

# Efficient LLM Decoding on Ryzen AI NPUs

Zhenyu Xu<sup>1</sup>, Miaoxiang Yu<sup>1</sup>, Jillian Cai<sup>1</sup>, Qing Yang<sup>2</sup>, Tao Wei<sup>1</sup>

<sup>1</sup>Electrical and Computer Engineering, Clemson University, Clemson, USA

<sup>2</sup>Electrical and Computer Engineering, University of Rhode Island, Kingston, USA

{z xu3, miaoxiy, cai18}@clemson.edu, qyang@uri.edu, twei2@clemson.edu

**Abstract**—We propose an efficient and scalable LLM decoding framework optimized for AMD Ryzen AI NPUs, leveraging two novel techniques: FusedDQP and FlowKV. FusedDQP fuses dequantization with projection to minimize memory operations and latency, while FlowKV introduces a pipelined, bandwidth-optimized approach for KV cache access across compute tiles (CT). Together, these methods deliver substantial improvements in both speed and energy efficiency without altering model accuracy. Our solution achieves up to 14.2× speedup and 2.66× power efficiency gains compared to existing state-of-the-art (SOTA) NPU baselines, demonstrating linear scalability with CT count and robustness across LLaMA-3.1/3.2 model variants (1B, 3B, and 8B parameters). We also benchmark against CPU and iGPU on the same platform, our performance surpasses CPU and iGPU (up to 1.8x and 16.2x speedup), while delivering substantially improved energy efficiency (up to 3.63x and 11.38x for CPU and iGPU, respectively).

**Index Terms**—AMD Ryzen AI NPU, LLM decoding, Hardware-aware optimization, Close-to-metal optimization

## I. INTRODUCTION

The deployment of Large Language Models (LLMs) on local devices is gaining traction due to the need for reduced latency, offline functionality, and privacy. Running LLMs locally ensures that sensitive data remains on-device, mitigating privacy concerns associated with cloud-based processing. Moreover, local inference eliminates network-induced delays, providing faster response times essential for real-time applications [1].

Traditionally, CPUs and integrated GPUs (iGPUs) have been utilized for on-device LLM inference. However, the computational intensity of LLMs can monopolize these resources, adversely affecting the performance of other concurrent tasks. To address this challenge, Neural Processing Units (NPUs) have emerged as dedicated hardware accelerators designed specifically for AI workloads. NPUs enable efficient execution of AI tasks without impeding the performance of general-purpose processors [2].

Chip manufacturers, including AMD, Intel, Apple, and Qualcomm, have integrated NPUs into their processors [3], [4]. AMD’s Ryzen AI NPUs are based on the XDNA architecture, which consists of a 2D array of AI Engine tiles. These tiles function as a Coarse-Grained Reconfigurable Array (CGRA) with Very Long Instruction Word (VLIW) compute units [5], [6]. Unlike CPUs or GPUs, the Ryzen AI NPU features a flexible data movement architecture that enables direct communication between compute tiles [7]–[11]. This helps alleviate

data movement bottlenecks, leading to more efficient compute in terms of both speed and power consumption.

However, current state-of-the-art (SOTA) LLM decoding is lacking [12]. This paper introduces two novel techniques—FusedDQP and FlowKV—designed to enhance LLM decoding performance on Ryzen AI NPUs. FusedDQP fuses dequantization and projection, which is essentially matrix-vector multiplication (MVM) operations. FlowKV restructures attention into a dataflow model, optimizing the handling of key-value pairs (KV cache). Collectively, FusedDQP and FlowKV achieve a constant high read memory bandwidth utilization ( $U_{mem}^{rd}$ ), approaching theoretical performance limits.

We evaluate our approach using 4-bit quantized LLaMa 3.1 and 3.2 models of varying sizes (1B, 3B, and 8B parameters) on an AMD Ryzen AI 5 340 processor [13], [14]. Our method outperforms the current SOTA [15] (up to 14.2x speedup). Additionally, our approach demonstrates significant power efficiency gain over SOTA method (up to 2.66x). Furthermore, we benchmark our method against CPU and iGPU executions on the same hardware platform [12]. Particularly at longer sequence lengths, the NPU-accelerated decoding not only matches but surpasses CPU (up to 1.8x speedup) and iGPU (up to 16.2x speedup) performance, all while delivering substantially improved energy efficiency (up to 3.63x and 11.38x for CPU and iGPU, respectively).

Main contributions are:

- **FusedDQP:** A fused dequantization + projection (MVM) kernel for all projection layers in LLM decoding, designed to approach memory bandwidth limits on Ryzen AI NPUs.
- **FlowKV:** An IO-aware attention scheme on Ryzen AI NPUs that streamlines KV-cache access to maximize bandwidth utilization during LLM decoding.

## II. BACKGROUND

This section explains the architecture of the Ryzen AI NPU and typical LLM models (focus on LLM decoding) to provide the necessary context for understanding our proposed methods.

### A. AMD Ryzen AI NPU Architecture

1) *Platform Overview:* The AMD Ryzen AI NPU architecture is built around interconnected 2D processing units [5]. Exemplified by the NPU2, these units are referred to as Compute Tiles (CTs). It features a 32-CT grid (8 CT column and each contains 4 CTs) in a checkerboard layout, shown in Figure 1. Designed as VLIW processors, CTs can issue up to seven instructions per cycle. These instructions are dispatched

Identify applicable funding agency here. If none, delete this.

across two parallel processors: the scalar processor, which handles complex pointer arithmetic, and the vector processor, which is optimized for SIMD (Single Instruction Multiple Data) operations. Each CT runs at a clock speed of 1 GHz.

