XtraMAC: An Efficient MAC Architecture for Mixed-Precision LLM Inference on FPGA

Feng Yu[*], Hongshi Tan[*], Yao Chen[†,‡], Weng-Fai Wong[*], Bingsheng He[*,‡]

[*]*School of Computing, National University of Singapore*, Singapore

[†]*School of Computer Science and Technology, Huazhong University of Science and Technology*, Wuhan, China

{yuf, hongshi}@u.nus.edu, {dcswwf, dcsheb}@nus.edu.sg, chenyao_cs@hust.edu.cn

arXiv:2605.06052v1 [cs.AR] 7 May 2026

***Abstract*—The widespread adoption of mixed-precision quantization in large language models (LLMs) has created demand for hardware that can efficiently perform multiply–accumulate (MAC) operations across mixed datatypes and switch datatypes at runtime. Existing FPGA-based MAC solutions fall short due to limitations in fixed-datatype design, inefficient spatial or temporal resource sharing, and poor support for mixed-precision execution. These limitations collectively lead to under-utilization of DSP resources, limiting achievable parallelism and throughput. In this work, we present XtraMAC, a novel MAC architecture that unifies integer, floating-point, and mixed-precision operations within a single, datatype-adaptive microarchitecture. XtraMAC decomposes all supported MAC formats into a shared integer mantissa product with lightweight sign and exponent handling, enabling dynamic operand packing and efficient DSP resource sharing with constant latency and initiation interval of one across all datatypes. Evaluated on an AMD Xilinx U55c FPGA, XtraMAC achieves 1.4–2.0× higher compute density, reduces per-operation LUT, FF, and DSP consumption by 27–51%, and delivers up to 1.9× greater energy efficiency and 1.2× speedup on representative mixed-precision LLM workloads. The implementation of XtraMAC is open-sourced at https://github.com/Xtra-Computing/XtraMAC.**

***Index Terms*—MAC architecture, mixed-precision arithmetic, runtime datatype switching, DSP packing, FPGA.**

I. Introduction

Multiply–accumulate (MAC) operations are the fundamental building blocks for a broad spectrum of computational tasks, ranging from digital signal processing to deep learning. In recent years, large language models (LLMs) have emerged as the most prominent and demanding consumers of MAC computation, as their inference workloads are dominated by matrix multiplications involving billions of parameters. This explosive growth in MAC demand has catalyzed the adoption of low-precision quantization techniques to reduce both memory footprint and computational energy. Quantization methods [6], [11], [13], [22], [39], [44] target different model components (weights, activations, MoE experts) with mixed numeric formats and assign different datatypes across layers based on their quantization sensitivity, e.g., lower-bit integers for compute-intensive layers and floating-point formats for numerically sensitive ones [13], [22], [39]. As summarized in Table I, this results in diverse MAC datatype combinations across different quantization schemes and model components.

TABLE I
DIVERSITY OF MACs ACROSS LLM QUANTIZATION SCHEMES.

Category	Examples	Projection / FFN MACs	Attention MACs
Weight-only quant.	AWQ, GPTQ, SpQR	INT × FP + FP → FP	FP × FP + FP → FP
Weight–act quant.	SmoothQuant, Atom	INT × INT + INT → INT	FP × FP + FP → FP
Native LLMs	GPT-oss-20b, 120b	MXFP4/BF16[*]× FP + FP → FP	FP × FP + FP → FP

[*] In GPT-oss models, the MoE blocks use MXFP4, others use BF16 datatypes.

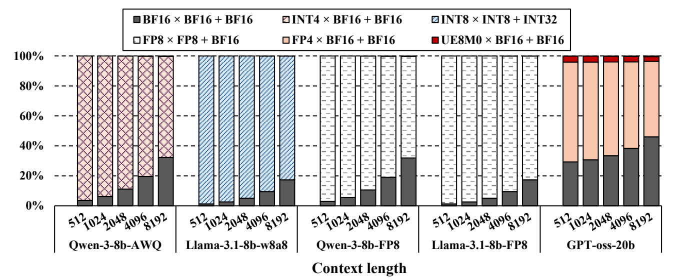

Fig. 1. Distribution of MAC operations during the decode stage for various quantized LLM checkpoints in Table VI across different context lengths. Each segment represents a unique MAC configuration; non-MAC operations are omitted as they account for less than 1% of total operations.

However, this proliferation of mixed-precision quantization introduces a new challenge for hardware design. We define a MAC operation as P = A×B + C, and identify two distinct computational patterns that arise in practice. The first is **mixed-precision MAC**, in which the multiplicands A and B are represented in heterogeneous numeric formats or bitwidths (e.g., INT4×BF16). The second is **runtime datatype switching**, in which a single hardware unit must alternate among distinct MAC datatypes as execution traverses different model components. For example, a single forward pass may transition from INT4×BF16 in projection layers to BF16×BF16 in attention layers. As shown in Figure 1, Qwen-3-8B-AWQ executes over 68% of its decode-stage MACs in INT4×BF16 for projection layers while its attention layers retain BF16×BF16, exemplifying the coexistence of both patterns within a single model. This dynamic heterogeneity demands hardware that can natively support both patterns without performance degradation.

Field-programmable gate arrays (FPGAs) have emerged as a promising platform for addressing these challenges, owing to their ability to implement customized compute pipelines with

[‡]Yao Chen and Bingsheng He are the corresponding authors.

TABLE II

SURVEY OF FPGA-BASED MAC DESIGNS FOR MIXED PRECISION (MP) AND RUNTIME DATATYPE SWITCHING (RDS) SUPPORT.

Name	Year	DataType(A)	DataType(B)	DataType(C)	DataType(P)	MP	RDS	Note
FINN [35]	2017	Binary	Binary	Integer	Integer	✗	✗	Fixed architecture for binary neural networks
FP-BNN [21]	2017	Binary	Binary	Integer	Integer	✗	✗	Fixed architecture for binary neural networks
DNNBuilder [43]	2018	Fixed-point	Fixed-point	Fixed-point	Fixed-point	✗	✗	Precision set at synthesis time
BISMO [36]*	2019	Integer	Integer	Integer	Integer	✓	✓	Runtime switchable integer bit width
Xilinx FP Operator [1]†	2020	**Floating-point**	**Floating-point**	**Floating-point**	**Floating-point**	✗	✗	Format chosen at synthesis time
Triple MAC [20]	2021	Fixed-point	Fixed-point	Fixed-point	Fixed-point	✗	✗	Static precision; width chosen at design time
TATAA [38]‡	2025	INT8 or BF16	INT8 or BF16	INT32 or BF16	INT32 or BF16	✗	✓	Runtime switchable between INT8 and BF16
Ours	2025	**Integer, floating-point**	**Integer, floating-point**	**Integer, floating-point**	**Integer, floating-point**	✓	✓	Runtime switchable across all datatypes

* BISMO supports arbitrary integer bit-width selection at runtime, but the maximum permissible precision is fixed at synthesis time due to FPGA resource constraints.
† Operands A, B, C and output P must share the same floating point datatype. ‡ Operands A, B, C and output P must share the same datatype, either all INT8 or all BF16.

fine-grained, bit-level control over arithmetic and dataflow. In particular, the datapaths leading to the primary computational units in FPGAs, the DSP cores, can be tailored to accommodate diverse datatype requirements. Nevertheless, existing solutions for mixed-precision MAC support on FPGAs remain suboptimal. Conventional approaches to leveraging DSPs in FPGAs for mixed-precision computation and runtime datatype switching can be broadly categorized as follows:

- **Operand upcasting**, which promotes low-precision operands to match a fixed high-precision MAC unit, resulting in significant waste of the DSP bit space;
- **Spatial replication or temporal sharing**, which instantiates multiple datatype-specific datapaths or reuses a single datapath across cycles to support runtime datatype switching, leading to low effective DSP utilization, as only a subset of resources is active at any moment.

For example, the AMD Xilinx Floating-Point Operator [1] achieves average **32.4%** DSP bit-utilization efficiency when executing low-precision workloads, as all operands must be upcast into a fixed high-precision floating-point format. For runtime datatype switching, the spatial-replication based design duplicates multiple datatype-specific datapaths using the same Floating-Point Operator IP [1] further reduces efficiency: only one datapath is active at a time while the others remain idle, resulting in an effective DSP utilization of average **26.7%**. Temporal-sharing architectures such as TATAA [38] decompose a BF16 MAC into four sequential INT8 operations, yielding 71.1% utilization for INT8-based MAC computation but only **8.9%** effective utilization for BF16-based MAC computation due to the multi-cycle decomposition.

The root cause of these inefficiencies is a theoretical disconnect between the processing patterns and the fixed resource granularity of FPGA DSP slices. While DSP packing has been shown to substantially boost utilization for integer-only workloads [8], [24], current solutions lack the bit-level analysis needed to extend this technique to current workloads. As a result, each DSP delivers throughput far below its arithmetic ceiling, and this shortfall widens as LLMs increasingly mix precisions and switch datatypes within a single forward pass.

In this work, we bridge this gap by introducing a resource-compact MAC architecture that is natively aware of mixed precision and supports runtime datatype switching within a unified datapath. Our key contributions are:

- We present a unified formulation showing that the multiplication component of integer, floating-point, and mixed-precision MACs can all be decomposed into an integer mantissa product with lightweight sign and exponent handling.
- We propose XtraMAC, a datatype-adaptive MAC architecture that decouples format interpretation from arithmetic execution, unifying integer, floating-point, and mixed-precision MAC operations within a single shared datapath.
- We develop a DSP-centric design principle that exploits dynamic bit mapping and multi-lane packing to maximize multiplier utilization, and adopt a fixed four-stage pipeline to sustain a constant latency and initiation interval of one across all supported datatypes.
- We demonstrate that XtraMAC improves compute density by 1.4–2.0× and reduces LUT, FF, DSP usage by 30.0%, 47.9%, 50.0% compared with state-of-the-art FPGA baselines, achieving 1.2× lower GEMV latency and 1.9× higher energy efficiency compared with GPU baseline.

