple-Carry or Carry-Lookahead Adder) to yield the final result. The width of this adder is $N + M - 2$, where M is the width of the *multiplicand*. Note that the final two rows of a Dadda multiplier can be considered as a result in *Carry-Save Format, (CSF)*, that can be transferred as such and fed to *Carry-Save Adders (CSA)*, possibly along with with other CSF results.

Bickerstaff *et al.* analytically derive the hardware cost of Dadda multipliers [8], shown in Table III. From these formulas, we estimate the cost of a full-blown Dadda multiplier versus the cost of the set of smaller Dadda multipliers needed to construct an equivalent TANGRAMFP multiplier for any two-way p:q or three-way r:p:q split. In Figure 10(a), we present an analytical hardware cost estimate for various bit-width *stand-alone* Dadda multipliers, derived from theoretical formulas and translated into gate equivalents. The results demonstrate a significant correlation between these estimates and the actual hardware synthesis outcomes for the same multipliers, validating the area efficiency benefits of segmenting the multiplier into smaller p:q (or 1:p:q) sub-modules. As shown later, this "divide-and-conquer" approach not only provides sufficient margin for the reduction tree and all sub-multipliers, but also potentially reduces the overall area to levels similar to those of the original unsegmented Dadda multiplier. In contrast, despite the efficiencies projected in Figure 10(a), integrating the original Dadda multiplier *within a MAC unit* necessitated the addition of padding gates to manage timing discrepancies caused by the long critical path in the final stage of reduction, as depicted in Figure 10(b). These modifications led to increased area and power consumption, deviating from the initial cost-effectiveness estimates.

Multiplier width (N)	Number of Dadda stages
$2 \le N \le 4$	0–2
$5 \le N \le 6$	3 (e.g., for 1:5:5)
$7 \le N \le 9$	4
$10 \le N \le 13$	5 (e.g, 11x11 Dadda)

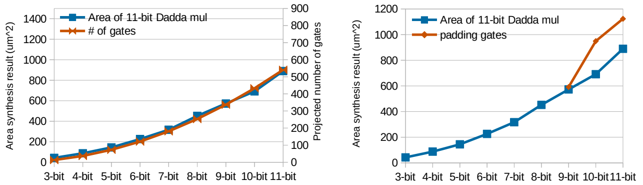
TABLE II: Dadda stages as a function of multiplier width [8].

B. Hardware synthesis

To thoroughly evaluate the hardware costs of TANGRAMFP, we conducted a detailed synthesis of a MAC unit featuring both an 11-bit Dadda multiplier and a 1:5:5 TANGRAMFP with four 5-bit sub-multipliers and a reduction adder-tree of carry-save adders. We used Synopsys Design Compiler (DC) and TSMC's 40 nm low power library. The synthesis emphasized optimizing efficiency in terms of area, timing, and power (max effort). Figure 11 displays the schematic of TANGRAMFP highlighting out the sub-multipliers and the reduction adders, presented in an unflattened format for clarity. The path-slack analysis from DC shows that in contrast to the 11-bit Dadda multiplier, which is bottlenecked by a long critical path at the final 22-bit adder stage, TANGRAMFP utilizes a module-based approach, breaking up the critical path, and leading to more evenly distributed slack.

HW	N multiplier $\times M$ multiplicand
AND gates	$N \cdot M$
(3:2) counter (FA)	$N \cdot M - 2 \cdot (N + M) + 3$
(2:2) counter (HA)	$N - 1$
CLA Adder width	$N + M - 2$
(4:2) CSA Adder width	$2 \cdot N$

TABLE III: Analytical hardware estimates for $N \times M$ Dadda multipliers [8].



(a) Theoretical gate number vs. area.

(b) Area with padding gates.

Fig. 10: Gate number prediction [8] and synthesis area results for Dadda multiplier.

C. Synthesis results

Figure 12(a) presents the power synthesis results for both a standalone multiplier and an MAC unit, each configured with a 1:5:5 TANGRAMFP. Figure 12(b) details the power consumption breakdown of the MAC unit by its various components. Overall, the MAC unit consumes 0.56mW power (with TANGRAMFP in `Full Mode`) and occupies an area of 2903.01μm^2. In the standalone multiplier, TANGRAMFP achieves power savings of 17.05% in `Skip-BD Mode`, 48.84% in `AC Mode`, and 94.47% in `Null Mode`. The corresponding number in an MAC unit are 12.89% in `Skip-BD Mode`, 36.93% in `AC Mode`, and 88.79% in `Null Mode`. As expected, the power savings are less significant in the MAC unit since TANGRAMFP primarily impacts the multiplier and does not affect other components like the FP16 adder.