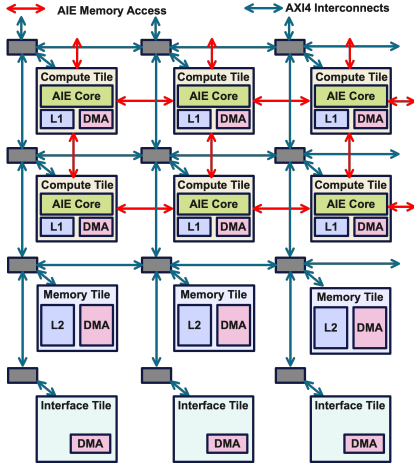


Fig. 1. NPU architecture overview

2) *Compute Tiles (CTs)*: The vector processor supports both floating-point and fixed-point operations. Integrated pre-add units handle basic vector functions such as minimum/maximum calculations and direct vector comparisons. CT local data memory (DM), also referred to as L1 memory, is divided into eight banks (a total of 64 KB), each accessible via a DMA (Direct Memory Access) interface, shown in Figure 2.

CTs communicate through two primary interfaces. The first is AIE (AI Engine) memory access, enabling direct load/store operations between neighboring CTs. The second is the AXI4 (Advanced eXtensible Interface 4) interconnect, which allows DMA transfers to non-adjacent CTs via AXI4 streams. This interface also supports input broadcasting by duplicating data for multiple CTs.

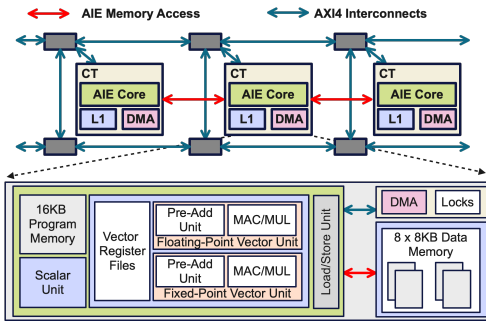


Fig. 2. CT overview

3) *Memory Tiles (MTs) and Interface Tiles (ITs)*: The platform includes 8 Memory Tiles (MTs), positioned at the bottom of the CT array, shown in Figure 1. Each tile contains 512 KB of high-density, high-bandwidth memory, also referred to as L2 memory. Interface Tiles (ITs) provide communication with access to main memory (off-chip DDR memory). MT interfaces with IT to access main memory via AXI4 streams. Lightweight

DMA engines within the MT and IT ensure efficient data transfer between the main memory and the CT array.

4) *Programming*: Programming the platform relies on two main components. Kernels are directly mapped to CTs to handle core computations, while the Multi-Level Intermediate Representation (MLIR) manages connections and creates data buffers between tiles, facilitating efficient communication and coordination [8]. These features collectively ensure scalability and the efficient execution of large-scale computations.

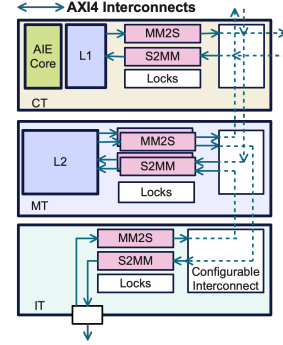


Fig. 3. Data movement architecture on AMD NPU

## B. Data Movement on NPU

The primary data transfers occur between main memory and L2, as well as between L2 and L1. Additionally, the architecture supports flexible interconnects, enabling direct transfers such as main memory to L1 and L1 to L1. A data transfer involves a source DMA channel (MM2S, or Memory-Mapped to Stream), a destination DMA channel (Stream to Memory-Mapped), and a configured stream interconnect connecting the two (see Figure 3). The MM2S channel reads data from memory and places it on the stream, while the S2MM channel retrieves the data from the stream and writes it to memory. Each CT, MT, and IT is equipped with a stream interconnect switch and multiple MM2S and S2MM DMA channels (AXI4 interconnects). In the IT, DMA channels read and write to the main memory. Additionally, double buffering of input and output buffers in L1 memory allows data transfers to overlap with computation, effectively hiding data transfer latency.

## C. LLM Decoding

1) *Model Architecture Overview*: We use LLaMA 3.1 8B model as an example to illustrate the architecture and computational challenges [16]. This model consists of 32 identical transformer layers; one representative layer is shown in Fig. 4. At a high level, LLM decoding can be categorized into three types of operations:

- Projection operations or Matrix-Vector Multiplication (MVM), which include all projection layers,
- Attention computation, enclosed by the dotted red box,
- Nonlinear functions, including RoPE, RMS normalization, SiLU activation, and element-wise multiplications.