II. BACKGROUND AND DESIGN MOTIVATION

Motivated by the mixed-precision and runtime datatype-switching demands of LLM workloads, we first formalize DSP utilization under widely adopted conventions, and then review prevailing MAC microarchitectures in FPGA-based solutions, with particular emphasis on their DSP utilization.

A. DSP Utilization

In modern FPGAs, hardened DSP slices center on a dedicated multiplier that serves as the primary arithmetic resource and performs integer multiplication. The associated pre-addition, post-addition, and pipeline logic is structurally simpler and incurs substantially lower area and power overheads compared with the multiplier itself [4], [40]. Therefore, following the operand-bit-based utilization model in [9], we quantify DSP utilization based on how effectively the multiplier hardware is exercised. Let w_a and w_b denote the effective bit-widths of the multiplicands involved in an operation, we define the DSP utilization as

$$U_{\text{DSP}} = (w_a + w_b)/W_{\text{mul}},$$

where W_{mul} is the sum of the two input-port widths of the DSP multiplier. In this paper, we target the DSP48E2 primitive widely deployed in modern Xilinx FPGAs, whose multiplier accepts a 27-bit A-port operand and an 18-bit B-port operand, giving $W_{\text{mul}} = 45$ bits [4].

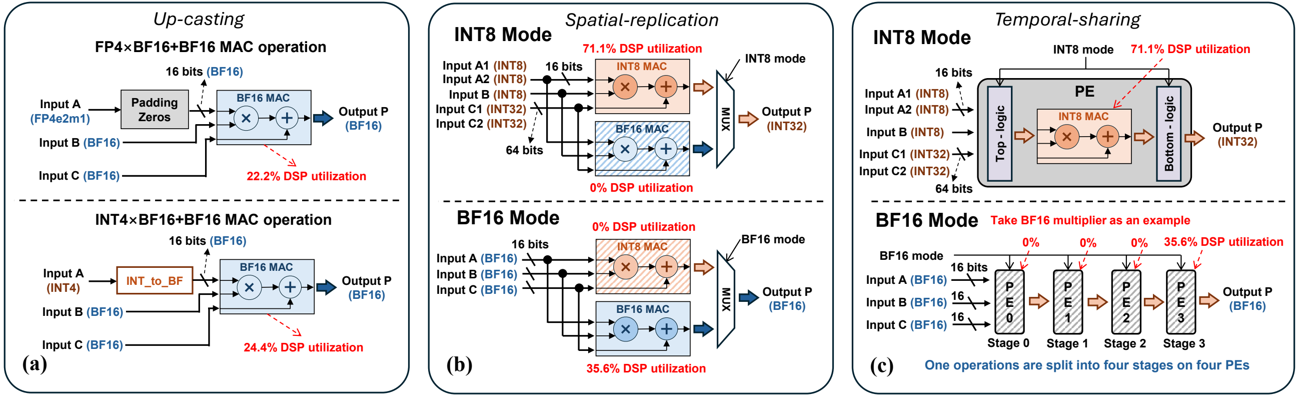

Fig. 2. Overview of existing FPGA-based MAC architectures supporting mixed precision and runtime datatype switching. (a) Upcasting-based method using FP Operator [1] for mixed precision. (b) Spatial replication for multi-datatype support [1]. (c) Temporal-sharing based multi-datatype support (TATAA [38]).

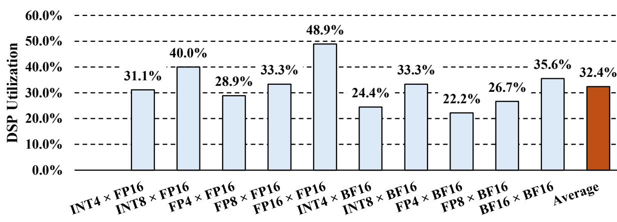

Fig. 3. DSP utilization of the upcasting-based design under different mixed-precision datatype combinations (FP8 = E4M3, FP4 = E2M1).

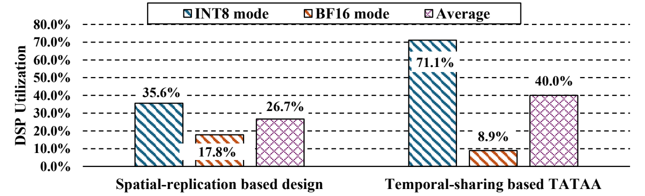

Fig. 4. DSP utilization comparison on existing FPGA-based MAC architectures supporting runtime datatype switching in Fig.2.

B. Current MAC designs in FPGA-based solutions

MAC designs in FPGA-based solutions have evolved to support a variety of numerical formats, ranging from binary, integer, fixed-precision arithmetic to full-precision floating-point. As summarized in Table II, these designs demonstrate the flexibility of reconfigurable logic for arithmetic specialization but reveal persistent limitations in mixed-precision support and runtime datatype switching.

1) Lack of Efficient Mixed-Precision Support: Early FPGA-based accelerators, such as FINN [35], FP-BNN [21], and DNNBuilder [43], target fixed integer or fixed-point formats with precision determined at synthesis time. Consequently, most FPGA MAC architectures are optimized for fixed or uniform operand types and provide limited support for mixed-precision computation. In practice, mixed-precision operations are typically handled by upcasting low-precision operands to the highest supported precision and executing them on a high-precision MAC datapath. As illustrated in Fig. 2(a), and exemplified by the AMD Xilinx Floating-Point Operator [1], low-precision operands are padded or promoted to match the highest supported precision, and all operations are executed on a fixed high-precision datapath. This design results in significant hardware inefficiency: a large portion of the DSP bit capacity remains unused, as quantified in Fig. 3.

2) Inefficient DSP Utilization and Limitations in Runtime Datatype Switching: Recent designs supporting runtime datatype switching on FPGAs predominantly adopt one of two microarchitectural strategies: spatial replication or temporal sharing. Representative designs are illustrated in Fig. 2(b) and (c). While each approach offers certain advantages, both introduce inherent limitations and inefficiencies in DSP utilization, particularly when accommodating the diverse numeric requirements of LLM workloads.

Spatial Replication. As shown in Fig. 2(b), designs such as those described in [5] address runtime datatype switching by duplicating MAC units for each datatype and employing a multiplexer (MUX) to select the active datapath at runtime. For example, an INT8/BF16-configurable instance instantiates both INT8 and BF16 MACs, toggling between them with a control signal. While this method enables zero-latency switching and avoids pipeline bubbles, it incurs substantial hardware overhead as the number of supported formats increases. Idle MAC units for inactive datatype remain unused, leading to poor resource efficiency and failing to leverage the concurrency benefits inherent to low-precision computation. In Fig. 4, the average DSP utilization drops to 26.7%.

Temporal Sharing. As shown in Fig. 2(c), a more sophisticated approach in TATAA [38], which reuses integer MAC units for both INT8 and BF16 computation by decomposing each BF16 operation into a sequence of INT8 micro-operations executed over multiple cycles. This strategy avoids the need for separate BF16 logic, reducing area overhead. However, it comes at the expense of spatial parallelism and throughput: each BF16 operation monopolizes four processing elements (PEs) and pipeline stages, effectively capping peak BF16 throughput at one quarter of that achievable for INT8, given the same hardware footprint. As quantified in Fig. 4, the DSP

utilization drops to 8.9% when supporting BF16-based MAC, which significantly reduces the hardware effectiveness.

These pronounced inefficiencies in DSP utilization underscore a fundamental microarchitectural problem, one that becomes increasingly problematic as mixed-precision floating-point computation and runtime datatype switching grow prevalent in state-of-the-art LLM inference workloads.

C. Our Observations

The above quantitative results point to a common root cause: a mismatch between the bit-level processing patterns of low-precision MAC operations and the fixed resource granularity of FPGA DSP slices. Existing designs either upcast low-bit operands to a wide format or serialize high-precision operations over a narrow low-precision core. In both cases, **the DSP multiplier is used as an opaque**, single-lane primitive, rather than as a bit-space that can be systematically partitioned and shared across datatypes. Furthermore, both approaches **treat datatype switching as a coarse control problem over whole datapaths**, instead of a fine-grained organization of workload allocation within the DSP bit-space.

These observations suggest that substantial further gains are unlikely to be achieved by incremental datapath tweaks alone. Instead, we rethink mixed-precision MAC support from a processing-pattern perspective: how low- and mixed-precision INT/FP multiplications decompose into bit-level operations that can be allocated to the DSP multiplier based on its given processing characteristics, and how the DSP utilization could be improved with proper parallelism supported by the microarchitecture of the MAC. The next section develops this processing-pattern formulation, which forms the theoretical basis for the XtraMAC architecture and its ability to achieve high DSP utilization and fine-grained runtime datatype switch.

III. Processing Pattern Formulation

To fully exploit the sharing and performance potential of mixed-precision MAC architectures, we first analyze how low- and mixed-precision MAC operations map onto the native multipliers and accumulators of DSP slices in FPGAs. Unlike conventional fixed-precision designs, these MAC operations introduce heterogeneous operand widths that misalign with DSP bit-granularity, giving rise to distinct sub-DSP packing and lane parallelism patterns. Hence, a resource-compact MAC architecture must be grounded in a systematic characterization of these bit-level processing patterns. We begin with normalized, non-exceptional values to establish the core formulation, and extend the analysis in a later subsection to cover special values and rounding. Under these conventions, XtraMAC produces bit-exact results matching NVIDIA A100/H100 Tensor Cores [27], [28] and the official AMD Floating-Point Operator [1] across all supported datatypes.

