Execution Strategies for Transformer Layers on the Ryzen NPU

by

Curt John Bansil

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2026 by the
Graduate Supervisory Committee:

Aman Arora, Co-Chair
Aviral Shrivastava, Co-Chair
Joseph Melber

ARIZONA STATE UNIVERSITY
May 2026

ABSTRACT

Large Language Model (LLM) inference is typically served through datacenters, but rising costs of serving have motivated interest in deployment to edge devices. Whether edge deployment is practical depends on how efficiently these devices execute transformer layers, which account for the bulk of computation in LLM inference. Neural processing units (NPUs) are natural targets for edge deployment as they are domain-specific accelerators for AI workloads. This thesis show that efficient LLM deployment is feasible on AMD's Ryzen AI platform, with a focus on the XDNA Neural Processing Unit (NPU), whose software stack makes tiling, placement, and data movement explicit.

The thesis work includes a study on three execution strategies for transformer layers on the NPU: offload, runlist, and dataflow. Additionally, partial-sum staging with pipelined dataflow kernels is proposed as a reusable method for improving intermediate-result reuse, enabled through loop interchange that aligns the loop bounds of the producer stage with those of the consumer stage. In fused Multi-Head Attention with Output Projection (MHAO), staging output-projection partial sums in shared scratchpad yields speedups of up to 7.80x; in the feed-forward network (FFN), staging down-projection partial sums yields speedups of up to 6.99x. Lastly, a hybrid execution flow that combines dataflow kernels with ordered runlist dispatch is implemented, composing transformer layer execution as a short sequence of efficient coarse-grain kernels.

The evaluation covers three BERT-style encoder families and two GPT-2-style decoder families across sequence lengths from 64 to 16,384 tokens. Comparisons are made against a naive runlist baseline on the NPU as well as an iGPU baseline. End to end, the hybrid implementation reduces latency by 1.27× to 2.63× relative to the naive runlist baseline. With the iGPU baseline, the hybrid NPU implementation becomes more energy efficient at sequence lengths of 1,024 tokens and above for four model families, and at 2,048 tokens and above for all five.

The results demonstrate that efficient LLM deployment on the Ryzen AI platform is feasible, and that the choice of execution strategy strongly influences the performance of LLM inference.

ACKNOWLEDGMENTS

I would like to thank my co-chairs, Dr. Aman Arora, Assistant Professor of ASU School of Computing and Augmented Intelligence, and Dr. Aviral Shrivastava, Professor of ASU School of Computing and Augmented Intelligence, for their continuous support and patience throughout this thesis process. Being able to meet and discuss any issues I encountered weekly helped to keep me pushing through this work. Without them, I would not have been able to complete this work.

I would also like to thank Dr. Joseph Melber, Senior Member of Technical Staff of AMD Research and Development, for his technical support and consistent input throughout the process. Without his participation and input, the implementations and analysis of the work would not have been successfully conducted.

Thirdly, I would like to acknowledge Kaustubh Mhatre, Ph.D. in Computer Science at Arizona State University, and I am grateful for his valuable advice for the thesis.

Lastly, I would like to acknowledge family, because I love them.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER 1

INTRODUCTION

1.1 Motivation

Serving LLM inference through datacenters is already costly and is expected to become more expensive as AI adoption rises (Noffsinger *et al.*, 2025). McKinsey estimates that meeting global demand for compute power may require nearly $7 trillion in data-center capital outlays by 2030 (Noffsinger *et al.*, 2025). Energy demand is rising as well: the IEA projects that electricity generation for data centers will grow from 460 TWh in 2024 to over 1,000 TWh in 2030 (International Energy Agency, 2025). These costs make edge inference an attractive alternative. The central question, however, is whether transformer layers can be executed efficiently enough on edge processors to make local inference practical.

Neural processing units (NPUs) are designed for processing AI workloads and are strong candidates for edge inference. AMD's Ryzen AI platform is particularly relevant for edge deployment, given its commercial availability, and its open software stack for programming the XDNA NPU. The Ryzen NPU is a spatial array based on the AI Engine-ML architecture and is programmed through AMD's IRON toolchain (Melber *et al.*, 2024; Hunhoff *et al.*, 2025). Unlike a fixed-function deployment stack, this toolchain leaves tiling, data movement, and runtime boundaries explicit. The execution strategy employed to run inference therefore must be carefully consdiered in order to achieve efficient performance.

1.2 Problem 1: Multiple Execution Strategies on the NPU

Transformer layers can be executed on the Ryzen NPU in more than one way. Broadly, we categorize them into three options: *offload*, *runlist*, and *dataflow*. The offload strategy executes one heavily optimized LLM operation on the NPU—most naturally a GEMM—while the remaining operations execute on the host or another accelerator. The runlist strategy dispatches an ordered list of LLM operations to the device and removes the need for host–NPU synchronization during execution. The dataflow strategy maps LLM operations through the processing elements in the device, typically in a pipelined fashion, and lowers data movement through global memory.

These strategies differ in dispatch structure, data movement, reconfiguration overhead, and how much intermediate state remains on-chip. Their relative performance depends on workload parameters such as sequence length and model size. Offload incurs significant data movement overhead, runlist incurs high reconfiguration overhead, and pure dataflow becomes impractical for larger workloads because on-chip memory is limited. This motivates hybrid execution implementations that combine the strengths of individual strategies.

1.3 Problem 2: Preserving Reuse in Tiled Pipelines

Pipelining aligns naturally with the Ryzen NPU. Direct Memory Access units (DMAs) at each processing element can overlap data movement and compute, and intermediate data can be double-buffered in local scratchpads. Transformer layers, however, complicate this ideal execution pattern. Local scratchpads are limited in capacity, and the data required to process transformer workloads cannot fully fit within them. Tiling is therefore necessary.

Once execution is forced into tiled data, the key issue is temporal reuse distance rather than overlap alone. A stage may produce exactly the tile required by the next stage, but if a downstream operator cannot keep that tile on-chip long enough, the implementation

must drain it, reload it, or recompute it. Efficient pipelined execution on the Ryzen NPU therefore depends on techniques that preserve reuse of intermediate tiles on-chip.

1.4 Proposal Overview

Our work addresses these two prolbems. We propose a methodology for preserving reuse of intermediate results in pipelined dataflows and a hybrid execution that combines dataflow kernels with ordered runlist dispatch for transformer workloads. Concretely, we fuse Query, Key, and Value projections (QKV-proj), multi-head attention with output projection (MHAO), feed-forward network (FFN), and residual add with layer normalization (AN) into pipelined dataflow blocks and dispatch them through the runlist interface.

1.5 Contributions

We thus make three contributions:

- Propose partial-sum staging, a reusable dataflow-oriented method for preserving intermediate-result reuse in tiled pipelines. In MHAO, the staged state is the output-projection accumulation. In FFN, loop interchange aligns up-projection and down-projection loop bounds, and the staged state is the down-projection accumulation.
- Implement a hybrid transformer-layer execution that combines dataflow kernels with ordered runlist launch to compose transformer layer execution as a short sequence of efficient coarse-grain kernels.
- Evaluate the hybrid implementation across five transformer-layer families—three BERT-style encoders and two GPT-2-style decoders—over sequence lengths from 64 to 16,384, with quantitative comparison against a naive runlist NPU baseline and an iGPU baseline.

1.6 Organization of the Thesis

Chapter 2 provides the background needed for the study, including transformer-layer structure, Ryzen AI architecture, the software stack, and the tiled-execution constraints that motivate the work. Chapter 3 positions the thesis relative to prior work on AMD NPU programming, dataflow pipelining, GEMM mapping, and edge transformer deployment. Chapter 4 defines the execution strategies used throughout the thesis. Chapter 5 presents the proposed methodology, including partial-sum staging and the kernels that form the hybrid implementation. Chapter 6 describes the experimental setup, measured surface, metrics, and power methodology. Chapter 7 reports the evaluation results. Chapter 8 discusses deployment, software, and hardware implications together with future directions. Chapter 9 concludes.