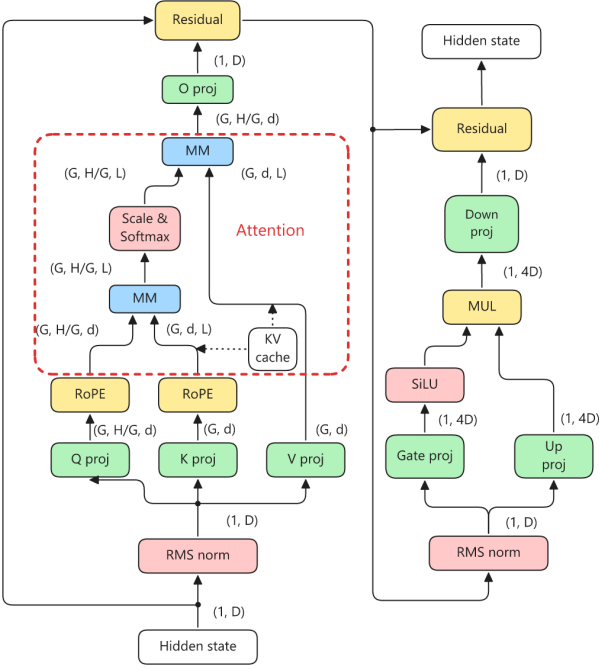


Fig. 4. LLaMA 3.1 8B model architecture (one transformer layer shown). Abbreviations: **proj** = projection, **RoPE** = rotary position embedding, **MM** = matrix multiplication, **MUL** = multiplier. Key model parameters:  $D = 4096$  is the model dimension,  $H = 32$  is the number of attention heads, and  $G = 8$  is the number of KV groups. The head dimension  $d = 128$  is calculated as  $d = D/H = 4096/32$ . This  $d$  is the dimensionality of each query ( $q$ ), key ( $k$ ), and value ( $v$ ) vector per head.  $L$  represents the sequence length, which increases progressively during autoregressive LLM decoding. The attention module is highlighted inside the dotted red box.

2) *Projection Operation–MVM*: At shorter sequence lengths, the overall computation is dominated by projection, where the weights  $\mathbf{W}$  (e.g.,  $\mathbf{W}_q$ ,  $\mathbf{W}_k$ ,  $\mathbf{W}_v$ , and  $\mathbf{W}_o$ ) are applied to the input vector. On edge devices, these weights are typically stored in low-precision formats—most commonly 4-bit integers (`int4`)—to alleviate off-chip memory bandwidth and capacity constraints.

During LLM decoding, the weights are first transferred to the compute unit (e.g., NPU or GPU), then dequantized back to `bf16` precision before projection is performed. This introduces two distinct operations: dequantization and MVM. The separation of these steps incurs additional memory access and latency overhead, which becomes a performance bottleneck during projection. Addressing this inefficiency is critical for optimizing LLM decoding.

3) *Attention Computation*: Shown in Eq. 1, the attention output at decoding step  $t$ , denoted as  $\mathbf{o}_t \in \mathbb{R}^d$ , is computed as a weighted sum over the value vectors  $\mathbf{v}_j \in \mathbb{R}^d$  from all previous time steps  $j = 1$  to  $t$ . Specifically, the attention weight for each position  $j$  is determined by the dot product between the current query vector  $\mathbf{q}_t \in \mathbb{R}^d$  and each key vector  $\mathbf{k}_j \in \mathbb{R}^d$ , followed by softmax normalization over all previous positions. For simplicity, the scaling factor  $\sqrt{d}$  is omitted here. This produces a probability distribution over prior tokens, reflecting their relative importance to the current token. The full attention

computation is given by:

$$\mathbf{o}_t = \sum_{j=1}^t \frac{\exp(\mathbf{q}_t^\top \mathbf{k}_j)}{\sum_{l=1}^t \exp(\mathbf{q}_t^\top \mathbf{k}_l)} \mathbf{v}_j \in \mathbb{R}^d. \quad (1)$$

Here,  $d$  denotes the dimension of each query, key, and value vector. LLaMA 3.1 8B uses grouped query attention (GQA), where  $H/G = 4$  queries in a KV group share 1 KV group. As decoding proceeds, the number of stored keys and values increases linearly with  $t$ , leading to growing memory access and compute requirements. This makes attention computation the primary performance bottleneck at longer sequence lengths.

4) *Nonlinear functions*: The computational cost of nonlinear functions is comparatively low. Thus, our design will focus on projection and attention operations.

### III. WORKING PRINCIPLES

The core computations in LLM decoding—projection and attention—are both memory-bound, meaning their performance is limited by off-chip memory bandwidth ( $U_{\text{mem}}^{rd}$ ). Maximum throughput is achieved when this bandwidth is fully utilized. We introduce two techniques to address this: FusedMV for projection operations and FlowKV for attention.

#### A. FusedDQP

At a high level, FusedDQP is a technique that combines dequantization (DQ) and projection (P) into a single fused kernel to improve computational efficiency.

1) *Quantization*: `int4` weight quantization is widely used for efficient LLMs deployment, while activations are typically retained in `bf16` or `fp16` precision. This is referred to as W4A16, striking a balance between model size and accuracy. Since Ryzen AI NPU natively supports `bf16`, we adopt `bf16` activations and non-projection weights. Importantly, the dequantization logic in FusedDQP is modular and can be easily adapted to other quantization formats. Therefore, it remains agnostic to the specific quantization scheme used.

Projection weights are quantized in groups—typically of size 32, 64, 128, or 256. Each group contains `int4` weights, along with a corresponding scale factor and zero-point. The dequantization process follows the equation:

$$\hat{w}_i = s_g \cdot (w_{q_i} - z_g) \quad (2)$$

where  $\hat{w}_i$  is the dequantized weight,  $w_{q_i} \in \{0, \dots, 15\}$  is the 4-bit quantized weight,  $s_g \in \mathbb{R}$  is the scale factor for group  $g$ , and  $z_g \in \mathbb{Z}$  is the zero-point for group  $g$ .