A. Shared Multiplication Datapath across Different Datatypes

For a floating-point operand $x = s_x \cdot 2^{e_x} \cdot m_x$ in IEEE format, we assume that the mantissa m_x includes the implicit leading-one bit (i.e., $m_x \in [1, 2)$ for normalized values). The product of two such values x and y naturally decomposes as

$$x \cdot y = (s_x \oplus s_y) \cdot 2^{\,e_x+e_y-\text{bias}} \cdot (m_x \cdot m_y), \tag{1}$$

when adopting the MAC constructed with FPGA DSP slice, the DSP computes only the mantissa product $m_x m_y$, while sign and exponent are processed separately [1], [4]. After multiplication, the product mantissa is normalized by a leading-zero count (LZC):

$$\Delta = \text{LZC}(m_x m_y), \quad m^{\text{norm}} = (m_x m_y) \ll \Delta, \tag{2}$$

$$e^{\text{out}} = e_x + e_y - \text{bias} - \Delta. \tag{3}$$

For mixed-precision INT×FP computation, the integer operand a is interpreted as a two's complement value and decomposed into sign and magnitude:

$$a = s_a \cdot m_a,$$

where m_a is treated as an integer mantissa after sign extraction. Since an integer carries no exponent encoding, we assign it a logical unbiased exponent of zero, which corresponds to a biased exponent value equal to the bias of the floating-point output format (e.g., 127 for FP32 and BF16, or 15 for FP16). The floating-point operand y is represented same as the x, with m_y including the leading-one bit. Their product becomes

$$a \cdot y = (s_a \oplus s_y) \cdot 2^{\,e_y-\text{bias}} \cdot (m_a \cdot m_y), \tag{4}$$

so that the DSP multiplier again computes only the mantissa product $m_a m_y$. The exponent path forwards e_y, subtracts the format bias, and applies the normalization shift:

$$\Delta = \text{LZC}(m_a m_y), \quad m^{\text{norm}} = (m_a m_y) \ll \Delta, \tag{5}$$

$$e^{\text{out}} = e_y - \text{bias} - \Delta. \tag{6}$$

Eqs. (1) and (4) show that both FP×FP and INT×FP multiplication share the same core operation: the DSP computes an integer mantissa product, while sign and exponent are handled outside the DSP. This indicates that a common multiplication datapath can be shared across different datatypes. The only differences arise in how operands are mapped into (s, m, e) fields and how the exponent is updated after normalization. Hence, across all integer, floating-point, and mixed-precision formats, multiplication can be described as follows:

1) **Mapping:** extract or construct sign, mantissa, and exponent for each input operand of the DSP slice.
2) **DSP multiplication:** compute product $m_{\text{prod}} = m_A \cdot m_B$.
3) **Post-compute:** apply LZC-based normalization and update the exponent according to the operand formats.

This formulation shows that datatype-specific behavior is confined to lightweight mapping and post-compute logic, while the DSP slice consistently performs the same integer multiplication. Consequently, a single multiplier datapath can support FP×FP, INT×FP, and pure integer multiplication with minimal additional hardware.

B. Datatype-specific Accumulation

Unlike the multiplication stage, which naturally shares a unified integer–mantissa structure across all data types, addition behaves fundamentally differently due to incompatible

computation patterns arising from their distinct bit semantics. For integer addition, two's-complement integers x, y satisfy

$$x + y = (x \oplus y) \oplus \mathrm{carry}(x, y),$$

which maps directly to the FPGA's ripple-carry chain. A w_{int}-bit integer adder therefore exhibits linear resource cost:

$$C_{\text{int}}(w_{\text{int}}) \approx \alpha_{\text{int}}\, w_{\text{int}}, \tag{7}$$

where α_{int} denotes the LUT cost per bit imposed by the carry-chain fabric.

For floating-point addition, given $x = s_x 2^{e_x} m_x$ and $y = s_y 2^{e_y} m_y$, define the exponent gap

$$\Delta e = e_x - e_y.$$

Assuming $e_x \geq e_y$, the smaller mantissa must be right-shifted by an *arbitrary* distance determined at runtime:

$$m_s = m_x \pm \left(m_y \cdot 2^{-\Delta e}\right).$$

The resulting mantissa must then be normalized using LZC:

$$\Delta = \mathrm{LZC}(m_s), \qquad m_{\text{out}} = (m_s \ll \Delta), \qquad e_{\text{out}} = e_x - \Delta.$$

Because Δe may take any value in the exponent range, the alignment step must support a variable-distance right shift across the full mantissa width w_{fp}; similarly, normalization requires a variable-distance left shift. Implementing such shifts on FPGA requires a logarithmic barrel shifter. A classical barrel shifter for an w_{fp}-bit mantissa contains $\log_2 w_{\text{fp}}$ stages of w_{fp} multiplexers [32], giving

$$N_{\text{MUX}} = w_{\text{fp}} \log_2 w_{\text{fp}},$$

and the LUT cost scales superlinearly:

$$C_{\text{shifter}}(w_{\text{fp}}) \approx \beta_{\text{sh}}\, w_{\text{fp}} \log_2 w_{\text{fp}}, \tag{8}$$

where β_{sh} denotes the LUT cost per multiplexer stage. Prior work confirms that these alignment and normalization shifters dominate the LUT footprint of FPGA floating-point adders [12], [25].

Thus, these characteristics highlight the inherently datatype-specific nature of accumulation. Integer addition typically requires a wide bit width w_{int} (for example, 32 bits for INT4/INT8 accumulation [26], [28]) but avoids expensive shifting by relying on efficient carry-chain logic. Floating-point addition operates on narrower mantissas w_{fp} (for example, 10-bit for FP16, 7-bit for BF16) but incurs substantial cost due to alignment and normalization shifters. Consequently, a unified INT–FP adder would force integer additions to traverse the same alignment and normalization shifters as floating-point additions, even though integer addition fundamentally requires no shifting. This wastes shifter area, which must be sized for the wider integer width w_{int}, making it significantly less resource-efficient than maintaining separate adder paths.

C. Parallel Mixed-Precision MACs on a Single DSP Slice

In FPGA implementation, a DSP slice computes a single wide integer multiplication

$$P_{\text{DSP}} = A_{\text{DSP}} \cdot B_{\text{DSP}},$$

where A_{DSP} and B_{DSP} are bit vectors on the two input ports. Conventional fixed-precision designs place each operand in the least-significant bits, leaving most of the multiplier unused and resulting in low DSP bit utilization. However, as shown in Section III-A, all MAC datatypes ultimately reduce to integer mantissa multiplication after appropriate operand mapping. This enables multiple low-precision lanes to be packed into disjoint bit regions of the DSP inputs, thereby exploiting the full multiplier width [8], [14], [24].

To perform this packing, the mapping stage assigns each mantissa or integer magnitude to a non-overlapping bit range:

$$A_{\text{DSP}} = \sum_i (a_i \ll s_i), \qquad B_{\text{DSP}} = \sum_j (b_j \ll t_j), \tag{9}$$

where s_i and t_j are per-lane shift offsets selected to avoid cross-lane interference. These offsets depend on the datatype combination (INT, FP, or INT×FP), but the DSP always receives well-formed integer operands regardless of precision.

Once packed, the DSP performs one wide multiplication, and all lane products appear at predetermined bit positions:

$$P_{\text{DSP}} = \sum_{i,j} (a_i b_j) \ll (s_i + t_j). \tag{10}$$

In the post-compute stage, each lane product is extracted using a fixed shift-and-mask operation:

$$P_{i,j} = (P_{\text{DSP}} \gg (s_i + t_j)) \;\&\; (2^S - 1), \tag{11}$$

where S is the bit stride allocated per lane. Let W_{lane} denote the maximum bit-width of any supported product $|a_i \cdot b_j|$. To guarantee no overlap, the stride must satisfy

$$S \geq W_{\text{lane}} + G,$$

where G is a small guard margin (typically one bit) used to absorb carries.

By incorporating lane packing into the mapping stage and integrating product extraction into the post-compute stage, the unified multiplication datapath of Section III-A naturally extends to support parallel mixed-precision MACs. As a result, multiple INT, FP, or INT×FP operations can be executed in parallel within a single DSP slice by sharing the same datapath across different datatypes.

The parallelism of shared DSP multiplier is constrained by DSP input widths. For a given datatype, once its per-lane stride S is determined, the maximum achievable parallelism is

$$\text{Parallelism} \leq \min\left(\left\lfloor \frac{L_A}{S} \right\rfloor, \left\lfloor \frac{L_B}{S} \right\rfloor\right), \tag{12}$$

where L_A = 27 and L_B = 18 are the DSP48E2 input widths. Importantly, different datatypes (e.g., INT4×BF16, FP8×FP8, INT8×INT8) have different operand precisions and thus correspond to different stride values S. Once the stride for a datatype is fixed, the achievable parallelism for that datatype follows directly from Eq. (12) and depends solely on L_A, L_B.

In summary, by reducing all numerical formats to products and packing these lanes into the DSP inputs at bit-precise offsets, XtraMAC enables a single DSP multiplier to be shared across integer, floating-point, and mixed-precision MACs. The polynomial structure of packed multiplication in Eq. (10) preserves strict lane isolation, while per-lane normalization and exponent reconstruction restore correct floating point and

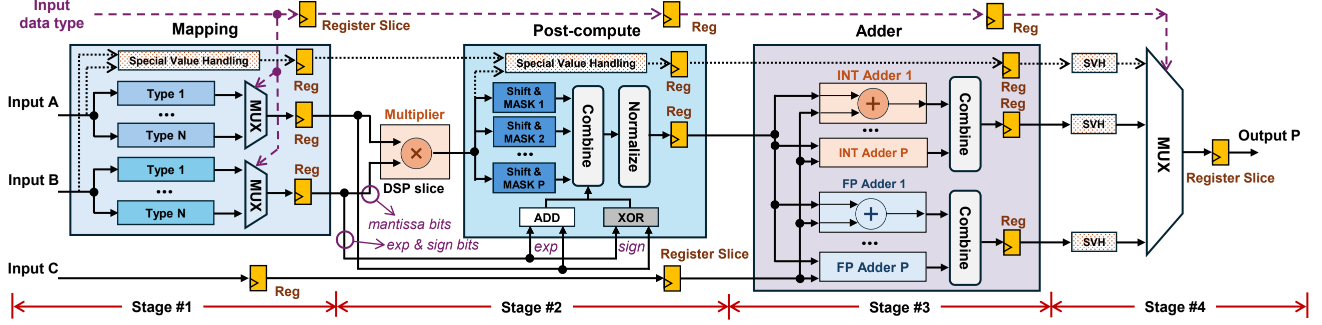

Fig. 5. Overview of the XtraMAC architecture supporting N datatype combinations and up to P-way parallelism. (SVH: special value handling).

mixed-precision semantics. The bit-level control forms the basis of our mixed-precision support and allows multiple low-precision MAC lanes to execute in parallel.