CHAPTER 2

BACKGROUND

2.1 Workload Characterization

Transformer layers comprise of linear projections, attention, residual connections, normalization, and a feed-forward network (Vaswani *et al.*, 2017). Our work focuses on BERT encoder layers and GPT-2 decoder layers, which capture the core structures used in modern LLMs. Figure 2.1 shows the encoder and decoder execution graphs, and Table 2.1 summarizes the model families used for evaluation. In both graphs, the hidden state is passed in as input, and a hidden state is produced as output.

With the BERT graph, the hidden state is projected to query (Q), key (K), and value (V), which feed multi-head attention (MHA). The dashed MHA region in Figure 2.1 contains attention-score calculation, softmax, and the attention weights times value (PV) calculation. An output (O) projection follows, then residual add and layer normalization. Finally, a separate feed-forward network (FFN) block operates on that normalized output, followed by a second residual add and layer normalization.

In the GPT-2 decoder graph, the first layer normalization precedes the Q/K/V projections, a causal mask is applied to the attention scores, and the FFN tail ends with only a residual add (Radford *et al.*, 2019).

These structural differences affect the runlist dispatch order discussed later.

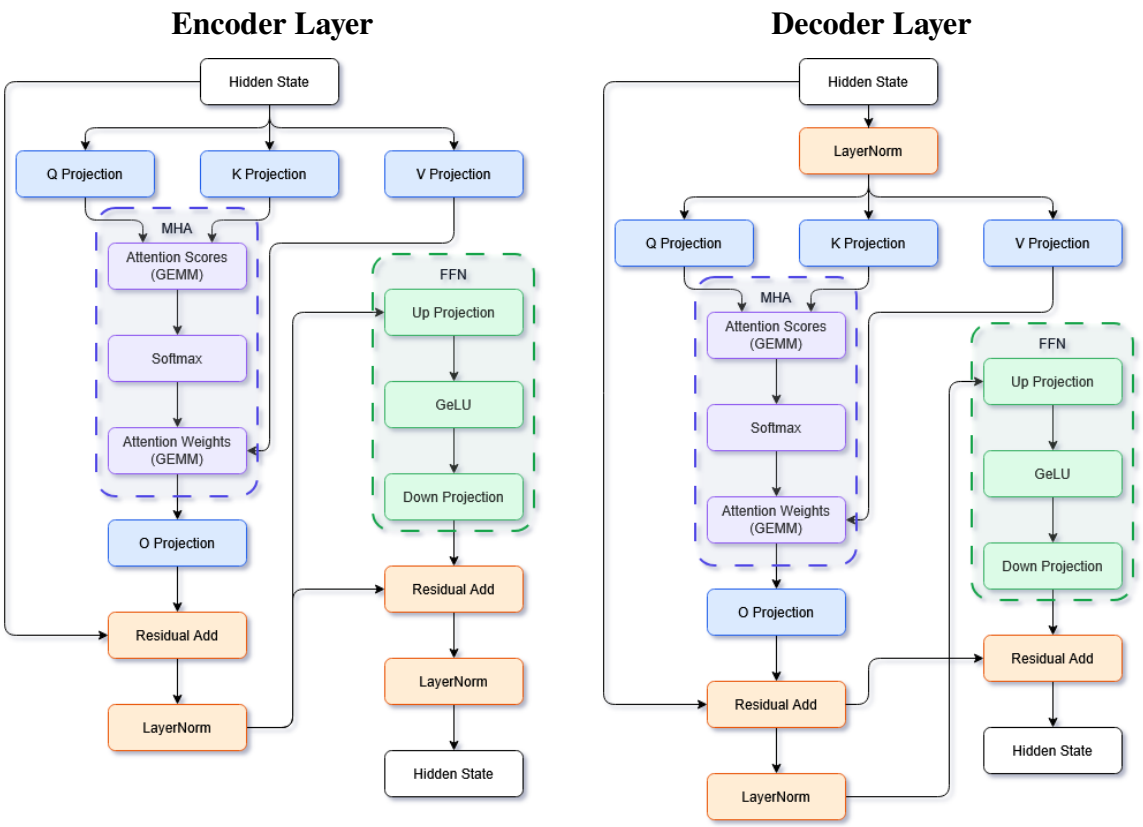

Figure 2.1: Encoder and decoder layer graphs. The BERT encoder path has two instances of residual add followed by layer normalization, while the GPT-2 decoder path has the first normalization before projection and only residual add after the feed-forward network. Both layers take in a hidden state as input and generate a hidden state as output.

We define the following variables:

$$S := \text{sequence length} \tag{2.1}$$

$$h := \text{number of heads} \tag{2.2}$$

$$d_{\text{head}} := \text{head dimension} \tag{2.3}$$

$$d_{\text{emb}} := \text{embedding dimension} = h \cdot d_{\text{head}} \tag{2.4}$$

$$d_{\text{ffn}} := 4\, d_{\text{emb}} \tag{2.5}$$

Table 2.1: Encoder and decoder layer workload parameters used in the thesis. Here, d_{emb} is the embedding dimension, d_{ffn} is the FFN dimension, and h is the number of attention heads.

Family	Layer Type	d_{emb}	d_{ffn}	Heads (h)
TinyBERT	Encoder	512	2048	8
BERT-Base	Encoder	768	3072	12
BERT-Large	Encoder	1024	4096	16
GPT-2 Small	Decoder	768	3072	12
GPT-2 Medium	Decoder	1024	4096	16

For both encoder and decoder layers, the total multiply-and-accumulate (MAC) count is

$$\text{MACs}_{\text{proj}} = 4Sd_{\text{emb}}^2 \tag{2.6}$$

$$\text{MACs}_{\text{MHA}} = 2S^2 d_{\text{emb}} \tag{2.7}$$

$$\text{MACs}_{\text{FFN}} = 2Sd_{\text{emb}}d_{\text{ffn}} \tag{2.8}$$

$$\text{MACs}_{\text{total}} = 4Sd_{\text{emb}}^2 + 2S^2 d_{\text{emb}} + 2Sd_{\text{emb}}d_{\text{ffn}} \tag{2.9}$$

$\text{MACs}_{\text{proj}}$ denotes the total MACs for projections: query, key, value, and output projections. MACs_{MHA} denotes the total MACs for multi-head attention and MACs_{FFN} for the feed-forward network.

With two FLOPs per MAC, the total floating-point operation count for each layer is

$$\text{FLOPs}_{\text{total}} = 8Sd_{\text{emb}}^2 + 4S^2 d_{\text{emb}} + 4Sd_{\text{emb}}d_{\text{ffn}} \tag{2.10}$$

2.2 Workload Scaling Intuition

The operations in a transformer layer do not scale in the same way. The Q/K/V/O projections and the FFN are dominated by dense matrix multiplications whose row dimension is the

sequence length, so total MACs for those operations scale linearly with sequence length. Because $d_{ffn} = 4d_{emb}$ for the BERT and GPT-2 layers implemented, the FFN requires twice as many MACs as the combined Q/K/V/O projections. In contrast, the GEMM operations in MHA scale quadratically with sequence length. As a result, short-sequence layers are often dominated by FFN execution, whereas longer-sequence layers shift become dominated by MHA execution.

2.3 Ryzen AI SoC and the XDNA 2 NPU

Figure 2.2 shows the architecture of the Ryzen AI platform. Ryzen AI integrates a central processing unit (CPU), an integrated graphics processing unit (iGPU), and an XDNA neural processing unit (NPU) on one client SoC. The platform used in this work is an ASUS Vivobook S 15 (M5506) with an AMD Ryzen AI 9 HX 370. For evaluations, we use the NPU and iGPU within the same system.

The NPU is a two-dimensional array of compute tiles, memory tiles, and interface tiles (AMD, Inc., 2024, 2025a). In the XDNA 2 Strix family, the array exposes 32 compute cores (AMD, Inc., 2026c). Each Compute Tile contains a very long instruction word (VLIW) vector processor, a scalar processor, and 64 KB of local scratchpad memory organized into eight banks (AMD, Inc., 2025b). Below the Compute Tiles, a row of Memory Tiles provides a larger shared scratchpad with 512 KB of data memory organized into 16 banks. A row of Interface Tiles below the Memory Tiles connects the NPU to external memory and the surrounding system.