2) *Q4NX (Quantized 4-bit NPU eXpress format)*: Different quantization methods use varying group sizes  $g$ , but we adopt  $g = 32$  as it is the smallest commonly used value and can be easily extended to larger groups (e.g., 64, 128). The zero-point  $z_g$  is stored as an `bf16`. Additionally, due to limited on-chip memory, it is not feasible to fully dequantize the projection weight matrix before MVM. Instead, the weight matrix is tiled into quantized blocks of size  $32 \times 256$ . Each such block stores  $256 \times (32 \times 4b + 16b + 16b) = 40,960$  bits, or 5 KB, which translates to an average of 5 bits per weight. Within each block, all quantized `int4` weight values are stored first, followed by

the corresponding scale factors and zero-points. The data is organized in column-major order to avoid runtime transposition overhead and to enable efficient access patterns on the NPU. We refer to this compact and hardware-friendly data format as Q4NX (quantized 4-bit NPU eXpress format).

3) *Projection*: We describe the MVM operation as

$$\mathbf{Y} = \mathbf{W} \times \mathbf{A}, \quad (3)$$

where  $\mathbf{W} \in \mathbb{R}^{M \times K}$  is the projection matrix and  $\mathbf{A} \in \mathbb{R}^{K \times 1}$  is the input activation vector.

We tile  $\mathbf{W}$  into smaller blocks of size  $m \times k = 32 \times 256$  (Fig. 5), denoted as  $\mathbf{w}$ , which matches the quantization block size. Similarly,  $\mathbf{A}$  is divided into small vectors  $\mathbf{a} \in \mathbb{R}^{k \times 1}$ , with  $k = 256$  chosen to align with Q4NX quantization. This naturally partitions the output  $\mathbf{Y}$  into tiles  $\mathbf{y} \in \mathbb{R}^{m \times 1}$ . In the single compute tile (CT) scenario, the first vector  $\mathbf{a}$  is loaded into the CT, and accumulation proceeds via:

$$\mathbf{y}_{\text{acc}} += \text{dequant}(\mathbf{w}) \times \mathbf{a} \quad (4)$$

Importantly, the dequantization in Eq. 4 does not operate on the entire  $\mathbf{w}$  block at once. Instead,  $\mathbf{w}$  is further partitioned into fine-grained sub-blocks of size  $16 \times 8$ , which are streamed into vector processor. For each sub-block, dequantization and MVM are executed in a fused kernel, involving only one load from and one store to L1 memory. This fusion maximizes register reuse and minimizes memory access overhead.

Once a  $\mathbf{w}$  is fully used, subsequent  $\mathbf{a}$  vectors and their corresponding  $\mathbf{w}$  blocks (to the right of the previous  $\mathbf{w}$ ) are fetched and processed until the full  $\mathbf{y}_{\text{acc}}$  is computed (Fig. 5).

When multiple CTs are used, the vector  $\mathbf{a}$  is broadcast to all CTs. Each CT independently fetches its associated quantized  $\mathbf{w}$  block from DRAM and performs MVM in parallel (Fig. 5).

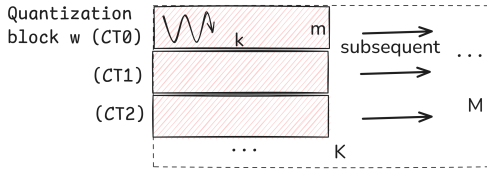


Fig. 5. Tiling strategy for FusedDQP: the weight matrix  $\mathbf{W}$  is partitioned into Q4NX-aligned blocks.

## B. FlowKV

FlowKV is an hardware-aware attention scheme that optimizes KV-cache access to maximize DRAM bandwidth.

1) *Chunked Attention*: During LLM decoding, the full KV cache grows with sequence length, creating pressure on memory bandwidth and on-chip storage. To mitigate this, we adopt a chunk-wise processing approach: the sequence is divided into chunks of size  $L_c$ , and the NPU processes one chunk at a time. Assuming GQA, each KV head is shared by  $H/G$  query heads. Let us consider the attention computation for a single KV head.

Let  $Q \in \mathbb{R}^{H/G \times d}$  be the query matrix corresponding to the  $H/G$  query heads that share this KV head, and  $K, V \in \mathbb{R}^{L_c \times d}$  be the key and value matrices for the current chunk. Here,  $d$

denotes the key/value dimension, and  $L_c$  is the chunk size. Attention is computed incrementally across chunks using a numerically stable accumulation scheme [17], [18]:

### 1) Compute Attention Scores:

$$S = \frac{QK^T}{\sqrt{d}} \quad (5)$$

where  $S \in \mathbb{R}^{H/G \times L_c}$  contains the attention scores for all  $H/G$  query heads in the current chunk.

### 2) Numerical Stabilization:

$$m = \max(S, m_{\text{left}}) \quad (6)$$

where  $m, m_{\text{left}} \in \mathbb{R}^{H/G \times 1}$  store the row-wise maximum scores for the current and previous chunks.

### 3) Exponentiation with Shifting:

$$F = \exp(S - m) \quad (7)$$

where  $F \in \mathbb{R}^{H/G \times L_c}$  contains the softmax numerators for each query head.