We evaluate TANGRAMFP's effects on system power consumption with a setting modeling Google TPUv2's [49] architecture. The system features 32×1024 MAC units and 32MB on-chip SRAM (8 banks) as on-chip vector buffer. Our energy evaluation methodology combines functional behavior simulation, system-level estimation, and low-level synthesis. Operational modes (`Full Mode`, `Skip-BD Mode`, `AC Mode`, `Null Mode`) of each MAC unit were obtained via PyTorch simulation. CACTI 7.0 with 40 nm technology was used to estimate the energy consumption of on-chip SRAM weight buffers. Power and area results for the Tangram

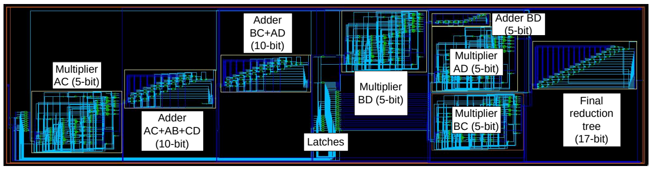
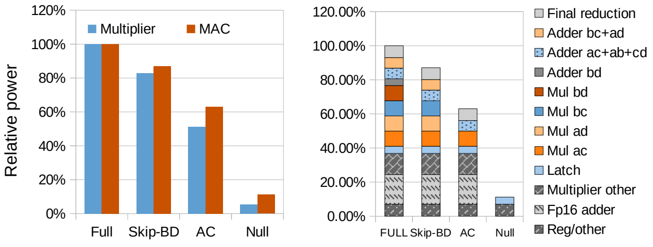


Fig. 11: Block diagram of TANGRAMFP in DC (unflatten), implementing the principle introduced in Figure 6.



(a) Tangram in mul. and MAC

(b) Power breakdown.

Fig. 12: Synthesis results of TANGRAMFP. The MAC unit featured a 11-bit TANGRAMFP consumes 0.56mW power (`Full Mode`) with an area of 2903.01μm^2. In (b), `Reg/other` denotes the output registers and all gluing logic to connect different components. `Mult-other` denotes the logic in the FP16 multiplier for normalization, etc.

multiplier and MAC unit were derived from synthesis. We focus on the dynamic power consumption of the MACs and the SRAM. While static power can comprise a sizable part of the total power consumption, its precise characterization depends on the mix of logic and memory, which can vary from architecture to architecture. Besides, static power is addressed primarily by technology advances, such as the introduction of CFET [60], which may notably reduce SRAM leakage in future nodes. CACTI results show that the SRAM's dynamic power consumption is 21.75W, making up about 54.84% of the total chip dynamic power, *assuming two operands are accessed per multiplication*. In *stationary* designs, where predominately only one operand is accessed per multiplication, the relative contribution of the SRAM to the dynamic power is reduced by half. The MAC units in aggregate consume 17.92W (all `Full Mode`), accounting for about 45.16% of the total dynamic power. `Skip-BD Mode` reduces MAC dynamic power by 5.82%, `AC Mode` by 16.68%, and `Null Mode` by 40.10%. While the analysis presented here is based on offline analysis, details on runtime power savings can be found in Section VI-C(b).

VI. Evaluation

A. Methodology

We assume a general architectural model based on a systolic-array accelerator akin to TPU [35], [49]. Since TANGRAMFP does not affect performance, we evaluate it using emulation to test its numerical accuracy and measure its power savings (in conjunction with the results from hardware synthesis (section V). Power savings are defined by the relative percentage of the modes that TANGRAMFP 1:5:5 dynamically selects in inference or training. More specifically, we implement TANGRAMFP 1:5:5 in C to create a Pytorch library that simulates the custom hardware that dynamically optimizes its configuration based on the actual values of the DNN as they are generated at runtime. We adapt PyTorch's layer implementations—such as conv2d, conv2d depthwise (utilized in compact neural networks like MobileNet and ShuffleNet), linear, and matmul (employed in Transformer models and large language models for multi-head attention mechanisms)—to replace standard multiplications with TANGRAMFP multiplications.