D. Special Value Handling

1) Numerical Coverage: XtraMAC adopts flush-to-zero (FTZ) and denormals-are-zero (DAZ) semantics throughout the computation datapath, consistent with the numerical conventions widely employed in modern floating-point compute hardware, such as NVIDIA A100/H100 Tensor Cores [27], [28] and the AMD Xilinx Floating-Point Operator [1]. Subnormal inputs are treated as zero upon ingestion and outputs falling below the minimum value are flushed to zero. NaN inputs propagate as canonical quiet NaN (qNaN), infinity is preserved with its sign, and conflicting cases such as $\infty \times 0$ and $+\infty + (-\infty)$ resolve to qNaN. For formats that do not encode infinity, all-ones exponent encodings are treated as NaN. Integer-to-floating-point conversion is exact, and floating-point accumulation applies round-to-nearest-even (RN-even) throughout.

2) Pipeline-Invariant Exception Handling: A key design requirement is that exception handling must not disrupt the timing behavior of the main datapath. XtraMAC achieves this by detecting all special cases, including NaN, infinity, subnormal, and overflow, at the input and encoding them as status flags that are forwarded through the same matched register slices as the operands, ensuring that control and data remain temporally aligned throughout the pipeline. At the output, the appropriate result is selected through purely combinational logic, requiring no stalls, pipeline flushes, or control flow divergence. Overflow is resolved by saturating the result to $\pm\infty$ through the same flag selection mechanism, ensuring uniform treatment across all exception types. Thus, the pipeline's latency and throughput are preserved unconditionally, independent of whether inputs are normal or exceptional.

IV. XtraMAC Architecture Details

Guided by the processing-pattern formulation in Section III, XtraMAC separates numeric interpretation from numeric execution: lightweight peripheral logic performs format decoding, bit-mapping, packing, and post-computation, while the arithmetic core comprises a datatype-invariant DSP multiplier and decoupled INT/FP accumulation paths. The architecture is parameterized by (i) the number of supported datatypes N, chosen at synthesis time, and (ii) the maximum parallelism P across these N datatypes, which defines the uniform per-lane structure shared by all stages and is chosen no larger than the bound in Eq. 12. This section details how XtraMAC realizes a fully pipelined mixed-precision MAC datapath with runtime datatype switching.

A. Overall Architecture

Figure 5 illustrates the four-stage pipeline of XtraMAC. A dedicated datatype-select signal is registered at pipeline entry and propagated through all four stages via matched delay slices, selecting one of the N supported datatypes each cycle. All datatype-specific mapping and reconstruction submodules are instantiated statically, and runtime switching is achieved entirely through input datatype controlled multiplexing without any form of reconfiguration. The pipeline processes up to P logical lanes per operation, where P corresponds to the maximum parallelism among the N supported datatypes, ensuring that all formats share a common parallel substrate. The four stages perform operand interpretation and DSP packing (Stage 1), datatype-invariant multiplication and per-lane post-computation (Stage 2), datatype-specific accumulation via decoupled INT/FP adder paths (Stage 3), and final output selection and assembly (Stage 4). The following subsections describe each stage in detail.

B. Stage 1: Operand Interpretation and Bit-Mapping

Stage 1 receives input operands A and B, which may encode multiple concatenated low-precision values, and translates them into datatype-specific bit-packed DSP-port operands for Stage 2. The datatype input identifies the active format among the N supported datatypes and determines how each data value is interpreted. To support N datatypes without reconfiguration, Stage 1 instantiates N parallel mapping submodules. Each submodule decodes A and B according to its datatype definition. Floating-point submodules extract the sign, exponent, and mantissa fields, restore the implicit leading 1, and pack the mantissa bits into the designated DSP input positions, while forwarding the exponent and sign fields as metadata. Integer submodules perform two's-complement decoding, pack the magnitude bits into the DSP inputs, and forward the sign bit. For pure integer datatypes, the exponent metadata is set to

zero, effectively treating the value as a fixed-point mantissa with exponent 0. After all submodules compute in parallel, the datatype signal selects a single pair of bit-packed DSP-port operands and the corresponding per-lane sign and exponent metadata, providing a uniform interface to the multiplication module in Stage 2 and enabling cycle-level switching across heterogeneous datatypes.

C. Stage 2: Multiplier and Post-Compute

Stage 2 receives the bit-packed DSP-port operands and the associated per-lane metadata produced by Stage 1. The DSP-based multiply module is fully datatype-invariant: the DSP slice performs a pure integer multiplication, consistent with the decomposition in Eq. 1 and Eq. 4, which shows that all supported INT/FP formats reduce to a mantissa-level product with separate exponent and sign handling. The resulting product from the DSP slice aggregates all packed lanes into a single wide bitfield. The post-compute module reconstructs the P logical lanes by applying deterministic per-lane shift-and-mask extraction based on the packing layout generated in Stage 1. Floating-point lanes perform leading-zero counting, normalization shifts, and exponent updates, while integer lanes require only magnitude extraction and sign XOR without normalization. The post-compute module instantiates P identical per-lane pipelines so that all lanes are reconstructed in parallel, and Stage 2 outputs P integer or floating-point values for accumulation in Stage 3.

D. Stage 3: Datatype-specific Accumulation

Stage 3 receives the per-lane products from Stage 2 together with the corresponding per-lane accumulator operands C and performs accumulation using decoupled adder modules. The integer module implements a bank of two's-complement adders, while the floating-point module performs exponent alignment, mantissa add/subtract, and renormalization. This structural separation, following the principles in Section III, avoids the area and latency overheads of a unified INT–FP adder. Both modules operate every cycle and compute P lane results in parallel; the datatype signal then selects the output.

E. Stage 4: Output Selection

Stage 4 receives the lane-wise accumulated results and selects the output based on the input data type. The P lane results are concatenated into the final packed output word (e.g., four FP8 lanes or two BF16 lanes forming 32 bits). No additional normalization or conversion is required, and the stage emits one multi-lane MAC result per cycle.

F. Pipeline Behavior

XtraMAC adopts a fixed four-stage logical pipeline in which each stage contains a bounded combinational block followed by a register boundary, cleanly separating logic evaluation from temporal sequencing. The DSP slice is configured with its internal pipeline registers disabled so that the multiplication behaves as a purely combinational block between Stage 1 and Stage 2 registers. All interface signals are time-aligned through matched delay slices to sustain continuous throughput. For example, the input datatype signal, consumed in both Stage 1 and Stage 4, is delayed by the appropriate number of register slices before reaching Stage 4, ensuring that control and data correspond to the same dynamic operation. Likewise, the input operand C, required only in Stage 3, is delayed to align with the products emerging from Stage 2.

XtraMAC fixes the pipeline at four logical stages but leaves the cycle count of each stage configurable at synthesis time. By default, every stage completes in a single clock cycle, yielding an end-to-end latency of four cycles. When a stage with complex logic (e.g., Stage 3) becomes the critical path, the designer can insert additional pipeline registers within it to trade latency for a higher clock frequency. This requires only extending the matched delay slices on parallel paths; the initiation interval remains one. By combining per-stage-configurable latency with pipeline-aligned propagation, XtraMAC provides a deterministic latency and an initiation interval of one across all supported datatypes.

G. System Integration

Although XtraMAC adds an input data type port for runtime datatype selection, this signal functions solely as pipeline-aligned control metadata and does not modify the operand interface or timing behavior of a standard MAC unit. All datatype-dependent formatting and reconstruction logic resides inside the XtraMAC pipeline, so external modules continue to provide and consume the same lane-packed operands used in existing accelerators. With a fixed latency and an initiation interval of one across all datatypes, XtraMAC can replace a conventional MAC unit without requiring any schedule or interface changes, enabling easy system integration.

V. Evaluation

In this section, we evaluate XtraMAC in terms of its mixed-precision coverage and runtime datatype adaptability. We further compare its resource efficiency and performance against state-of-the-art FPGA baselines, including the AMD Xilinx Floating-Point Operator [1] and TATAA [38].

A. Experimental Setting

All designs, including XtraMAC and the baselines, are implemented using Vivado 2022.2 and Vitis 2022.2 on the AMD Xilinx Alveo U55c FPGA card. To ensure a fair comparison, we apply the same tool configurations and compilation settings across all implementations. For floating-point operations, we adopt the round-to-nearest-even (RN-even) rounding mode and flush subnormal inputs to zero. For integer operations, we use two's-complement signed representation, apply saturation on overflow, and truncate extra bits during accumulation.

B. Mixed-Precision Evaluation

We evaluate XtraMAC under various datatype configurations to analyze its mixed-precision capability and adaptive parallelism. In this subsection, each row in Fig. 6 corresponds to a standalone MAC instance synthesized with a