### 4) Correction Factor for Prior Chunks:

$$C = \exp(m_{\text{left}} - m) \quad (8)$$

where  $C \in \mathbb{R}^{H/G \times 1}$  is used to align prior accumulations.

### 5) Accumulate Softmax Denominator:

$$l = C \cdot l_{\text{left}} + \sum F \quad (\text{sum over each row}) \quad (9)$$

where  $l, l_{\text{left}} \in \mathbb{R}^{H/G \times 1}$  store the softmax denominators.

### 6) Accumulate Softmax-Weighted Values:

$$Y = C \cdot Y_{\text{left}} + FV \quad (10)$$

where  $Y, Y_{\text{left}} \in \mathbb{R}^{H/G \times d}$  store the accumulated  $FV$ .

### 7) Final Attention Output (after all chunks):

$$O = \frac{Y}{l} \quad (11)$$

where  $O \in \mathbb{R}^{H/G \times d}$  is the final attention output for the group of query heads served by this KV head.

## C. Efficient Mapping to the NPU

As shown in Fig. 6, two CTs are used to implement the FlowKV attention pipeline. CT0 handles steps (1)–(5) of the FlowKV procedure (Sec. III-B1). CT1 performs steps (6)–(7). Intermediate results ( $F$ ,  $C$ , and  $l$ ) are passed directly from CT0 to CT1 via AIE on-chip memory (green arrow in Fig. 6). Both key and value chunks are streamed into the CTs using double buffering.

Ideally, the computation latency for each chunk overlaps with its transfer time, effectively hiding kernel execution overhead, illustrated in Fig. 7. While Fig. 6 illustrates a 2-CT example, the design is flexible: the chunk size  $L_c$ , number of CTs, or number of KV heads can be adjusted to data transfer time and compute intensity towards ideal case to achieve the best performance.

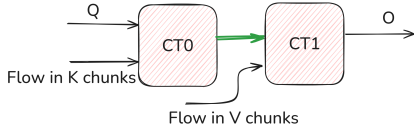


Fig. 6. FlowKV pipeline (2-CT scenario): CT0 streams in key chunks to compute attention scores and passes intermediate results ( $F$ ,  $C$ ,  $L$ ) to CT1. CT1 streams in value chunks and performs softmax-weighted accumulation to produce the final output. This pipeline enables concurrent key and value processing, maximizing NPU bandwidth utilization.

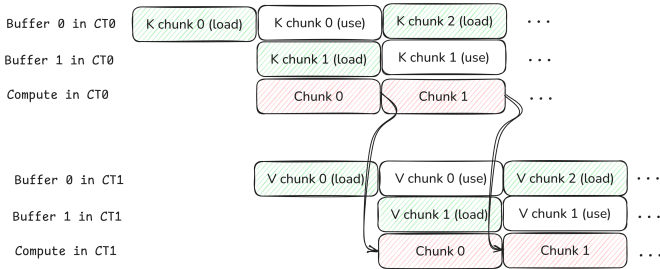


Fig. 7. Timing diagram of the FlowKV pipeline (2-CT configuration) under ideal conditions, where data transfer is fully overlapped with kernel execution in both CTs.

## IV. EXPERIMENTAL RESULTS

### A. Implementation

We implemented LLaMA 3.1 and 3.2 models (1B, 3B, and 8B variants) on the AMD Ryzen AI 5 340 NPU, integrated into the ASRock 4X4 BOX-AI340 mini-PC with SO-DIMM DDR5 5600 MHz [14], [16], [19]. This compact system features a 6-core “Kraken Point” APU, which combines Zen 5 and Zen 5c CPU cores with an XDNA 2-based NPU.

All models are quantized using AWQ (Activation-aware Weight Quantization) with asymmetric `int4` precision on AMD Quark, enabling low-bit inference while preserving accuracy [20], [21]. For projection operations, we allocate four compute tile (CT) columns (16 CTs) to the FusedDQP kernel.

For attention computation, two CT columns (8 CTs) are assigned to the FlowKV kernel. In this configuration, every two CTs operate as a pair (as illustrated in Fig. 6). Thus, a total of 4 pairs are formed. Two KV heads mapped to each CT pair.  $L_c$  is 16. This mapping balances the chunk data transfer time with kernel execution latency, enabling efficient pipelined execution across the CTs and maximizing  $U_{mem}^{rd}$ .

One CT is dedicated to each of the nonlinear operations: RoPE, RMS normalization with residual connection, and SiLU activation with MUL (Fig. 4). These functions operate on vector inputs, which are temporarily stored in MTs between compute stages. All nonlinear kernels are fully pipelined, and their execution incurs negligible latency compared to projection and attention operations during LLM decoding.

The full implementation and low-level kernel optimization are carried out using AMD’s AIE-MLIR infrastructure along with the IRON Python interface [8], [9]. These tools enable fine-grained control over tile-to-tile communication, DMA scheduling, and register allocation, which are critical

for achieving high performance on the AMD Ryzen AI NPU architecture. Link: (\*The GitHub link is currently hidden to preserve anonymity during the review process.)\*

### B. Evaluation and Benchmarking Results

1) *Generation Speed*: We evaluated the generation speed of LLM decoding using our proposed method across various sequence lengths, ranging from 1K to 128K (131,072) tokens, which is the maximum supported sequence lengths. For benchmarking, we compared our method against the SOTA NPU solution provided by AMD Ryzen AI Software 1.4 [12] via GAIA [22] and Lemonade [23]. Additionally, we conducted benchmarks using the iGPU [12] and the CPU with LM Studio [24], all on the same processor. Our results demonstrate that our solution consistently outperforms the SOTA NPU solution across all sequence lengths, shown in Fig. 8. The speedup is calculated in Table I. Furthermore, our design surpasses both the iGPU and CPU implementations at longer sequence lengths.