Our evaluation does not include MAC operations that have at least one zero operand. Thus, we compare against an optimized baseline which we assume efficiently eliminates ineffectual computation due to value sparsity (shown in Figure 1). In addition, we treat denormals as non-zero, by invoking TANGRAMFP's `Full Mode`. We are seeing less than 5% (about 3% on average) of all MACs having denormal inputs (X, Y, or Z), and we note that this is a peculiarity of FP16 — in other formats, e.g., bfloat16, TF32, denormals either do not exist or are exceedingly rare [50]. Finally, while we contrast TANGRAMFP to FPRaker in terms of error in section IV, we were unable to synthesize a working model of FPRaker from the description in [7] as many details are missing. In addition, FPRaker's benefit in performance and hence in energy efficiency comes in part from value sparsity which we consider as an orthogonal technique that exists in the baseline.

B. Workloads

TABLE IV: Networks

Network	Category	Size	Input
MobileNetV2 [61]	Image Classifier	4.0M	ImageNet1K [59]
MobileNetV3_large [61]	Image Classifier	5.4M	ImageNet1K
Regnet_x_800mf [54]	NN DSE	7.2M	ImageNet1L [59]
Regnet_y_800mf [54]	NN DSE	6.4M	ImageNet1L
ResNet18 [31]	Image Classifier	11.7M	ImageNet1K
ResNet50 [31]	Image Classifier	25.6M	ImageNet1K
ResNeXt101-32x4d [76]	Image Classifier	44.5M	ImageNet1K
Stable Diffusion [57]	Diffusion model	v1-4	Text Prompt
Llama2 [69]	LLM	7B	Text Prompt

Table IV details the DNNs used to evaluate TANGRAMFP. Our test suite includes five image classification neural networks, two neural-network design space exploration (NN DSE) models, one large-language model (LLM), and one Diffusion model. We utilize post-trained weights for MobileNetV2 to ResNeXt101 sourced from the official PyTorch website. For Stable Diffusion, we use version v1-4 available on Hugging Face. For Llama2, we employ the 7B model provided by Meta.TANGRAMFP is tested during both training and inference for most networks, but only during inference for Llama2 and Stable Diffusion due to their large sizes. We sample MAC operations for each network using a single batch for both inference and training. Moreover, when running Stable Diffusion, we sample MAC mode statistics for generating a 64×64 image with PMLS sampling. This approach is necessary due to the extensive memory requirements of generating larger images, such as 512×512.

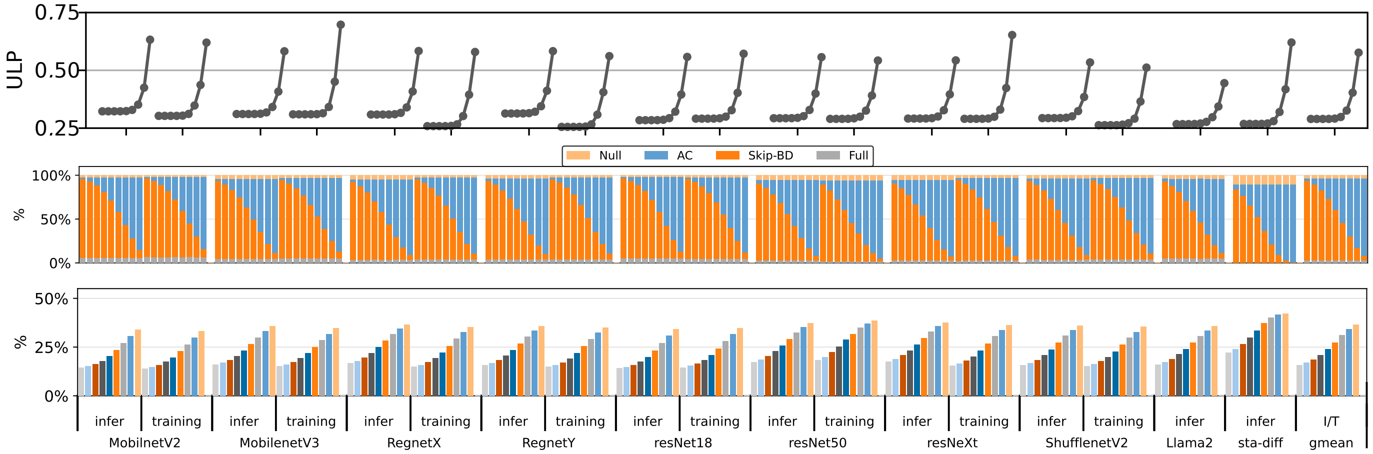


Fig. 13: Error (top), Mode Usage (middle), and Power Savings (bottom) under threshold 10, 9, ..., 2 (left to right).