Each Compute, Memory, and Interface Tile includes dedicated direct memory access (DMA) units for overlapping data movement and compute. Compute and Interface Tiles support up to three-dimensional DMA accesses, while Memory Tiles support up to four-dimensional accesses (AMD, Inc., 2025b). DMAs at Compute and Interface Tiles have 2 channels each for streaming data into and out of local memory, while DMAs at Memory

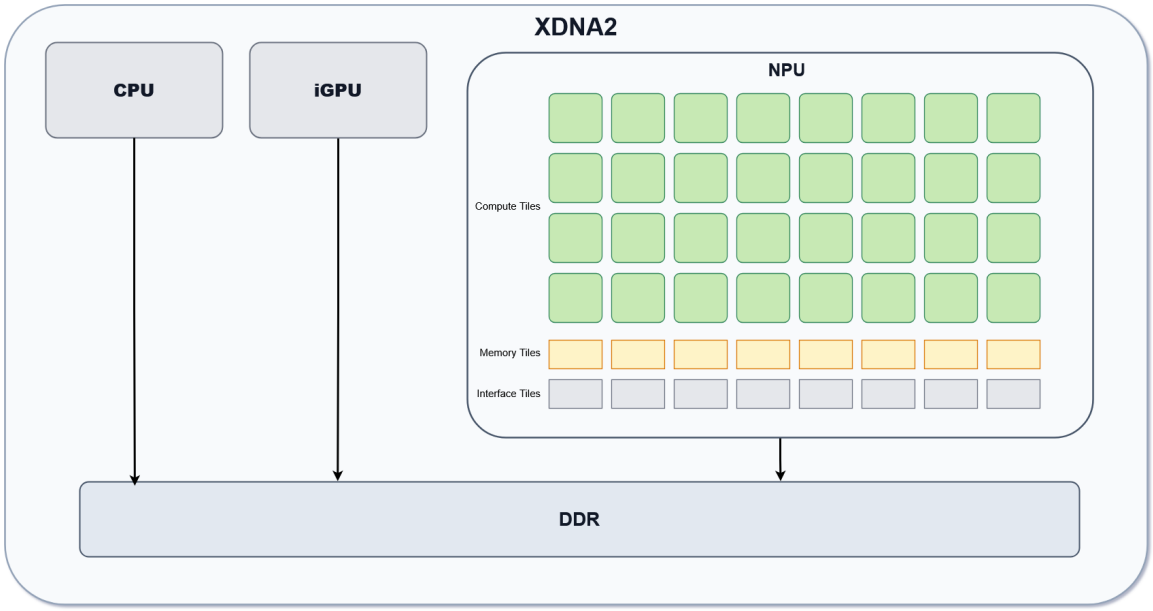

Figure 2.2: Ryzen AI platform organization with the XDNA 2 compute-tile array, memory-tile row, interface tiles, and shared DDR path

Tiles have 6 channels each for streaming data into and out of memory.

bfloat16 (BF16) is the data format used for this work. Under the assumption of 64 BF16 multiply-accumulates (MACs) per cycle per compute core at 1.8 GHz, one core has a BF16 peak of

$$P^{\mathrm{BF16}}_{\mathrm{NPU,core}} = 64 \times 2 \times 1.8 \times 10^{9} = 230.4\ \mathrm{GFLOP/s}, \tag{2.11}$$

where one MAC counts as two floating-point operations. Across 32 compute cores, the corresponding full-array BF16 peak is

$$P^{\mathrm{BF16}}_{\mathrm{NPU}} = 32 \times 230.4\ \mathrm{GFLOP/s} = 7.3728\ \mathrm{TFLOP/s}. \tag{2.12}$$

The iGPU is used as one of the baselines for this work, and the Ryzen AI 9 HX 370 contains an integrated Radeon 890M GPU. The Radeon 890M contains 16 graphics cores at up to 2.9 GHz, which have 512 FLOP/clock/compute-unit with FP16/BF16 matrix instructions on RDNA3 hardware (AMD, Inc., 2026b; Harada and Yoshimura, 2024; AMD,

Inc., 2026a). This gives a BF16 ceiling of

$$P_{\text{iGPU}}^{\text{BF16,WMMA}} = 16 \times 512 \times 2.9 \times 10^9 = 23.7568 \text{ TFLOP/s}. \tag{2.13}$$

Separate TDP values are unavailable for iGPU and NPU. However, for the Ryzen AI 9 HX, the default TDP is 28W and configurable TDP is 15-54W (AMD, Inc., 2026b).

The system used for our evaluations has an AMD Ryzen AI 9 HX 370 with LPDDR5X-7500 memory on a 128-bit interface (ASUSTeK Computer Inc., 2026; AMD, Inc., 2026b). Deshmukh *et al.* (2025) report that Ryzen AI 9 HX platforms allocate 60 GB/s to the NPU and 130 GB/s to the iGPU. That iGPU value likely corresponds to the LPDDR5X-8000 specification of the Ryzen AI 9 HX 375. For the Ryzen AI 9 HX 370 with LPDDR5X-7500, the peak DRAM bandwidth is 120 GB/s. We therefore assume that the NPU is allocated 60 GB/s of DRAM bandwidth, and the iGPU is allocated 120 GB/s of DRAM bandwidth.

2.4 Programming Stack, Artifacts, and Hardware Contexts

Ryzen AI exposes a compiler/runtime stack through IRON, MLIR-AIE, and XRT (Melber *et al.*, 2024; AMD Xilinx, 2025a; Hunhoff *et al.*, 2025). IRON provides Python APIs for describing compute placement and data movement, while compute kernels themselves are written in C++. A Python script written with IRON generates an intermediate representation in the mlir-aie dialect, which can be lowered through the MLIR-AIE compilation flow into device binaries and command streams used by the runtime.

Two runtime-visible artifact types matter here. An `xclbin` captures the device-side static configuration, while the command stream describes the instructions needed to configure tiles and data movement at runtime. The host then launches work through XRT under a hardware context shared between the host and the NPU. That hardware context determines which binaries remain visible, and which reconfiguration boundaries remain exposed to the runtime.

Within this stack, a *runlist* is an ordered list of `xrt::run` objects (AMD Xilinx, 2026b,a). The same runlist mechanism can express either a short chain of coarse-grain kernels, i.e. fused operations, or a long chain of fine-grain kernels, i.e. separate GEMM operations. A runlist therefore controls how much of a transformer graph's structure remains visible to the runtime.

2.5 Tiled Pipelines and Reuse Pressure

Pipelining can be expressed on the NPU by double buffering data and utilizing DMAs to overlap compute and data movement. However, LLM workloads do not fit entirely within on-chip memory, so tiled execution is necessary. The challenge with a tiled pipeline is that intermediate data needed by later stages may not remain on-chip long enough to be reused. As a result, the implementation is may have to revisit earlier computation.

Figure 2.3 illustrates the reuse pressure. The figure visualizes one MHAO example. The left panel shows a four-stage pipeline: attention score calculation (1), softmax (2), PV (3), and output projection (4).

The schedule assumes three heads for MHAO and tile sizes equal to the hidden dimension. Because $d_{emb} = h \cdot d_{head}$, three output tiles from stage 3 are required to generate one output tile in stage 4, and stage 4 must produce three output tiles to cover the full embedding dimension. If at most two tiles per stage can remain on-chip, the iteration 3 replaces the tiles from the iteration 1. Those values must then be regenerated to produce the remaining output-projection tiles.