TABLE I  
SPEEDUP OF THE PROPOSED DESIGN OVER OTHER BACKENDS AT VARIOUS SEQUENCE LENGTHS

Model	Speedup Over	1K	2K	4K	8K	16K	32K	64K	128K
Llama-3.2-1B	NPU (SOTA)	2.0x	2.4x	3.2x	4.5x	6.3x	8.6x	10.7x	12.3x
	iGPU	1.3x	1.9x	3.0x	4.9x	7.6x	10.8x	13.9x	16.2x
	CPU	0.7x	0.7x	0.8x	0.9x	1.0x	1.2x	1.4x	1.5x
Llama-3.2-3B	NPU (SOTA)	1.8x	2.5x	3.5x	5.2x	7.6x	10.3x	12.6x	14.2x
	iGPU	0.7x	0.8x	1.0x	1.4x	1.8x	2.3x	2.8x	3.1x
	CPU	0.7x	0.7x	0.8x	0.9x	1.1x	1.2x	1.3x	1.4x
Llama-3.1-8B	NPU (SOTA)	1.2x	1.6x	2.3x	3.4x	5.3x	7.8x	10.7x	13.3x
	iGPU	0.7x	0.8x	0.9x	1.2x	1.7x	2.3x	3.0x	3.7x
	CPU	0.7x	1.0x	0.9x	1.0x	1.0x	1.3x	1.6x	1.8x

All baselines used identical prompts, context lengths, and decoding parameters. Our method and the official AMD NPU backend (Ryzen AI SW) ran on the same AWQ `int4` weights, while CPU/iGPU baselines (LM Studio v0.3.16) used the closest GGUF 4-bit quantized models. The SOTA NPU backend only supports up to 2K tokens, so results beyond that are projected. All runs used the same NPU power mode (Performance) with fixed clocks and monitored thermals.

2) *Power Consumption*: Table II shows that our NPU solution is consistently more power-efficient than all other backends across LLaMA-3 model sizes. Power consumption, measured using HWINFO [25], indicates up to 2.66× efficiency improvement over the SOTA NPU and over 10× compared to iGPU and CPU backends. These results highlight the suitability of our design for low-power, edge deployment scenarios.

3) *Generalizability*: We validated our approach on the Qwen3 family [26], and Table III shows similar gains as with LLaMA, confirming that FusedDQP and FlowKV generalize beyond a single model family.

## V. DISCUSSION

The proposed method executes the exact LLM models without any algorithmic changes, thereby preserving accuracy. Both FusedDQP and FlowKV (8B model) have a theoretical sustained read memory bandwidth utilization ( $U_{mem}^{rd}$ ) of 57.6 GB/s (constant, not average) when the NPU runs at 1.8 GHz.

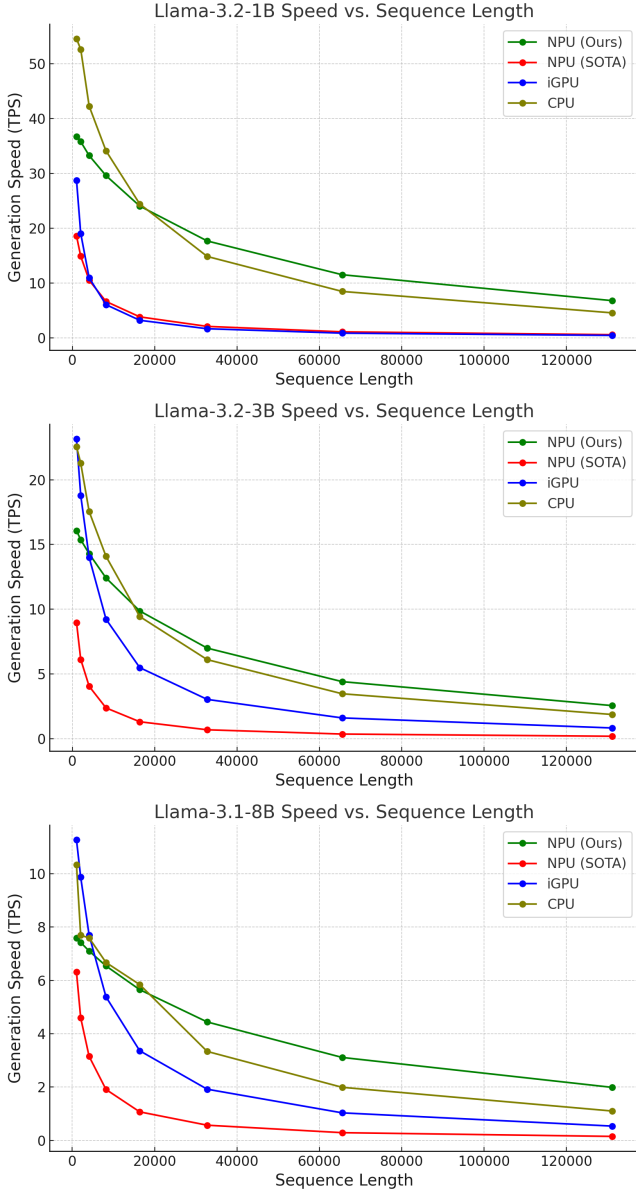


Fig. 8. (Top) 1B model generation speed (TPS: Token Per Second). (Middle) 3B. (Bottom) 8B. All models are evaluated on NPU, iGPU, and CPU in the same processor.