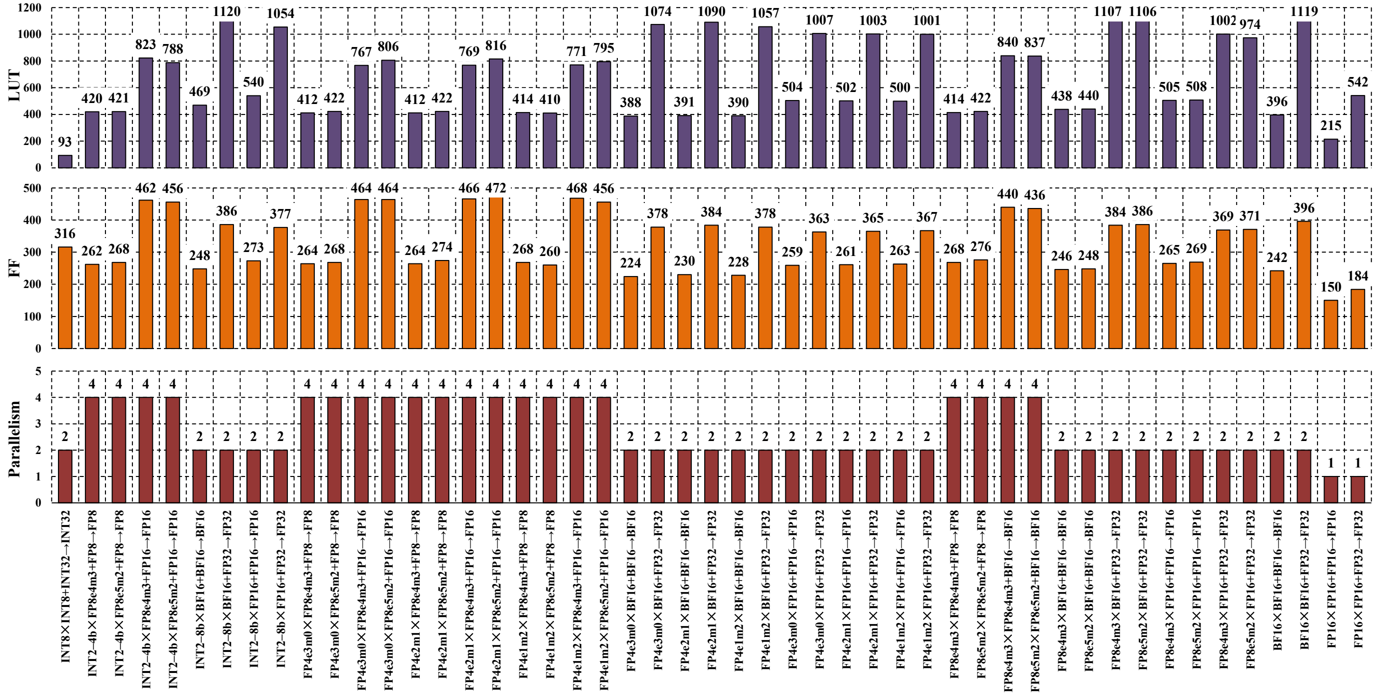

Fig. 6. Resource consumption and parallelism of XtraMAC across all $A \times B + C \to P$ configurations. For brevity, constant metrics (DSP = 1, latency = 4 cycles, II = 1) are omitted. The full exponent–mantissa formats (e.g., E4M3, E5M2) are shown only at the first occurrence of each data type configuration.

single active datatype configuration, and XtraMAC's runtime datatype switching mechanism is disabled. Fig. 6 reports resource consumption for LUT, FF, and DSP usage, along with performance metrics including latency, initiation interval, and achievable parallelism. Across all configurations, XtraMAC maintains a four-cycle latency and an initiation interval of one, indicating that datatype variation does not affect pipeline depth. In general, lower-precision formats such as FP4 and FP8 achieve higher packing parallelism (up to four MAC lanes per DSP), whereas higher-precision formats like BF16 are limited to two lanes per DSP because their wider mantissas reduce spatial packing opportunities.

For detailed resource characterization, the integer mode (INT8×INT8→INT32) provides two-way parallelism with the lowest LUT usage among all evaluated configurations. Floating-point configurations require substantially more LUTs and FFs due to the mantissa-alignment logic in the floating-point adder, where the barrel shifter dominates resource consumption. As the adder bitwidth increases, both the shift range and the shifter complexity grow super-linearly, leading to a significant increase in LUT and FF utilization.

C. Runtime Datatype Switching Evaluation

We evaluate the hardware efficiency of runtime datatype switching using representative datatype combinations derived from Fig. 1. Table III summarizes the post-synthesis resource consumption of XtraMAC for all evaluated cases. Across these configurations, XtraMAC maintains a constant DSP cost, shares LUT and FF usage across the supported datatypes, and preserves a latency of four clock cycles and an initiation interval of one. To quantify the benefits of resource sharing, Fig. 7 further breaks down LUT and FF usage into mapping and post-processing logic, arithmetic logic, and adder logic.

TABLE III
RESOURCE CONSUMPTION FOR RUNTIME-SWITCHING EVALUATION.

Config ID	Data Type Combinations	Model Name(s)	LUT	FF	DSP
I	INT4×BF16 + BF16 BF16×BF16 + BF16	Qwen3-8b-AWQ	436	302	1
II	INT8×INT8 + INT32 BF16×BF16 + BF16	Llama-3.1-8b-W8A8	568	513	1
III	FP8×FP8 + BF16 BF16×BF16 + BF16	Qwen3-8b-FP8 Llama-3.1-8b-FP8	948	622	1
IV	FP4×BF16 + BF16 BF16×BF16 + BF16	GPT-oss-20b†	395	274	1

† The multiplication of UE8M0 with BF16 in GPT-oss-20b is implemented by offsetting the BF16 exponent, so it is not treated as a separate MAC operation.

Config I reuses both the arithmetic core and the adder unit, while the mapping logic remains separate because integer and floating-point operands require different alignment procedures. As a result, the LUT and FF usage of the adder logic is reduced by approximately 50% relative to a naive design that instantiates separate MAC units for each datatype, whereas the mapping cost shows almost no change. Config II shares only the arithmetic core, as INT32 and BF16 accumulations follow different normalization and saturation rules. Moreover, the mapping logic cannot be reused because integer formats do not include exponent alignment, leading to negligible resource reduction. Config III enables partial reuse of the BF16 adder lanes because FP8 and BF16 operate at different levels of

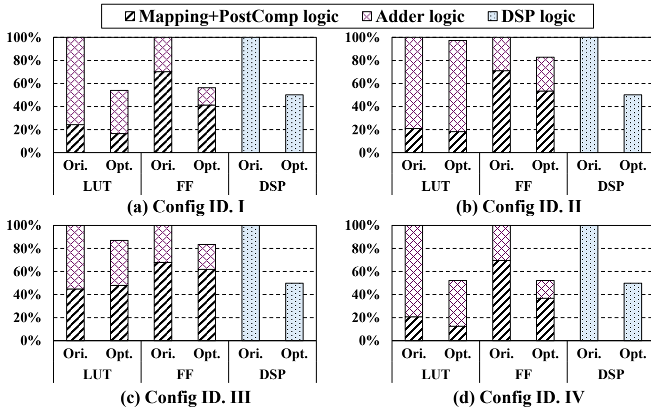

Fig. 7. Normalized resource breakdown for different Config IDs in Table III.

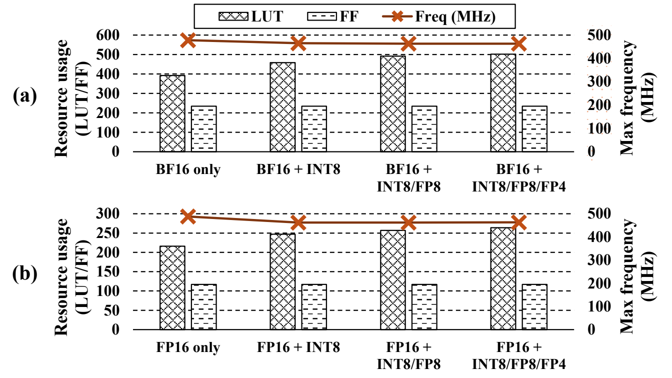

Fig. 8. Scalability evaluation of XtraMAC with increasing mixed-precision datatype support, using a single DSP slice. (a) FP16-based MAC, progressively adding INT8, FP8, and FP4 multiplier operand datatypes. (b) BF16-based MAC, progressively adding the same multiplier operand datatypes.

vector parallelism, four lanes versus two lanes. In this case, two of the four BF16 adder lanes can be shared directly to FP8 adders, which provides approximately 30% LUT and FF savings in the adder logic. Config IV achieves near-complete reuse across the mapping, arithmetic, and adder units. Both MAC types operate with a parallelism of two, and FP4 operands can be expanded to BF16 by zero-padding without requiring exponent alignment or rounding adjustment. This compatibility enables full hardware sharing and results in the lowest LUT and FF utilization among all configurations.

D. Scalability Evaluation of XtraMAC

To evaluate the scalability of XtraMAC, we progressively increase the number of supported datatype combinations starting from BF16 and FP16 MAC baselines, incrementally enabling INT8, FP8 (E4M3), and FP4 (E2M1) mixed-precision modes. At each stage, we synthesize the design and measure FPGA resource utilization and maximum frequency. As shown in Fig. 8, LUT usage increases gradually as additional datatypes are enabled, due to the extra peripheral logic required for operand mapping and post-compute normalization. DSP usage remains constant across all configurations, confirming that the core multiplier is fully shared across all formats. The maximum frequency decreases slightly, from 483 MHz to 462 MHz, indicating that XtraMAC scales gracefully with the number of supported datatypes.

E. Comparison with SOTA MAC IPs on FPGAs

1) Mixed-precision comparison: We compare XtraMAC with the AMD Xilinx Floating-Point Operator IP [1], the vendor's standard MAC implementation for floating-point data formats. Both designs use a non-blocking AXI interface, RN-even rounding, and flush subnormal values to zero. All implementations operate with an initiation interval (II) of 1 and same latency for a fair comparison, and resource utilization is reported after synthesis. Since the vendor operator does not support mixed-precision computation, we augment it with a commonly used open-source integer-to-floating-point conversion module [10], [17] to enable mixed-precision inputs.

Table IV reports resource consumption normalized by parallelism under representative mixed-precision operand formats;

TABLE IV
NORMALIZED RESOURCE UTILIZATION COMPARISON BY PARALLELISM.