To minimize individual alignment errors in a manner akin to the approach used in FPRaker [7], we implement a group MAC architecture. This setup processes 8 multiplications concurrently, followed by a collective accumulation using an adder tree. To enhance computational accuracy, we compare the sums of the exponents from each of the 8 multiplicand pairs with that of the accumulator. The largest resulting exponent then determines the operational mode for each MAC unit.

C. Results

Figure 13 shows the results for ULP error (top graph), mode usage (middle graph), and resulting power savings. For each DNN, we show nine sets of results for nine different *threshold values* that control the switch between the `Skip-BD Mode` and the `AC Mode` (see subsection IV-C). More specifically, if the exponent difference between the product XY and the accumulator Z is $s = e_z - (e_x + e_y)$, then

Condition	Mode
$s \leq 0$	`Full Mode`
$s \in 1..\text{threshold} - 1$	`Skip-BD Mode`
$s \in \text{threshold}..11$	`AC Mode`
$s > 11$	`NULL Mode`

TABLE V: Threshold and mode selection.

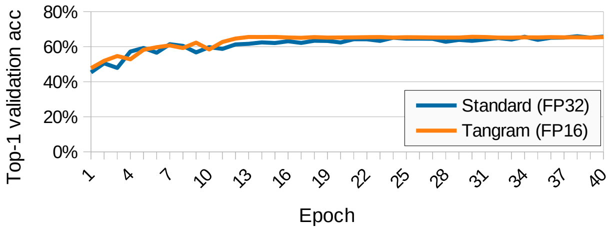


Fig. 14: Top-1 Validation accuracy: resnet18/CIFAR10

a) Modes versus Error: Figure 13, middle graph, reports the mode statistics (per DNN) as a function of the threshold value (from 10 down to 2, left to right). Over different benchmarks, we see that regardless of the threshold, the average use of `Full Mode` is ∼2.74%, and that of `Null Mode` is ∼3.65%. The average use of `Skip-BD Mode` is 91.05% for threshold 10, which decreases to 43.16% for threshold 5, and further decreases to 5.17% when threshold is 2. In contrast, the use of `AC Mode` increases from 2.62% when threshold is 10, to 50.39% when threshold is 5, and to 88.43% when threshold is 2. Decreasing the threshold for the transition from `Skip-BD Mode` to `AC Mode`, we see a dramatic increase in the use of `AC Mode`.

At the same time, Figure 13, top graph, plots the mean error in ULP (per DNN) that we see for the corresponding threshold. As the threshold is changed from 10 to 2, the error rises slowly at the beginning (0.29 ULP at threshold 10 to 6) and shoots up at around a threshold of five (0.30 ULP at threshold 5, but quickly increased to 0.58 ULP at threshold 2). In general, the mean error for TANGRAMFP stays within 0.30 ULP (for the thresholds 10 down to 5) which is close to the baseline mean error of 0.25 ULP for the standard FP16 (fused) MAC (recall that the error bound for the standard is 0.5 ULP). Note that, unless we cross a threshold of four in TANGRAMFP, its mean error is significantly less than 0.5 ULP, and is close to the mean error of the IEEE-754 standard FMAC [3]. To further test the numerical stability of TANGRAMFP, we trained ResNet18 on CIFAR-10 for 40 epochs. It is important to note that the standard training of ResNet18 on CIFAR-10 utilizes FP32 accuracy, while TANGRAMFP trains the network with FP16. We observed that for Top-1 validation accuracy, TANGRAMFP shows a similar convergence to the standard FP32 FMAC (see Figure 14). Given that we have established in section IV that the error bound of TANGRAMFP can be brought close to the standard (by choosing appropriate thresholds), these results demonstrate remarkably robust numerical stability, especially for training and for important models such Transformer and Diffusion, for a wide range of the `Skip-BD` to `AC` threshold.

b) Energy Estimation: In conjunction with the power synthesis results of section V, the bottom graph of Figure 13 shows the runtime dynamic power savings for the corresponding DNN/thresholds per MAC unit. Stable Diffusion achieves the biggest savings (over 42.27% for a threshold of 2), while the overall average tops at 36.57% savings (threshold 2). For the more conservative threshold 5 that gives a mean ULP error close to the mean of the standard, the savings reach 27.44%.