TABLE II  
POWER CONSUMPTION (W) COMPARISON ACROSS DIFFERENT HARDWARE BACKENDS DURING LLM DECODING.

Model	Method	CPU	NPU	iGPU	Total	Efficiency Gain
Llama-3.2-1B	NPU (Ours)	0.07	1.57	0	<b>1.64</b>	–
	NPU (SOTA)	0.85	2.05	0	2.90	1.77×
	iGPU	0.12	0	14	14.12	8.61×
	CPU	4.90	0	0	4.90	2.99×
Llama-3.2-3B	NPU (Ours)	0.06	1.33	0	<b>1.39</b>	–
	NPU (SOTA)	0.95	2.05	0	3.00	2.16×
	iGPU	0.11	0	13	13.11	9.43×
	CPU	4.50	0	0	4.50	3.24×
Llama-3.1-8B	NPU (Ours)	0.07	1.17	0	<b>1.24</b>	–
	NPU (SOTA)	0.80	2.50	0	3.30	2.66×
	iGPU	0.11	0	14	14.11	11.38×
	CPU	4.50	0	0	4.50	3.63×

TABLE III  
GENERATION SPEED (TPS) OF QWEN 3 MODELS.

Model	1k	2k	4k	8k	16k	32k
Qwen-3-0.6B	50.4	45.3	36.0	25.7	16.4	9.6
Qwen-3-1.7B	27.8	26.1	22.7	18.3	13.1	8.3
Qwen-3-4B	14.0	13.3	11.9	10.1	7.7	5.3
Qwen-3-8B	8.1	7.9	7.4	6.6	5.5	4.1

$U_{\text{mem}}^{rd}$  scales linearly with clock speed. On a specific hardware, actual NPU memory bandwidth depends on vendor’s configuration. If actual read memory bandwidth falls below 57.6 GB/s, the system becomes memory-bound; otherwise, it is compute-bound. The design is fully compatible with all XDNA2 NPUs. Notably, both FusedDQP and FlowQKV enjoy linear scalability with the number of CTs. Hypothetically, scaling CT columns from 4 to 8—adding 16 CTs—for FusedDQP would increase its theoretical  $U_{\text{mem}}^{rd}$  to 115.2 GB/s. Similarly, scaling CT columns from 2 to 4 for FlowQKV would increase its  $U_{\text{mem}}^{rd}$  to 115.2 GB/s.

The NPU baseline kernel (Ryzen AI SW) is closed-source, but based on behavior we speculate it relies on a standalone MVM accelerator rather than fused kernels. Its dataflow efficiency appears underutilized, and significant portions of the workload are offloaded to the CPU, leading to both performance gaps and power inefficiency.

CPU/iGPU perform better at short sequences thanks to higher allocated memory bandwidth for MVM, but our design excels from 2K tokens onward as attention dominates; future NPUs with larger allocated bandwidth will widen this advantage.

While demonstrated on AMD Ryzen AI NPUs, FusedDQP and FlowKV apply broadly to CGRA-style chips with SRAM and DMA, and can extend to other AMD AIE fabrics as well.

## VI. CONCLUSION

We introduce a hardware-aware LLM decoding framework on AMD Ryzen AI NPUs, featuring two key contributions: FusedDQP, which integrates dequantization with projection to minimize memory traffic and latency, and FlowKV, a bandwidth-efficient, pipelined KV cache scheduler designed for multi-tile NPU execution. Without modifying the model architecture or accuracy, our implementation on AMD Ryzen AI 5 340 processor achieves up to 14.2× speedup and 2.66× power efficiency improvement compared to the state-of-the-art NPU backend. Furthermore, the design scales linearly with CT count and maintains robust throughput across model sizes ranging from 1B to 8B parameters. These results demonstrate that efficient, accurate LLM inference on edge-class NPUs is feasible using targeted low-level optimizations without architectural compromise.

## ACKNOWLEDGMENT

This work was supported in part by ONR under grant N000142412623, NSF under grant IIS-2435833, and NSF under grant EECS-2530406.