Type(A)†	Type(B)	Type(C)	Type(P)	Res.	VendorIP [1]	XtraMAC‡	Red.(%)	Comp.Den.(↑)
INT2-8	BF16	BF16	BF16	LUT	331	235	29.0%	1.4x
				FF	222	124	44.1%	1.8x
				DSP	1	0.5	50.0%	2.0x
INT2-8	FP16	FP16	FP16	LUT	387	270	30.2%	1.4x
				FF	262	137	47.7%	1.9x
				DSP	1	0.5	50.0%	2.0x
FP4	BF16	BF16	BF16	LUT	301	196	34.9%	1.5x
				FF	226	115	49.1%	2.0x
				DSP	1	0.5	50.0%	2.0x
FP4	FP16	FP16	FP16	LUT	357	251	29.7%	1.4x
				FF	266	131	50.8%	2.0x
				DSP	1	0.5	50.0%	2.0x
FP8	BF16	BF16	BF16	LUT	301	219	27.2%	1.4x
				FF	226	123	45.6%	1.8x
				DSP	1	0.5	50.0%	2.0x
FP8	FP16	FP16	FP16	LUT	357	253	29.1%	1.4x
				FF	266	133	50.0%	2.0x
				DSP	1	0.5	50.0%	2.0x

† FP4 = E2M1, FP8 = E4M3.
‡ XtraMAC includes a non-blocking AXI wrapper for fair comparison. This introduces additional resource usage relative to Fig 6, which reports only the core MAC datapath.

we divide each design's total LUT, FF, and DSP usage by its number of MAC lanes to obtain per-operation utilization. Across these configurations, XtraMAC reduces LUT usage by an average of 30.0%, FF usage by 47.9%, and DSP usage by 50.0% compared with the vendor IP. These reductions stem from XtraMAC 's adaptive parallelism for low-precision computation, which packs multiple mixed-precision lanes into a single DSP slice and increases effective multiplier utilization. Moreover, the bit-mapping datapath eliminates redundant bit operations and datatype-specific logic, reducing overall LUT and FF overhead. As illustrated in Fig. 9, our data-packing strategy sustains high DSP utilization across all datatype combinations, enabling substantially more resource-efficient MAC execution than the vendor IP [1]. We further report compute density for LUTs, FFs, and DSPs as the ratio between the vendor IP resource usage and the XtraMAC usage; across all evaluated formats and resource types, XtraMAC achieves a compute density improvement between 1.4× and 2.0×, indicating that each unit of hardware contributes more effective compute capability.

We further evaluate the maximum frequency against the vendor IP with a single DSP slice. As shown in Fig. 10,

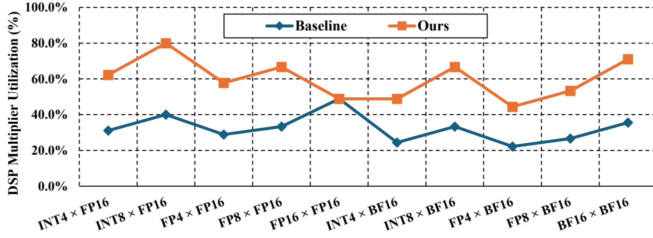

Fig. 9. DSP utilization under different data types. (FP8=E4M3, FP4=E2M1).

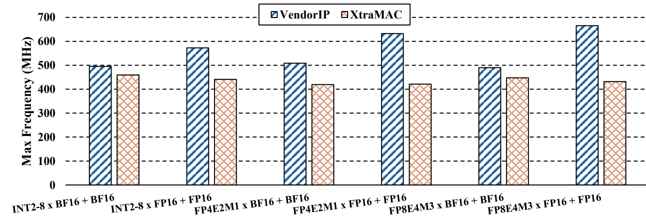

Fig. 10. Maximum frequency comparison with a single DSP slice.

XtraMAC runs on average 22% slower than the vendor IP [1] based MAC implementation. This reduction is expected: each XtraMAC configuration provides 2× data parallelism per DSP compared to the 1× baseline, which increases peripheral LUT and FF usage and introduces heavier routing congestion. Despite the frequency reduction, all XtraMAC configurations exceed 400 MHz, well above the typical operating frequency of FPGA accelerator deployments. Because the 2× parallelism doubles the per-DSP throughput, XtraMAC still delivers approximately 1.56× higher effective throughput per DSP even after accounting for the frequency gap, confirming that the improved compute density more than compensates for the modest frequency overhead.

2) Runtime datatype switching comparison: We evaluate XtraMAC's capability to switch between data formats dynamically during execution. Two baselines are included for comparison: the vendor's IP [1] with spatial replication of INT8 and BF16 MAC units, and the TATAA accelerator [38]. All designs alternate between INT8-based and BF16-based MAC operations to emulate workloads that require runtime datatype switching. Resource per operation is obtained by dividing the total resources by the number of BF16 or INT8 lanes realized in each design. For a fair comparison, all implementations use identical interface configurations and operate with an initiation interval (II) of 1 and a four-cycle latency.

Table V shows that XtraMAC provides substantial reductions in per-operation resource consumption under runtime precision switching. Relative to TATAA [38], XtraMAC reduces LUT usage by 59.7%, FF usage by 72.5%, and DSP usage by 93.8%. Relative to the vendor IP [1], the reductions are 35.5% for LUTs, 58.7% for FFs, and 75.0% for DSPs. These results indicate that XtraMAC achieves significantly higher hardware efficiency for floating-point computation while maintaining reasonable per-INT8 resource usage compared to INT8-optimized designs such as TATAA [38].

The inefficiency of the baselines in floating-point computation arises from how each design handles datatype heterogeneity. The vendor IP [1] spatially replicates independent INT8 and BF16 pipelines, preventing reuse of arithmetic or alignment logic across precisions and thereby duplicating hardware. TATAA [38] instead maps BF16 MACs onto its INT8 datapath, but must decompose each BF16 operation into multiple INT8 micro-operations with additional control and operand-adjustment hardware, which reduces utilization and BF16 efficiency. In contrast, XtraMAC employs a shared bit-mapping front-end that normalizes heterogeneous formats before entering the arithmetic core, so all precisions reuse the same multiplier–adder pipeline without duplication, substantially improving overall hardware efficiency under the same resource budget.

TABLE V
NORMALIZED RESOURCE CONSUMPTION PER OPERATION.

Design	Resource per BF16 operation			Resource per INT8 operation		
	LUT	FF	DSP	LUT	FF	DSP
Vendor IP [1]	220.0	310.5	1	110.0	155.3	0.5
TATAA [38]†	352.0	467.0	4	22.0	29.2	0.25
XtraMAC	142.0	128.3	0.25	142.0	128.3	0.25

† For TATAA, we evaluate only the FP adder and multiplier and omit the `fapp` units (approximate sqrt/div), yielding lower resource usage than the full design in [38].

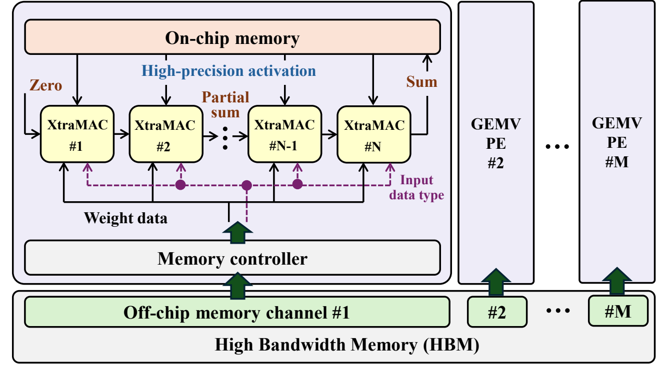

Fig. 11. Mixed-precision GEMV architecture based on XtraMAC.

VI. Case Study: Mixed-Precision LLM Inference

This section demonstrates how XtraMAC can be integrated into a practical tile-based GEMV pipeline and quantifies its benefits for mixed-precision LLM inference. We first describe the system-level integration of XtraMAC, then characterize the mixed-precision requirements of representative LLM workloads. Next, we evaluate a mixed-precision GEMV kernel based on XtraMAC and compare it against state-of-the-art GPU kernels. Finally, we extend an analytical simulation framework to estimate end-to-end LLM inference latency using XtraMAC-accelerated GEMV operators.

A. Integration of XtraMAC into Tile-Based GEMV

Figure 11 illustrates how XtraMAC is incorporated into a tile-parallel GEMV engine. Modern FPGA GEMV pipelines [23], [42] implement streaming multiply–accumulate (MAC) chains within each processing element (PE), where

TABLE VI
REPRESENTATIVE QUANTIZED LLM DEPLOYMENT PROFILES.

Checkpoints	Downloads*	Practical use cases†
Qwen-3-8B-AWQ	222,126	Edge AI reasoning with 2–3% accuracy loss; long-context NLP with 91–97% benchmark retention and a 4× context extension.
Llama-3.1-8B-W8A8	27,536	Commercial chatbots with 50% memory reduction, 2× throughput, and 99–100% accuracy for multilingual customer service.
Qwen-3-8B-FP8	429,968	Local agentic tasks and data analytics workflows; supports 131k-token contexts for scalable multilingual RAG.
Llama-3.1-8B-FP8	168,122	Multilingual assistants with 99.52% accuracy recovery; optimized for fast commercial API deployments.
GPT-oss-20B	4,633,438	Consumer fine-tuning, customizable inference latency, and low-latency analysis for agent-based data research.

* Download counts recorded on HuggingFace during October 2025.
† Accuracy loss in use cases only reflects model-level quantization effects.

weights are supplied directly from HBM and activations are buffered on-chip for reuse. XtraMAC preserves this interface and replaces the scalar MACs in prior designs without modifying the PE topology, tiling strategy, or global dataflow.

During execution, each wide memory-interface word (typically 256 or 512 bits) read from HBM is split into per-lane weight segments and dispatched to the corresponding XtraMAC instances within the PE. Each XtraMAC instance receives its weight segment alongside the corresponding activation value from the on-chip buffer. A per-tile datatype control signal, stored in memory alongside the weight tiles, is propagated synchronously with the operands to all XtraMAC units within the tile, ensuring that each instance applies the correct mapping rule and accumulation path at runtime. The resulting per-lane partial sums are accumulated across the cascaded MAC chain and written back as the final GEMV output, with multiple PEs operating concurrently across HBM channels to preserve bandwidth-scaling behavior.

B. Mixed-Precision Requirements in LLM Workloads

Modern LLM deployments such as Llama [16], Qwen [41], and GPT-oss [30] heavily rely on quantization to reduce memory footprint and latency while preserving accuracy. Rather than adopting a single numeric format, practical systems combine integer and floating-point datatypes across layers. As summarized in Table I, weight-only quantization uses integer weights with FP activations; weight–activation approaches quantize both operands; and mixed-format checkpoints (e.g., GPT-oss) interleave FP4/FP8 with BF16 for stability. Table VI profiles representative quantized checkpoints deployed in multilingual assistants, long-context RAG, and on-device LLMs. Collectively, these models indicate active deployment of INT4, FP4, and FP8 formats and highlight the need for GEMV engines that natively support mixed-precision and runtime datatype switching.