The right panel shows the same behavior across four processing elements. The horizontal axis denotes space and the vertical axis denotes time. The first three processing elements remain occupied by stages 1–3, while the fourth accumulates three output tiles over 12 time steps, with outputs produced at $t = 6$, $t = 9$, and $t = 12$. This mapping highlights the cost of insufficient on-chip storage: additional iterations are required to recompute intermediate

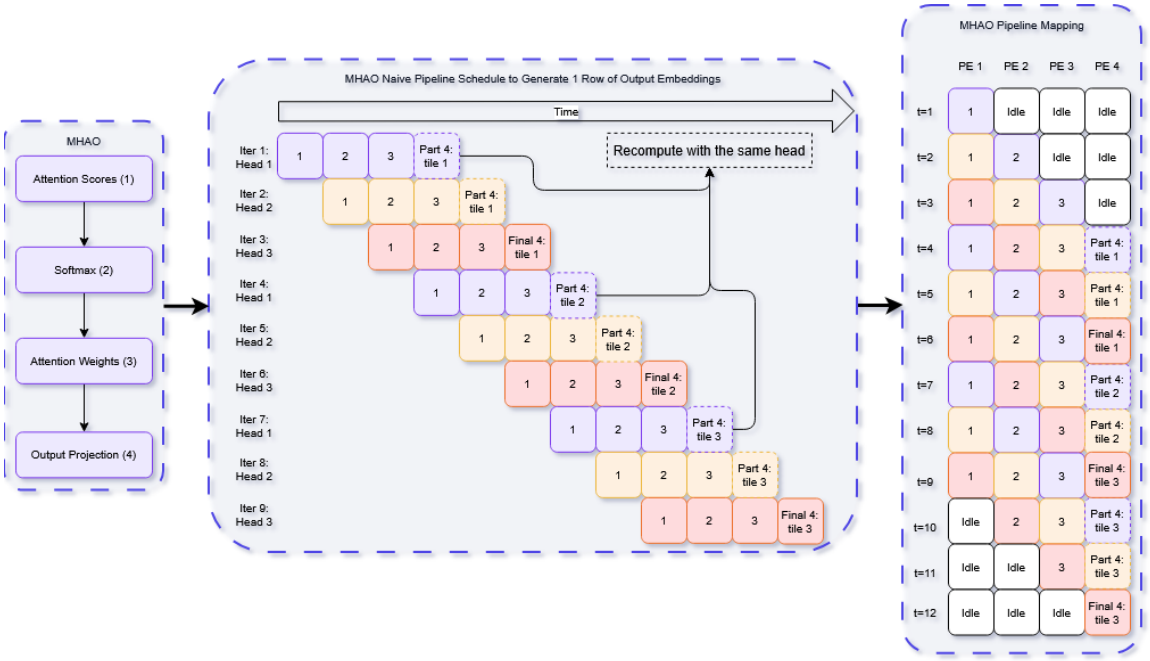

Figure 2.3: Tiled MHAO pipeline. Stages 1–4 denote attention scores, softmax, attention weights, and output projection, respectively. Three iterations of stages 1–3 are needed to generate one output tile for stage 4, which accumulates over the first two iterations. With at most two tiles per stage that can remain on-chip, the third iteration replaces tiles from the first iteration and forces recomputation when generating the output tiles of stage 4 in iterations 4-9.

data, increasing total execution time to 12 time steps.

CHAPTER 3

RELATED WORK

3.1 Programming AMD NPUs

Recent work has improved the software stack for programming AMD NPUs. Hunhoff *et al.* (2025) extend the IRON programming flow with abstractions for compute placement, data movement, and data transformation, improving designer productivity while preserving low-level control. Those IRON abstractions are used throughout this thesis. At a higher level, Wang *et al.* (2025b) present MLIR-AIR, a compiler framework for mapping AI workloads onto AMD NPUs through explicit constructs for asynchronous scheduling, data movement, and spatial placement.

3.2 GEMM Mapping and Kernel Optimization

Because most transformer-layer computation is made up of GEMM operations, work on optimizing GEMM operations with NPUs is highly beneficial. Rösti and Franz (2025) accelerate GPT-2 training by offloading GEMMs to the NPU. Their implementation employs an accumulate-in-place mapping strategy and updates kernel loop bounds at run time to support multiple GEMM shapes with low reconfiguration overhead. Taka *et al.* (2025) study GEMM optimization across NPU generations and show that high performance depends on balancing compute intensity, data movement, and reconfiguration cost. Wang *et al.* (2025a) introduce an asymmetric tile-buffering strategy that reduces buffer pressure and improves arithmetic intensity relative to symmetric buffering. Altogether these works demonstrate

that efficient NPU execution depends strongly on kernel mapping, tiling, and buffering choices.

3.3 Pipelined and End-to-End Mappings

Beyond GEMM kernels, Xu *et al.* (2025) pipeline stencil computation through the NPU array, using ping-pong buffering to overlap compute and data movement while preserving necessary data within local memory of Compute Tiles. Du *et al.* (2026) present an end-to-end mapping of Gemma3 onto the NPU for prefill and decode stages. Through hardware-aware scheduling and bandwidth-centric kernels, they report speedups and energy-efficiency gains over CPU and iGPU baselines. These results show that applications can be mapped effectively to NPUs when execution structure and data movement are co-designed with the hardware.

CHAPTER 4

EXECUTION STRATEGIES

4.1 Offload

The offload strategy focuses on highly optimizing one operation to run on the NPU, with other operation executed on the host (or another accelerator) instead. The natural choice for offloading an operation with LLM inference is the GEMM operation, as they make up the bulk of compute with transformer layers.

One benefit of this strategy is that all 32 Compute Tiles can be dedicated to executing one operation, which (Rösti and Franz, 2025) demonstrated with their mapping of accumulate-in-place GEMM to accelerate GPT=2 training. Their GEMM kernel implementation could be partially reconfigured with low overhead at run time, because the function running at each core was the same–an outer loop over bounded by the total output tiles to generate and an inner loop bounded by the total iterations to accumulate for each output tile. The bounds for those two loops are essentially registers that can be reconfigured at run time, which can be compiled to an instruction binary for each unique GEMM workload and can be loaded to the command processor. Thus, the GEMM kernel can be adjusted to process different workload sizes by using the same xclbin and loading the relevant instruction binary to modify two registers at each Compute Tile for a GEMM workload, requiring little overhead for reconfiguration.

One drawback of this strategy is that by only executing one operation on the NPU, the other operations executed on the host will result in increased energy consumption during

the end-to-end run. Another drawback is that switching between host and NPU compute incurs synchronization and launch overheads. Synchronization cost is incurred since the NPU only sees global memory–it has no cache-coherent view of data in the caches. As a result, there's a need to make sure that the NPU "sees" the data in the cache. With the Ryzen NPU, this is done by explicitly flushing or invalidating data in the caches. This is taken care of in the userspace rather than the operating system, and is a high-latency operation (AMD Xilinx, 2025b).

Figure 4.1 shows an example of the offload strategy. For both BERT (encoder) and GPT-2 (decoder) graphs, the GEMM operations are offloaded to the NPU. The remaining operations in the graphs are executed on the NPU. Between each switch from CPU and NPU compute and vice versa, synchronization is required since the NPU only sees global memory. Additionally, for each time the NPU has to process a GEMM workload, launch overheads from starting the NPU and the command processor reading the instruction buffer to reconfigure the NPU are incurred.

The offload strategy allows a developer to focus on one optimized computation at the cost of synchronization and launch overheads. In order to lower these costs, more of the transformer graph needs to be packaged to execute on the NPU, which requires the use of a different execution strategy.

4.2 Runlist

The runlist strategy implements the transformer graph more directly. Operators are stitched together into an ordered list of kernels to execute sequentially on the NPU, and the host submits this list to the runtime. The benefit of this strategy is that it removes the need for synchronization between host and NPU for the intermediate nodes in the submitted graph–it's now only required at the start and end of the graph. Moreover, all 32 Compute Tiles are available to implement each kernel.