## REFERENCES

- [1] Exxact Corporation, “Why Edge AI Inferencing is Crucial,” <https://www.exxactcorp.com/blog/deep-learning/why-edge-ai-inferencing-is-crucial>, April 2025, accessed: 2025-05-27.
- [2] D. Xu, H. Zhang, L. Yang, R. Liu, G. Huang, M. Xu, and X. Liu, “Fast On-device LLM Inference with NPUs,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 445–462. [Online]. Available: <https://doi.org/10.1145/3669940.3707239>
- [3] Laptop Mag Editors. (2024) Best AI PCs in 2024. Laptop Mag. Accessed: 2025-05-27. [Online]. Available: <https://www.laptopmag.com/laptops/best-ai-pcs>
- [4] Microsoft. (2024) Copilot+ PCs developer guide. Microsoft. Accessed: 2025-05-27. [Online]. Available: <https://learn.microsoft.com/en-us/windows/ai/npu-devices/>
- [5] A. Rico *et al.*, “AMD XDNA NPU in Ryzen AI Processors,” *IEEE Micro*, vol. 44, no. 6, pp. 12–22, 2024. [Online]. Available: <https://ieeexplore.ieee.org/document/10592049>
- [6] A. Rosti and M. Franz, “Unlocking the AMD Neural Processing Unit for ML Training on the Client Using Bare-Metal-Programming Tools,” *arXiv preprint arXiv:2504.03083*, 2025. [Online]. Available: <https://arxiv.org/abs/2504.03083>
- [7] Advanced Micro Devices, Inc. (2024) Rialto Video Overview. AMD. Accessed: 2025-05-27. [Online]. Available: [https://riallto.ai/ryzenai\\_video\\_overview.html](https://riallto.ai/ryzenai_video_overview.html)
- [8] AMD/Xilinx, “mlir-ai: An MLIR-based toolchain for AMD AI Engine-enabled devices,” <https://github.com/Xilinx/mlir-ai>, 2025, accessed: 2025-05-27.
- [9] E. Hunhoff, J. Melber, K. Denolf, A. Bisca, S. Bayliss, S. Neuendorffer, J. Fifield, J. Lo, P. Vasireddy, P. James-Roxby, and E. Keller, “Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface,” in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 85–94. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/FCCM62733.2025.00043>
- [10] J. Zhuang, S. Xiang, H. Chen, N. Zhang, Z. Yang, T. Mao, Z. Zhang, and P. Zhou, “ARIES: An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines,” in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 92–102. [Online]. Available: <https://doi.org/10.1145/3706628.3708870>
- [11] Z. Xu, M. Yu, Y. Zhang, J. Cai, Q. Yang, and T. Wei, “Tile-Level Pipeline for Linear Scalable Stencil Computation on AMD AI Engines,” in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 172–178. [Online]. Available: <https://doi.org/10.1145/3706628.3708822>
- [12] AMD. (2025) Release Notes — Ryzen AI Software 1.4 Documentation. Advanced Micro Devices, Inc. Accessed: 2025-05-27. [Online]. Available: <https://ryzenai.docs.amd.com/en/latest/relnotes.html>
- [13] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, P. Rodriguez *et al.*, “LLaMA 3: Open Foundation and Instruction Models,” *Meta AI Blog*, 2024, accessed: 2025-05-27. [Online]. Available: <https://ai.meta.com/blog/meta-llama-3/>
- [14] AMD. (2025) AMD Ryzen AI 5 340 Processor. Advanced Micro Devices, Inc. Accessed: 2025-05-27. [Online]. Available: <https://www.amd.com/en/products/processors/laptop/ryzen/ai-300-series/amd-ryzen-ai-5-340.html>
- [15] AMD. (2024) RyzenAI-1.4\_LLM\_NPU\_Models. Advanced Micro Devices, Inc. Accessed: 2025-05-27. [Online]. Available: <https://huggingface.co/collections/amd/ryzenai-14-llm-npu-models-67da3494ec327bd3aa3c83d7>
- [16] Meta AI, “LLaMA 3: Open Foundation and Instruction Models,” <https://ai.meta.com/blog/meta-llama-3/>, 2024, accessed: 2025-05-27.
- [17] M. Milakov and N. Gimelshein, “Online normalizer calculation for softmax,” *CoRR*, vol. abs/1805.02867, 2018. [Online]. Available: <http://arxiv.org/abs/1805.02867>
- [18] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 16 344–16 359.
- [19] ASRock Industrial, “ASRock 4X4 BOX-AI340 Mini-PC,” <https://www.asrockind.com/en-gb/4X4%20BOX-AI340>, 2025, accessed: 2025-05-29.
- [20] 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*, P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., vol. 6, 2024, pp. 87–100.
- [21] Advanced Micro Devices, Inc., “AMD Quark: Cross-Platform Deep Learning Model Quantization Toolkit,” <https://quark.docs.amd.com/>, 2025, version 0.8.1. Accessed: 2025-05-29.
- [22] AMD, “GAIA: Generative AI Is Awesome,” GitHub repository, 2025, <https://github.com/amd/gaia>.
- [23] lemonade-sdk, “Lemonade: Local llm server with npu acceleration,” GitHub repository, 2025, <https://github.com/lemonade-sdk/lemonade>.
- [24] Element Labs, Inc., “LM Studio: Your Local AI Toolkit,” <https://lmstudio.ai/>, 2025, version 0.3.16. [Online]. Available: <https://lmstudio.ai/>
- [25] REALiX Corp., “HWiNFO: System Information and Diagnostics Tool,” <https://www.hwinfo.com/>, 2024, accessed: 2025-05-29.
- [26] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, C. Zheng, D. Liu, F. Zhou, F. Huang, F. Hu, H. Ge, H. Wei, H. Lin, J. Tang, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Zhou, J. Lin, K. Dang, K. Bao, K. Yang, L. Yu, L. Deng, M. Li, M. Xue, M. Li, P. Zhang, P. Wang, Q. Zhu, R. Men, R. Gao, S. Liu, S. Luo, T. Li, T. Tang, W. Yin, X. Ren, X. Wang, X. Zhang, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Zhang, Y. Wan, Y. Liu, Z. Wang, Z. Cui, Z. Zhang, Z. Zhou, and Z. Qiu, “Qwen3 technical report,” *arXiv preprint arXiv:2505.09388*, 2025.