VII. Conclusion

In this work we present TangramFP, an innovative tunable floating-point MAC design intended to replace existing MAC units without altering the overall accelerator architecture. TangramFP aims to reduce energy consumption by dynamically adjusting computations to eliminate unnecessary operations. It achieves this through a novel mechanism that selectively generates partial products by splitting input operands and enabling or disabling different areas of the multiplier array. We demonstrate that TangramFP, in its power-efficient operation, offers better precision compared to other approaches like bit-serial truncated multiplication. TangramFP matches standard MAC designs in precision, area, and latency, while being able to save up to 36.57% dynamic power, without significantly impacting error rates.

References

[1] "The bfloat16 numerical format," https://cloud.google.com/tpu/docs/bfloat16.

[2] "Graphcore ipu-pod64 reference design datasheet, 2022." docs.graphcore.ai/projects/ipu-pod64-datasheet/en/latest/.

[3] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[4] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell *et al.*, "Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 145–158.

[5] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th annual IEEE/ACM international symposium on microarchitecture*, 2017, pp. 382–394.

[6] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[7] O. M. Awad, M. Mahmoud, I. Edo, A. H. Zadeh, C. Bannon, A. Jayarajan, G. Pekhimenko, and A. Moshovos, "Fpraker: A processing element for accelerating neural network training," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 857–869.

[8] K. C. Bickerstaff, M. J. Schulte, and E. E. Swartzlander, "Parallel reduced area multipliers," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 9, pp. 181–191, 1995.

[9] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on fpga," in *2017 IEEE International symposium on circuits and systems (ISCAS)*. IEEE, 2017, pp. 1–4.

[10] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[11] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[12] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.

[13] J. Choquette and W. Gandhi, "Nvidia a100 gpu: Performance amp; innovation for gpu computing," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2020, pp. 1–43. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HCS49909.2020.9220622

[14] M. Cornea, "Ulps and relative error," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, 2017, pp. 90–97.

[15] L. Dadda, "Some schemes for parallel multipliers," in *Computer Arithmetic, Volume I*. World Scientific, 2015.

[16] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner *et al.*, "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point," *Advances in neural information processing systems*, vol. 33, pp. 10 271–10 281, 2020.

[17] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training dnns with hybrid block floating point," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[18] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPR 2011 workshops*. IEEE, 2011, pp. 109–116.

[19] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, pp. 13–es, 2007.

[20] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on computers*, vol. 60, no. 7, pp. 913–922, 2010.

[21] V. Garofalo, N. Petra, D. De Caro, A. G. Strollo, and E. Napoli, "Low error truncated multipliers for dsp applications," in *2008 15th IEEE international conference on electronics, circuits and systems*. IEEE, 2008, pp. 29–32.

[22] F.-Y. Gu, C. Lin, and J.-W. Lin, "A low-power and high-accuracy approximate multiplier with reconfigurable truncation," *IEEE Access*, vol. 10, pp. 60 447–60 458, 2022.

[23] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto customized hardware," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2016, pp. 24–29.

[24] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[25] L. Gwennap, "Groq rocks neural networks," *Microprocessor Report, Tech. Rep., jan*, 2020.

[26] A. Habibi and P. A. Wintz, "Fast multipliers," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 153–157, 1970.

[27] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[28] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[29] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[30] M. Hasan and B. Ray, "Tolerance of deep neural network against the bit error rate of nand flash memory," in *2019 IEEE International Reliability Physics Symposium (IRPS)*, 2019, pp. 1–4.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[32] A. Hock, "Introducing the cerebras cs-1, the industry's fastest artificial intelligence computer-cerebras," 2019.

[33] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021.

[34] E. Hokenek, R. K. Montoye, and P. W. Cook, "Second-generation risc floating point with multiply-add fused," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1207–1213, 1990.

[35] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles *et al.*, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.