The naive runlist baseline used later in the evaluation is the simplest instance of this strategy. Figure 4.2 shows the naive runlist impelemntation. Each operator in the transformer graph is configured and executed separately on the NPU. One drawback is that every operator in the list still incurs own launch overhead. Additionally, the NPU reconfiguration cost is high, even when some operators have much lower arithmetic intensity compared to the GEMM operations in the graph. The overhead of reconfiguration may not be worth incurring to run some operations by themselves on the NPU, so operator fusion becomes a consideration for such operations.

4.3 Dataflow

The dataflow strategy fuses operations into one NPU kernel. Instead of having every operator executed by the runtime, a subgraph of the transformer graph, or even the full graph itself, are handled within the NPU array. Utilizing this strategy reduces the reconfiguration overhead incurred by the runlist strategy. And since the DMAs combined with double buffered data can be used to implement pipelines on the NPU, mapping these operators in a pipelined fashion means the potential for higher throughput.

One drawback is that each operation no longer has access to all 32 cores for compute, which was a benefit with the offload and runlist strategies. Another drawback is that a balanced pipeline is difficult to implement when operations scale differently during execution–one example being that mult-head attention scales quadratically with sequence length, while projections scale linearly. An imbalanced pipeline will lead to higher latency with execution. A third drawback is that limited on-chip memory means pipelining needs careful mapping to avoid repeated compute, as discussed in Chapter 2.

Figure 4.3 demonstrates an example implementation of the dataflow strategy. The left panel is a short pipeline with three stages. The center panel shows a pipeline schedule in which stages 1, 2, and 3 overlap rather than waiting for one another to finish sequentially.

The right panel shows the spatial mapping over twelve processing elements, where different stages occupy different processing elements over time.

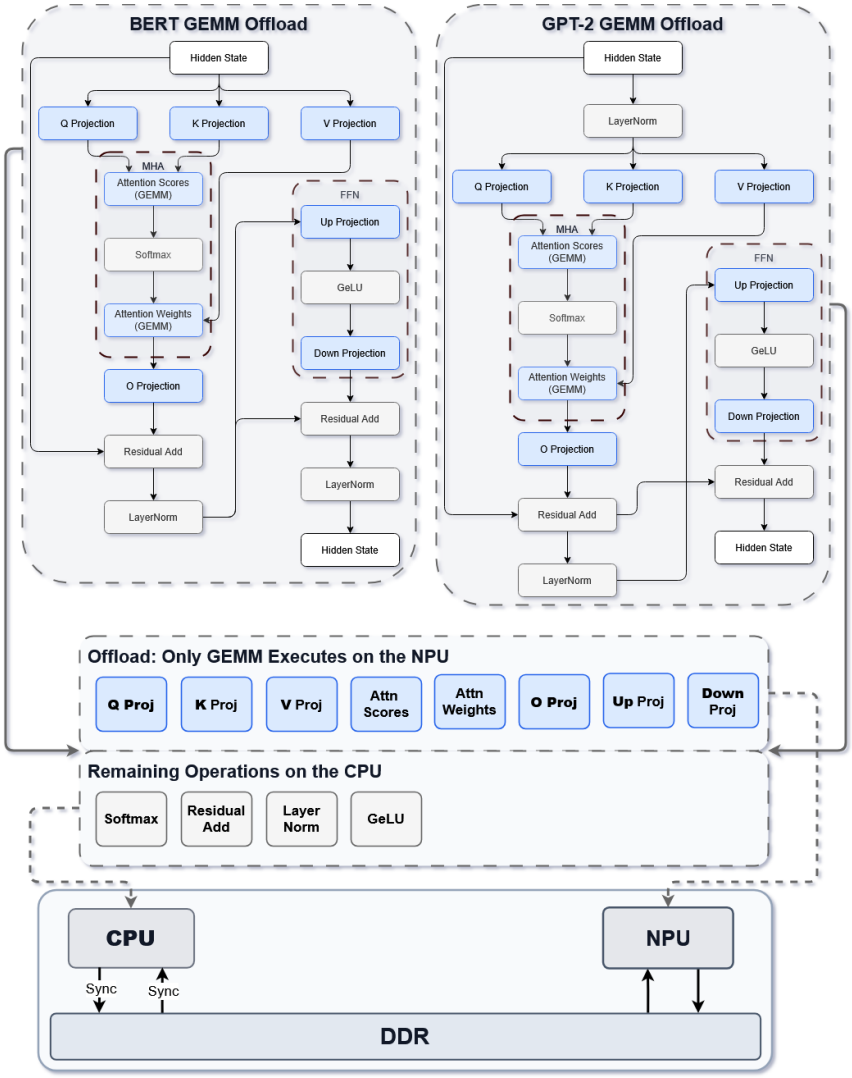

Figure 4.1: The offload execution strategy example, where blue nodes in the transformer graphs indicate a GEMM workload executed on the NPU, and gray nodes in the graphs indicate an operation executed on the CPU. When switching between CPU and NPU during the run, synchronization is required, e.g. GeLU between up and down projections in FFN.

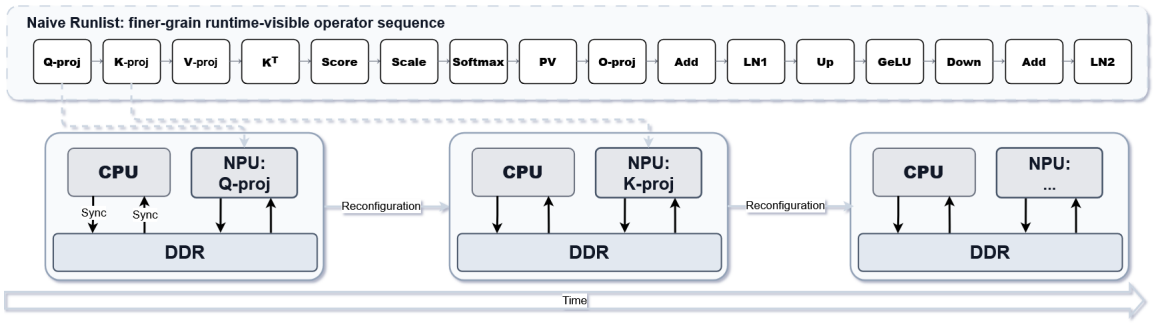

Figure 4.2: Runtime-visible structure for a naive runlist. Each operator in the transformer graph is implemented as a separate kernel that is executed on the NPU. At the start and end of the runlist, CPU synchronization is required, but is not required for the operations in between. After one operation is executed, the NPU must be reconfigured for the next operation, thus incurring reconfiguration overhead per individual operation in the runlist.

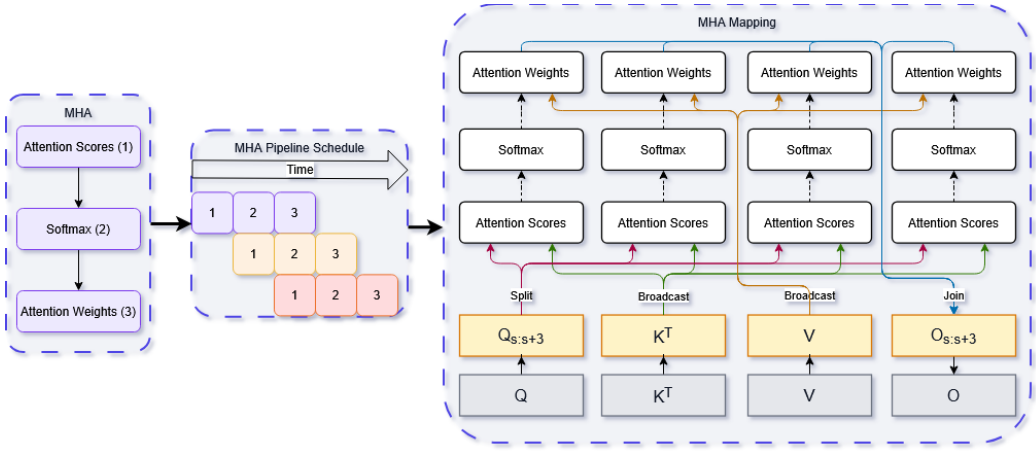

Figure 4.3: Dataflow example implementing a subgraph of the transformer graph, multi-head attention. The left shows the nodes in the subgraph, while the center shows how those operations are scheduled on the NPU. The right shows the mapping of those operations on twelve Compute Tiles, and how the dataflow is mapped through the NPU array.