C. Mixed-Precision GEMV Kernel Evaluation

Across all evaluated models, these two patterns consistently dominate decode-time GEMV: (i) INT4×BF16→BF16 and (ii) FP4×BF16→BF16. These two patterns therefore serve as representative GEMV kernels for evaluating XtraMAC. All FPGA experiments are conducted on an AMD Xilinx U55c accelerator card equipped with 32 HBM channels. The XtraMAC-based GEMV kernel is synthesized and implemented using Vivado 2022.2 and Vitis 2022.2 with post-place-and-route timing. Power consumption is measured using `xbutil` under steady-state streaming. For comparison, GPU baselines are conducted on an NVIDIA H100 PCIe card using CUTLASS [29]; execution time is measured with official GEMV kernels, and power is monitored using `nvidia-smi`. This setup ensures a consistent and reproducible evaluation environment across both platforms.

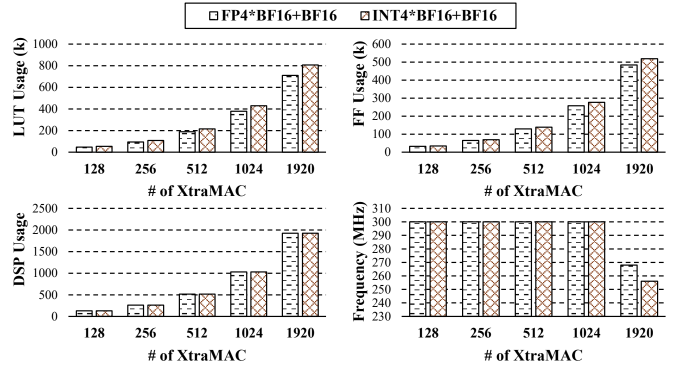

Fig. 12. Post-implementation resource usage (LUT, FF, DSP) and frequency for mixed-precision GEMV kernel with various numbers of XtraMAC.

To exploit HBM bandwidth, the GEMV workload is partitioned into M tiles, each mapped to a processing element (PE) connected to a distinct HBM channel. Weights are stored in HBM, whereas activations are buffered on-chip. Inside each PE, the dot product is realized through a chain of cascaded XtraMAC modules. For a given HBM-channel bitwidth, weight precision, and datatype-dependent parallelism P, the number of cascaded XtraMAC instances per channel is determined by

$$N_{\text{MAC}} = \frac{\text{BitWidth}_{\text{channel}}}{\text{BitWidth}_{\text{weight}} \times P}$$

In implementation, each HBM channel provides a 512-bit interface; with INT4 weights (4 bits) and $P = 2$ lanes per XtraMAC, a single channel supplies $512/(4 \times 2) = 64$ MAC inputs per cycle. Thus, a design with 32 HBM channels can instantiate up to 2048 XtraMAC instances in principle, while our implementation adopts a configuration with 1920 cascaded XtraMAC instances (spread across 30 active HBM channels) to reserve one channel for activation read and one for write-back, and to ensure routing timing closure.

Post-implementation results in Figure 12 indicate that LUT, FF, and DSP utilization scale linearly with the number of instantiated XtraMAC modules. The design sustains 300 MHz across configurations up to 1024 instances and exhibits only moderate frequency degradation (250–270 MHz) at 1920 instances due to routing congestion near the HBM interface. Because the kernel is bandwidth-bound at scale, these timing

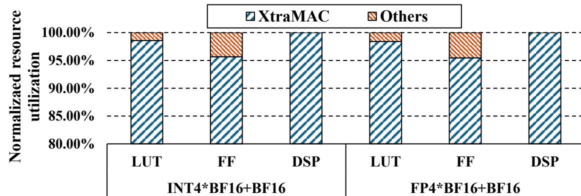

Fig. 13. System-level resource breakdown (512-XtraMAC configuration).

TABLE VII
COMPARISON OF MIXED-PRECISION GEMV PERFORMANCE.

Design	Time (ms)	Power (W)	Energy (J)	Speedup	Energy Eff.
$1 \times 4096 \times 4096$ **GEMV**					
CUTLASS (H100)	0.0294	135	0.00397	1.0×	1.0×
XtraMAC (U55c)	**0.0246**	**85**	**0.00209**	**1.2×**	**1.9×**
$1 \times 4096 \times 12288$ **GEMV**					
CUTLASS (H100)	0.0879	135	0.01187	1.0×	1.0×
XtraMAC (U55c)	**0.0743**	**85**	**0.00632**	**1.2×**	**1.9×**

reductions have negligible impact on overall throughput. Figure 13 further presents the system-level resource breakdown under the 512-XtraMAC configuration. The XtraMAC instances dominate resource consumption, accounting for 98.5% of LUTs, 95.6% of FFs, and 100% of DSPs. The remaining resources are consumed by supporting logic, including datatype control registers and on-chip activation buffers. HBM controllers reside in the Alveo static shell [2] and are excluded from user-logic resource accounting. All non-MAC components are registered in parallel with the operand datastream and introduce no additional pipeline stages, preserving the fixed latency and II = 1 of the MAC pipeline.

Table VII presents the performance comparison against the H100 GPU. Despite the GPU's substantially higher peak memory bandwidth (2 TB/s versus 460 GB/s), the XtraMAC-based FPGA kernel achieves 1.2× lower latency and 1.9× higher energy efficiency on the $1 \times 4096 \times 4096$ and $1 \times 4096 \times 12288$ GEMV workloads. These gains arise from the efficient mixed-precision MAC datapath in XtraMAC, the absence of format-conversion overheads, and the kernel's ability to sustain high effective HBM utilization (approximately 74%). As a result, the FPGA design operates close to its bandwidth roofline and surpasses the GPU on bandwidth-bound GEMV workloads.

D. End-to-End Mixed-Precision LLM Inference Simulation

We evaluate system-level behavior using the analytical framework of [7], which models transformer layers as alternating compute and memory phases under idealized streaming and weight-reuse assumptions. All models in Table VI are simulated on the AMD Alveo V80 accelerator (2.6M LUTs, 10,848 DSPs, 300 MHz, and 810 GB/s HBM). The baseline instantiates the vendor Floating-Point IP using the resource profiles in Table IV, while our configuration replaces these units with XtraMAC. All other components, including datatype packing rules, checkpoint-derived MAC counts, and tiling behavior, are kept fixed so that performance differences reflect only the arithmetic-unit density.

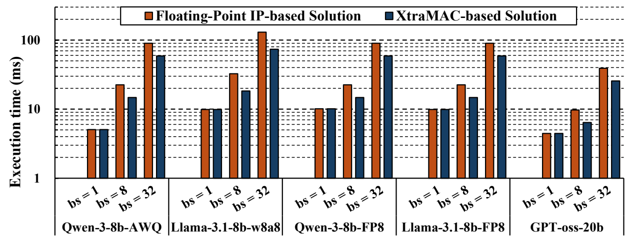

Fig. 14. Decode-stage execution time for different LLMs at context length 512 and batch sizes $\{1, 8, 32\}$.

Figure 14 reports decode latency at a context length of 512 for batch sizes $\{1, 8, 32\}$. At batch size 1, all five evaluated models (Qwen-3-8B-AWQ, Llama-3.1-8B-W8A8, Qwen-3-8B-FP8, Llama-3.1-8B-FP8, and GPT-oss-20B) operate in a memory-bound regime. Weight streaming dominates the end-to-end latency (approximately 4.4–10.0 ms), and compute differences between the baseline and XtraMAC have negligible impact. As the batch size increases, the compute workload grows while the weight-fetch cost remains nearly constant, placing all models in a compute-bound regime at batch sizes 8 and 32. In this region, the reduced LUT and DSP footprint of XtraMAC enables more MAC units to be instantiated on the FPGA platform, which produces consistent reductions in total latency. At batch size 32, XtraMAC delivers improvements ranging from 1.5× to 1.8× for the end-to-end LLM inference task. These trends align with the proportion of mixed-precision GEMV operations in each model and confirm that arithmetic-unit density, rather than memory bandwidth, becomes the dominant performance limiter at large batch sizes.

E. Design Guidelines

In this subsection, we summarize several guidelines for integrating XtraMAC into system-level designs.

First, XtraMAC maintains a constant latency and an initiation interval of one across all supported datatypes, and can therefore serve as a drop-in replacement for scalar MAC units in existing GEMV/GEMM pipelines, requiring no changes to pipeline depth, scheduling logic, or interface timing. Second, runtime datatype switching is equally straightforward: a per-tile control signal propagated synchronously with the operand datastream is sufficient, so mixed-precision workloads can be scheduled as if all datatypes share a single unified pipeline, without the need for pipeline flushes or reconfiguration. Last, from a quantization perspective, lane packing efficiency scales inversely with operand bitwidth. Sub-8-bit quantization schemes (e.g., INT4, FP4) can therefore be prioritized in FPGA deployments to maximize per-DSP parallelism, directly translating into higher overall throughput.

VII. RELATED WORK

In this section, we review existing approaches to mixed-precision MAC computation and runtime datatype switching, and position XtraMAC against their remaining limitations.

A. Fixed-Precision Designs

General-purpose processors expose multiple numerical formats via fixed hardware pipelines. Modern CPUs support integer and low-precision floating-point arithmetic through instruction-set extensions such as AVX-512, AMX, and ARM SVE, while GPUs accelerate machine learning workloads using Tensor Cores with support for FP16, BF16, FP8, and INT8 [27], [28]. Domain-specific accelerators such as Google TPU [15], [18], [37] employ systolic arrays optimized for a fixed set of precisions (e.g., BF16 and INT8), with each datatype served by dedicated compute pipelines. However, across all these designs, mixed-precision computation is handled either by upcasting low-precision operands to a common high-precision format, or by routing different datatypes through physically separated execution pipelines. Both strategies leave hardware resources idle when only a subset of datatypes is active, and neither exploits the fine-grained bit-level parallelism available within a shared datapath.