[36] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *ieee Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[37] 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, pp. 1–12.

[38] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[39] S. Khoram and J. Li, "Adaptive quantization of neural networks," in *International Conference on Learning Representations*, 2018.

[40] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," *Advances in neural information processing systems*, vol. 30, 2017.

[41] Y.-C. Lo and R.-S. Liu, "Bucket getter: A bucket-based processing engine for low-bit block floating point (bfp) dnns," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1002–1015.

[42] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, pp. 774–787, 2020.

[43] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu *et al.*, "Fp8 formats for deep learning," *arXiv preprint arXiv:2209.05433*, 2022.

[44] B. Moons and M. Verhelst, "A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets," in *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. IEEE, 2016, pp. 1–2.

[45] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 107–116.

[46] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 258–261.

[47] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–7.

[48] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[49] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[50] V. Popescu, A. Venigalla, D. Wu, and R. Schreiber, "Representation range needs for 16-bit neural network training," *arXiv preprint arXiv:2103.15940*, 2021.

[51] E. Quinnell, E. E. Swartzlander, and C. Lemonds, "Floating-point fused multiply-add architectures," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*. IEEE, 2007, pp. 331–337.

[52] E. Quinnell, E. E. Swartzlander, and C. Lemonds, "Bridge floating-point fused multiply-add design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 12, pp. 1727–1731, 2008.

[53] V. Radhakrishnan, "Bit error tolerance of deep neural network accelerators," Ph.D. dissertation, 2020.

[54] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," in *CVPR*, 2020.

[55] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," *ACM Computer Architecture News*, vol. 44, no. 3, pp. 267–278, 2016.

[56] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *2020 IEEE high performance extreme computing conference (HPEC)*, 2020, pp. 1–12.

[57] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 10 684–10 695.

[58] S. M. Rump, "Error estimation of floating-point summation and dot product," *BIT Numerical Mathematics*, vol. 52, pp. 201–220, 2012.

[59] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, pp. 211–252, 2015.

[60] J. Ryckaert, P. Schuddinck, P. Weckx, G. Bouche, B. Vincent, J. Smith, Y. Sherazi, A. Mallik, H. Mertens, S. Demuynck, T. H. Bao, A. Veloso, N. Horiguchi, A. Mocuta, D. Mocuta, and J. Boemmels, "The complementary fet (cfet) for cmos scaling beyond n3," in *2018 IEEE Symposium on VLSI Technology*, 2018, pp. 141–142.

[61] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[62] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 145–150.

[63] M. J. Schulte and E. E. Swartzlander, "Truncated multiplication with correction constant [for dsp]," in *Proceedings of IEEE workshop on VLSI signal processing*. IEEE, 1993, pp. 388–396.

[64] E. M. Schwarz, M. Schmookler, and S. D. Trong, "Fpu implementations with denormalized numbers," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 825–836, 2005.

[65] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[66] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2d and 3d cnns on fpga," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 97–106.

[67] I. Soloveychik, I. Lyubomirsky, X. Wang, and S. Bhoja, "Block format error bounds and optimal block size selection," *arXiv preprint arXiv:2210.05470*, 2022.

[68] J. E. Stine and O. M. Duverne, "Variations on truncated multiplication," in *Euromicro Symposium on Digital System Design, 2003. Proceedings.* IEEE, 2003, pp. 112–119.

[69] H. Touvron, L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. M. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. S. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. M. Kloumann, A. V. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *ArXiv*, vol. abs/2307.09288, 2023.

[70] W. J. Townsend, E. E. Swartzlander Jr, and J. A. Abraham, "A comparison of dadda and wallace multiplier delays," in *Advanced signal processing algorithms, architectures, and implementations XIII*, vol. 5205. SPIE, 2003, pp. 552–560.

[71] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/pdf/1706.03762.pdf

[72] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 40–47.

[73] J. S. Vitter, "Random sampling with a reservoir," vol. 11, no. 1, p. 37–57, 1985.

[74] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on electronic Computers*, no. 1, pp. 14–17, 1964.

[75] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking tpu, gpu, and cpu platforms for deep learning," *arXiv preprint arXiv:1907.10701*, 2019.

[76] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5987–5995.

[77] S. Q. Zhang, B. McDanel, and H. Kung, "Fast: Dnn training under variable precision block floating point with stochastic rounding," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 846–860.