CHAPTER 5

PROPOSED METHODOLOGY

5.1 Partial-Sum Staging

As discussed in Chapter 2, when intermediate data cannot stay on-chip long enough for a later stage to fully use it, recompute is required to obtain and use that data again. We propose a method for improving the reuse of intermediate data by placing partial-sums inside the Memory Tiles, which have larger capacity to hold the data.

Figure 5.1 shows the methodological change directly. Figure 2.3 in Chapter 2 already established the naive schedule: three heads, equal tiling dimensions, and a no-reuse pipeline that takes 12 time steps to generate one row of output embeddings. The proposed schedule uses the larger scratchpad to store partial accumulations so that output tiles from MHA can be reused rather than regenerated for generation of output tiles 2-3 with the output projection stage. Under the same assumptions, MHAO now completes in 6 time steps rather than 12.

5.2 Hybrid Mapping Strategy

Our work combines runlist and dataflow strategies into a hybrid approach that maps the transformer graph into five coarse-grain kernels, reducing reconfiguration overhead without sacrificing performance. Figure 5.2 contrasts this hybrid design with a naive runlist: the naive approach preserves a long operator chain, whereas the hybrid shortens it by pipelining operations through the NPU.

In the naive runlist (top row), the BERT graph executes as a sequence of fine-grain

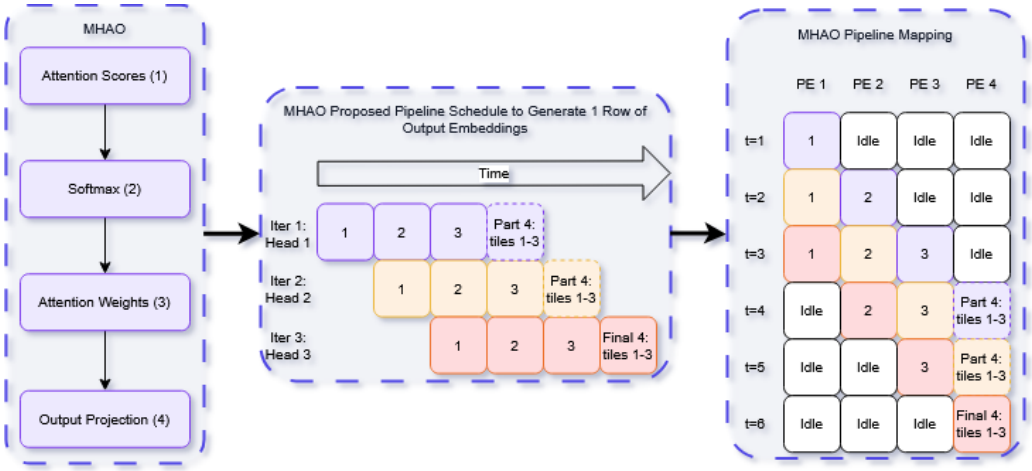

Figure 5.1: Proposed MHAO schedule with partial-sum staging. The left shows the nodes in the subgraph, while the center shows how those operations are scheduled on the NPU and iterates across heads. The right shows that output tiles are generated by time step 6 using the proposed methodology. Larger shared scratchpad capacity keeps partial sums of tiles 1-3 live for the output projection stage.

operators—Q, K, and V projections; attention; output projection; residual add; layer normalization; FFN; another residual add; and layer normalization. Each operator is independently scheduled, with all intermediate data stored in DRAM and the NPU reconfigured between steps. The hybrid version (bottom row) instead groups these into five kernels: QKV-proj, MHAO, AN1, FFN, and AN2, reducing runtime-visible operations while still using a runlist.

These five logical blocks define the layer boundary used throughout this work. Since AN1 and AN2 share the same kernel, the BERT implementation requires only four unique xclbins despite executing five blocks.

GPT-2 reuses the QKV-proj, MHAO, and FFN blocks but differs in surrounding operations: it includes a pre-attention layer normalization, uses causal attention, and ends with a residual add instead of AN2. These changes affect the operator sequence but not the kernel

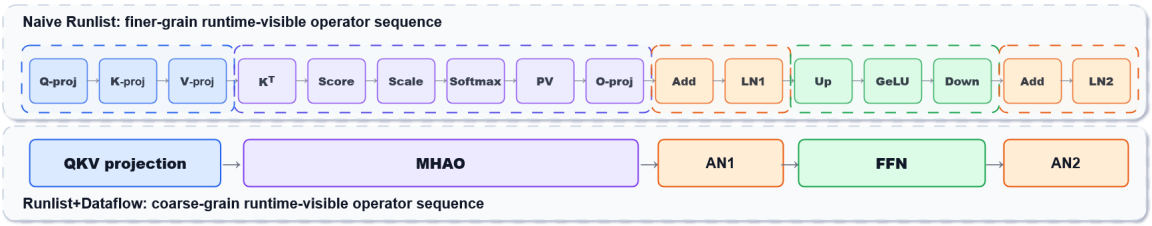

Figure 5.2: Runtime-visible structure for a naive runlist and the hybrid implementation. The latter compresses the graph into a shorter chain of coarse kernels while preserving the execution of the transformer graph.

decomposition.

The hybrid implementation combines pipelined dataflow kernels, enabled by partial-sum staging, with an ordered runlist launch. BERT executes as QKV-proj, MHAO, AN1, FFN, and AN2, while GPT-2 executes as LN, QKV-proj, MHAO, AN, FFN, and Add.

This approach reduces kernel launches and NPU reconfigurations compared to the naive runlist, while partial-sum staging preserves performance when pipelining MHA with output projection and FFN.

5.3 Coarse-Grain Kernels for the Hybrid Implementation

5.3.1 MHAO

The MHAO block combines MHA sub-operations and the output projection operation into one pipeline. Our implementation parallelizes MHAO across two parameters: the sequence dimension and the number of heads. Parallelizing across the sequence dimension is beneficial at longer sequence lengths as the K/V and output projection weight matrices get better reuse as the pipeline is duplicated across the NPU array. At shorter sequence lengths, it may not be possible to fully utilize the NPU array if only parallelizing across the sequence length due to the practical requirements of the microkernels used with the implementation.

Thus, head parallelism was implemented to improve utilization of the array. However, this incurs a sequential cost at the output projection stage, as splitting across the heads means the output projection tiles generate partial outputs, which need to be reduced across all Compute Tiles executing the output projection stage.

This implementation uses our proposed partial-sum staging method to improve the reuse of output tiles from the Attention Weights stage. Figure 5.3 shows simiplified visual mapping of the implementation. The dotted boxes indicate data streamed in through DDR and data streamed out to DDR. The orange tiles show the partial-sum accumulation of output projection in the Memory Tiles. The left panel shows the mapping for head parallelism. Here, MHA for two heads execute in parallel to generate attention scores, apply softmax, produce MHA output tiles, and then contribute to the same output projection tiles with separate partial sums Y_1 and Y_2. The right panel shows the mapping for sequence parallelism. Two batches of tokens, shown as (Y_1, Y_2) and (Y_3, Y_4), are generated via separate pipelines mapped to the NPU array. The key and value tiles and output projection weights are shared across each pipeline, but the staged partial sums remain distinct for each one. Our implementation can be chosen to use only head parallelism, only sequence parallelism, or both.

Algorithm 1 outlines the partial-sum staging method. In it, the partial sums $P_{q,h,n}$ is indexed by the sequence tile of the Query matrix, the head index, and output projection tile index. Each partial sum is accumulated by reusing the output tile of attention, $u_{q,h}$, to multiply with different tiles of the output projection weights. When parallelizing across the heads, the for-loop in lines 8-10 is required to finish the accumulation of the output projection tiles.