B. Configurable-Precision Designs

A second class of designs introduces runtime precision adaptivity. BitFusion [33] fuses 2-bit processing elements dynamically to match the integer bitwidth of each DNN layer. Stripes [19] employs bit-serial computation so that execution latency scales proportionally with operand precision. OLAccel [31] supports operand-level precision variability through heterogeneous MAC units and temporal decomposition. More recently, FlexiBit [34] proposes a configurable bit-parallel reduction tree supporting arbitrary FP and INT precisions, including non-power-of-two formats. However, BitFusion [33], Stripes [19], and OLAccel [31] support only integer formats and lack native floating-point computation. FlexiBit, while supporting FP formats, is an ASIC design and thus requires hardware re-fabrication to accommodate new quantization schemes. None of these designs support cycle-level runtime switching between heterogeneous INT and FP formats within a shared datapath.

XtraMAC bridges these gaps by unifying integer, floating-point, and mixed-precision multiplication within a single resource-compact FPGA datapath. By reducing all MAC formats to a shared integer mantissa product and decoupling accumulation paths, XtraMAC achieves cycle-level runtime datatype switching with constant latency and high throughput.

VIII. Conclusion

MAC operations dominate the computational cost of modern LLM inference, yet existing FPGA MAC designs remain inefficient because they either upcast low-precision operands or replicate datatype-specific datapaths, resulting in poor DSP utilization and limited runtime flexibility. This work introduces XtraMAC, a compact, datatype-adaptive MAC architecture that supports mixed-precision computation with cycle-level runtime switching. The key architectural novelty of XtraMAC lies in unifying INT and FP formats through a shared integer-product formulation, enabling multiple low-precision lanes to be packed within a single DSP while maintaining a fixed latency. Compared with state-of-the-art MAC designs, XtraMAC significantly improves hardware efficiency, delivering 1.4–2.0× higher compute density while reducing LUT, FF, and DSP usage by 30.0%, 47.9%, and 50.0%, respectively.

Acknowledgment

This work is supported by the Ministry of Education AcRF Tier 3 grant, Singapore (MOE-MOET32024-0003), and a Google Gift 2025. We also thank the AMD Heterogeneous Accelerated Compute Clusters (HACC) program [3] for the generous hardware donation.

References

[1] *AMD Xilinx Floating-Point Operator v7.1 LogiCORE IP Product Guide (PG060)*, Advanced Micro Devices, Inc., accessed Oct. 2025. [Online]. Available: https://docs.amd.com/v/u/en-US/pg060-floating-point

[2] AMD, "Alveo U55C accelerator card: Support and downloads," https://www.amd.com/en/products/accelerators/alveo/u55c/a-u55c-p00g-pq-g.html, accessed Oct. 2025.

[3] AMD Xilinx, "Heterogeneous accelerated compute cluster (HACC) at NUS," https://xacchead.ddns.comp.nus.edu.sg/, accessed Oct. 2025.

[4] AMD Xilinx, Inc., "Vivado design suite reference guide: Model-based dsp design using system generator (ug958)," accessed Oct. 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug958-vivado-sysgen-ref/DSP48E2

[5] A. Arora, M. Ghosh, S. Mehta, V. Betz, and L. K. John, "Tensor slices: FPGA building blocks for the deep learning era," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, Dec. 2022.

[6] J. Chee, Y. Cai, V. Kuleshov, and C. De Sa, "Quip: 2-bit quantization of large language models with guarantees," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[7] H. Chen, J. Zhang, Y. Du, S. Xiang, Z. Yue, N. Zhang, Y. Cai, and Z. Zhang, "Understanding the potential of fpga-based spatial acceleration for large language model inference," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 18, no. 1, Dec. 2024.

[8] Y. Chen, C. Gong, and B. He, "HiPACK: Efficient sub-8-bit direct convolution with SIMD and bitwise management," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025, pp. 1579–1591.

[9] Y. Chen, J. Dotzel, and M. S. Abdelfattah, "M4bram: Mixed-precision matrix-matrix multiplication in fpga block rams," in *2023 International Conference on Field Programmable Technology (ICFPT)*, 2023, pp. 69–78.

[10] J. Dawson, "Integer-to-floating point converter (Verilog implementation)," https://github.com/dawsonjon/fpu/blob/master/int_to_float/int_to_float.v, accessed Oct. 2025.

[11] T. Dettmers, R. Svirschevski, V. Egiazarian, D. Kuznedelev, E. Frantar, S. Ashkboos, A. Borzunov, T. Hoefler, and D. Alistarh, "Spqr: A sparse-quantized representation for near-lossless llm weight compression," *arXiv preprint arXiv:2306.03078*, 2023.

[12] A. Ehliar, "Area efficient floating-point adder and multiplier with IEEE-754 compatible semantics," in *2014 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2014, pp. 131–138.

[13] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pretrained transformers," *arXiv preprint arXiv:2210.17323*, 2022. [Online]. Available: https://arxiv.org/abs/2210.17323

[14] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with INT8 optimization on Xilinx devices," Xilinx, Inc., White Paper WP486 v1.0.1, April 2017. [Online]. Available: https://docs.amd.com/api/khub/documents/z7yAy_aweTmRYkGaTVyhbw/content

[15] Google Cloud, "Cloud TPU v5e," https://cloud.google.com/tpu/docs/v5e, accessed Oct. 2025.

[16] A. Grattafiori, A. Dubey *et al.*, "The Llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[17] D. K. Hartman, "Floating point multiply/add unit for the M-Machine node processor," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1996. [Online]. Available: http://hdl.handle.net/1721.1/38791

[18] 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 (ISCA)*, 2017, pp. 1–12.

[19] P. Judd, J. Albericio, T. H. Hetherington, T. M. Aamodt, and A. Moshovos, “Stripes: Bit-serial deep neural network computing,” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 19:1–19:12.

[20] M. Kerner, K. Tammemäe, J. Raik, and T. Hollstein, “Triple fixed-point MAC unit for deep learning,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1404–1407.

[21] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, “FP-BNN: Binarized neural network on FPGA,” *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[22] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, “AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration,” in *Proceedings of Machine Learning and Systems (MLSys)*, vol. 6, 2024, pp. 87–100.

[23] J. Liu, S. Zeng, L. Ding, W. Soedarmadji, H. Zhou, Z. Wang, J. Li, J. Li, Y. Dai, K. Wen, S. He, Y. Sun, Y. Wang, and G. Dai, “FlightVGM: Efficient video generation model inference with online sparsification and hybrid precision on FPGAs,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2025, pp. 2–13.

[24] X. Liu, Y. Chen, P. Ganesh, J. Pan, J. Xiong, and D. Chen, “HiKonv: High throughput quantized convolution with novel bit-wise management and computation,” in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 140–146.

[25] Y. O. M. Moctar, N. George, H. Parandeh-Afshar, P. Ienne, G. G. Lemieux, and P. Brisk, “Reducing the cost of floating-point mantissa alignment and normalization in FPGAs,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2012, pp. 255–264.

[26] NVIDIA Corporation, “NVIDIA A100 tensor core GPU architecture,” NVIDIA, White Paper, May 2020. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/nvidia-ampere-architecture-whitepaper

[27] ——, “NVIDIA A100 tensor core GPU datasheet,” https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf, 2021, accessed Oct. 2025.

[28] ——, “NVIDIA H100 tensor core GPU architecture,” NVIDIA, White Paper, 2022. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/nvidia-h100-whitepaper

[29] ——, “CUTLASS 3.x performance profiling results,” https://github.com/NVIDIA/cutlass, 2025, accessed Oct. 2025.

[30] OpenAI, “gpt-oss-120b & gpt-oss-20b model card,” *arXiv preprint arXiv:2508.10925*, 2025. [Online]. Available: https://arxiv.org/abs/2508.10925

[31] E. Park, D. Kim, and S. Yoo, “Energy-efficient neural network accelerator based on outlier-aware low-precision computation,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 688–698.

[32] M. R. Pillmeier, M. J. Schulte, and E. G. Walters III, “Design alternatives for barrel shifters,” in *Advanced Signal Processing Algorithms, Architectures, and Implementations XII*, vol. 4791. SPIE, 2002, pp. 436–447.

[33] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, “Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural network,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[34] F. Tahmasebi, Y. Wang, B. Y. H. Huang, and H. Kwon, “FlexiBit: Fully flexible precision bit-parallel accelerator architecture for arbitrary mixed precision AI,” 2024. [Online]. Available: https://arxiv.org/abs/2411.18065

[35] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “FINN: A framework for fast, scalable binarized neural network inference,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 65–74.

[36] Y. Umuroglu, L. Rasnayake, and M. Själander, “BISMO: A scalable bit-serial matrix multiplication overlay for reconfigurable computing,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 307–314.

[37] A. Vahdat, “Ironwood: The first Google TPU for the age of inference,” https://blog.google/products/google-cloud/ironwood-tpu-age-of-inference/, accessed Oct. 2025.

[38] J. Wu, M. Song, J. Zhao, Y. Gao, J. Li, and H. K.-H. So, “TATAA: Programmable mixed-precision transformer acceleration with a transformable arithmetic architecture,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 18, no. 1, pp. 14:1–14:31, 2025.

[39] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, “Smoothquant: Accurate and efficient post-training quantization for large language models,” in *International conference on machine learning*. PMLR, 2023, pp. 38 087–38 099.

[40] Xilinx, Inc., *UltraScale Architecture DSP Slice User Guide (UG579)*, Xilinx, Inc., 2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp

[41] A. Yang *et al.*, “Qwen3 technical report,” *arXiv preprint arXiv:2505.09388*, 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[42] S. Zeng, J. Liu, G. Dai, X. Yang, T. Fu, H. Wang, W. Ma, H. Sun, S. Li, Z. Huang, Y. Dai, J. Li, Z. Wang, R. Zhang, K. Wen, X. Ning, and Y. Wang, “FlightLLM: Efficient large language model inference with a complete mapping flow on FPGAs,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2024, pp. 223–234.

[43] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, “DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[44] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci, “Atom: Low-bit quantization for efficient and accurate LLM serving,” in *Proceedings of Machine Learning and Systems (MLSys)*, vol. 6, 2024, pp. 196–209.