The BERT and GPT-2 graphs use the same mapping. However, they do not have the same attention implementation. The BERT layer is noncausal, whereas the GPT-2 layer uses causal self-attention. In our implementation, this only changes the function executed within

Algorithm 1 MHAO with Staged Output projection Partial Sums

for all query tiles q **do**
  for all head h **do**
    $u_{q,h} \leftarrow \text{ATTENTION}(q, h)$
    for all output projection tiles n **do**
      $P_{q,h,n} \leftarrow P_{q,h,n} + u_{q,h} W^{O}_{h,n}$
    end for
  end for
  for all heads h **do**
    $Y_{q,n} \leftarrow \sum_h P_{q,h,n}$
  end for
end for

the Compute Tiles. Additionally, the GPT-2 graph takes in Q/K/V projections from a layer normalized hidden state, while the Q/K/V projections in the BERT graph are not generated from layer normalized hidden state. This difference also doesn't affect data movement through the NPU for this kernel. Thus, the core idea for the implementation is the same for BERT and GPT-2 graphs: output projection partial-sums are staged in Memory Tiles, and the kernel can be parallelized across heads and/or the sequence dimension.

5.3.2 FFN

The FFN block contains an up projection, a GeLU activation, and a down projection. In our implementation, the GeLU activation can be fused with either up projection or down projection. The two projection operations have different loop bounds for reduction: up projection is an $M \times K \times N$ GEMM, while the down projection is an $M \times N \times K$ GEMM. Thus, pipelining one to the other with the same loop bounds, i.e. outer loop as the output tiles to generate and inner loop as the reduction dimension, will result in an imbalanced

pipeline.

The FFN implementation addresses that problem with loop interchange. The down projection is reordered so that its inner loop follows the same K/k loop that the up projection iterates through in its inner loop. This aligns well with our partial-sum staging method: memory tiles can be used to store the down projection partial sums using the up projection's output tile, while up projection generates its next output tile.

Figure 5.4 shows a similar visual mapping as the one used for MHAO. The dotted boxes indicate data streamed in through DDR and data streamed out to DDR. The orange tiles show the partial-sum accumulation of output projection in the Memory Tiles.

The left panel shows the mapping for FFN dimension parallelism. Here, tiles across the FFN dimension execute in parallel to generate up projection tiles, and then contribute to the same down projection tiles with partial sums Y_1 and Y_2. The right panel shows the mapping for sequence parallelism. Two batches of tokens, shown as (Y_1, Y_2) and (Y_3, Y_4), are generated via separate pipelines mapped to the NPU array. The up and down projection weights are shared across each pipeline, but the staged partial sums remain distinct for each one. Our implementation can be chosen to use only FFN dimension parallelism, only sequence parallelism, or both.

Algorithm 2 outlines the partial-sum staging method and shows the effect of loop interchange. Once the up projection and down projection inner loops are the same, staged partial sums let the down projection keep using the same up projection tile before it is replaced. When parallelizing across the FFN dimension, the for-loop in lines 8-10 is required to finish the accumulation of the down projection tiles.

The BERT and GPT-2 implementations use the same FFN mapping. The difference lies in the surrounding residual add and layer normalization stages. In the BERT graph, FFN sits between AN1 and AN2. In the GPT-2 graph, FFN happens after AN, and is followed by only a residual add operation. This difference doesn't affect data movement through the

Algorithm 2 FFN with Staged Down Projection Partial Sums

for all sequence tiles s **do**
  for all up projection tiles n **do**
    $u_{s,n} \leftarrow \text{GELU}(X_s W_n^{up})$
    for all down projection tiles k **do**
      $P_{s,n,k} \leftarrow P_{s,n,k} + u_{s,n} W_{n,k}^{down}$
    end for
  end for
  for all up projection tiles n **do**
    $Y_{s,k} \leftarrow \sum_n P_{s,n,k}$
  end for
end for

NPU for this kernel. Thus, the core idea for the implementation is the same for BERT and GPT-2 graphs: down projection partial-sums are staged in Memory Tiles, and the kernel can be parallelized across the FFN dimension and/or the sequence dimension.

5.3.3 Q/K/V Projection and Residual Add + Layer Normalization

Query, key, and value projections are treated as one larger projection workload rather than three unrelated matrix multiplications. The accumulate-in-place mapping can be used to execute these projections as one operation. This means one xclbin is needed for QKV-proj with the hybrind implementation rather than the three separate xclbins needed with the naive runlist.

For the BERT graph, QKV-proj consumes the hidden-state input directly and produces the concatenated query, key, and value outputs that feed MHAO. For the GPT-2 graph, QKV-proj consumes a layer normalized hidden state.

Figure 5.5 shows simplified visual mappings for the QKV-proj and AN kernels.

The left panel shows a small accumulate-in-place GEMM example. The output tiles $O_{1,1}$, $O_{1,2}$, $O_{2,1}$, and $O_{2,2}$ are stationary while left-matrix operand tiles A_1, A_2 and right-matrix operand tiles B_1, B_2 stream through them. This means that output tiles are accumulated at each Compute Tile, and streamed out once the output has been fully accumulated. The right panel shows the AN kernel pipeline. Inputs X and A are added and normalized, and the resulting tile is then scaled elementwise by the normalization weights. For our AN implementation, an algorithm was developed to determine the largest tile that can fit within the local scratchpad at compile-time to maximize memory utilization at the Compute Tiles.

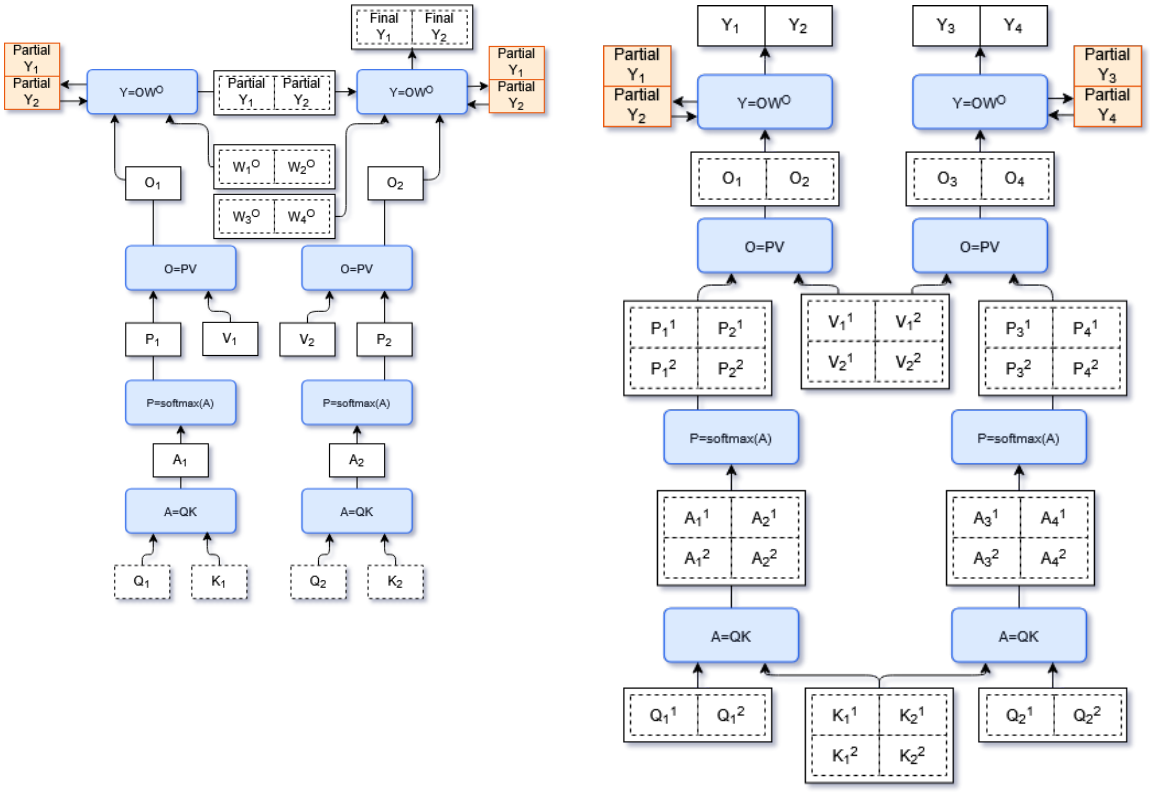

Figure 5.3: MHAO visual mapping: the left panel shows head-parallel execution, while the right panel shows sequence-parallel execution. In both panels, blue denotes execution of stages in Compute Tiles, dotted boxes denote data streams through DDR, orange denotes data in Memory Tiles, and solid boxes denote data streamed through neighboring memory connections.

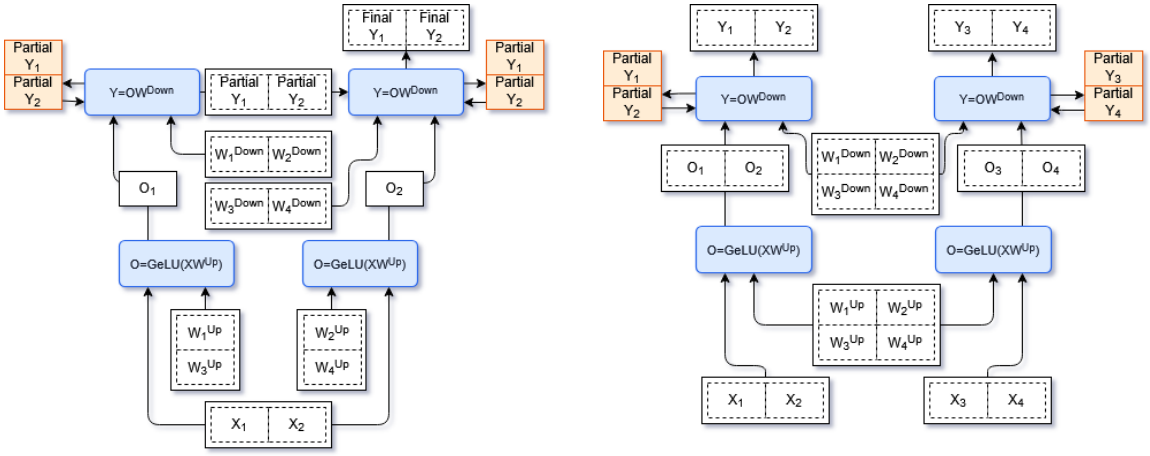

Figure 5.4: FFN visual mapping: the left panel shows parallelization across the FFN dimension, while the right panel shows parallelization across the sequence dimension. In both panels, blue denotes execution of stages in Compute Tiles, dotted boxes denote data streams through DDR, orange denotes data in Memory Tiles, and solid boxes denote data streamed through neighboring memory connections.

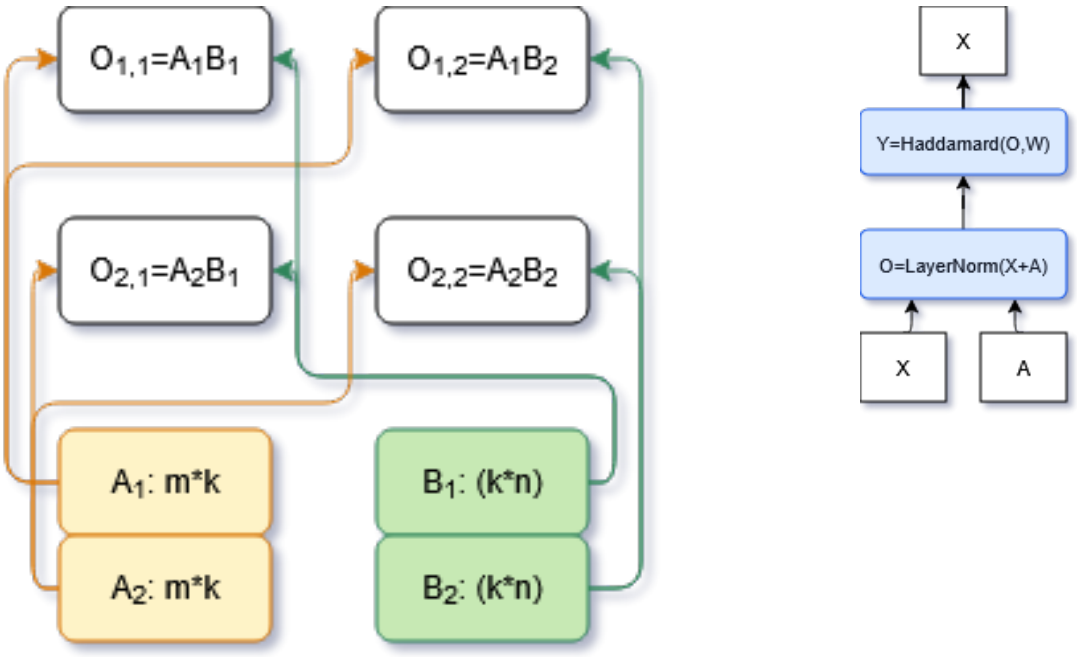

Figure 5.5: Simplified visual mappings for the QKV-proj kernel on the left using accumulate-in-place and the AN kernel on the right.

CHAPTER 6

EXPERIMENTAL SETUP

6.1 Platform and Software Environment

All NPU measurements are collected on an ASUS Vivobook S 15 (M5506) built around AMD's Ryzen AI 9 HX 370 and its integrated Radeon 890M graphics path (ASUSTeK Computer Inc., 2026; AMD, Inc., 2026b). The NPU implementations are built through the IRON, MLIR-AIE, and XRT stack described in Chapter 2 (Melber *et al.*, 2024; AMD Xilinx, 2025a; Hunhoff *et al.*, 2025).

6.2 Evaluated Surface and Baselines

Table 6.1 summarizes the evaluation surface. We evaluate our BERT layer and GPT-2 layer implementations, and compare to two baselines: a naive runlist and an iGPU implementing the same layers. We perform two sensitivity studies by varying the sequence length from 64 to 16384 tokens, and varying embedding dimension by evaluating across different BERT/GPT-2 sizes. We perform two ablation studies by evaluating reconfiguration overhead and evaluating improvement with our partial-sum staging methodology.

The model triplets are written as $d_{\text{emb}}/d_{\text{ffn}}/h$ for embedding dimension, FFN dimension, and number of attention heads. The three encoder configurations correspond to TinyBERT, BERT Base, and BERT Large style settings (Jiao *et al.*, 2019; Devlin *et al.*, 2018). The two decoder configurations reuse the same model shapes in GPT-2-style decoder layers (Radford *et al.*, 2019). Keeping the implementations as fixed layers with inputs and outputs as the

Parameter	Evaluated surface
Families ($d_{\text{emb}}/d_{\text{ffn}}/h$)	TinyBERT (512 / 2048 / 8), BERT-Base (768 / 3072 / 12), BERT-Large (1024 / 4096 / 16), GPT-2 Small (768 / 3072 / 12), and GPT-2 Medium (1024 / 4096 / 16)
Layer variants	BERT-style encoder layers and GPT-2-style decoder layers
Sequence lengths	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384
Batch size and data type	Batch size 1, BF16
Baselines	naive runlist, iGPU

Table 6.1: Evaluation surface

hidden states allows for determining the end-to-end performance without introducing overhead from tokenizer behavior, multi-layer orchestration, or application-specific execution overhead. This is because our work aims to study the effect of execution strategies on the performance of LLM inference.

We compare our hybrid implementation with two baselines: a naive runlist baseline and an iGPU implementation written in PyTorch.

6.3 Metrics

Latency is the primary metric used for evaluation because it captures the combined effect of compute, movement, runtime launch, and staging. The study also reports effective GFLOP/s and effective GFLOP/s/W so that the end-to-end comparisons can be interpreted in both throughput and energy efficiency. Effective throughput is calculated using the following numerator

$$F_{dense} = 8Sd_{emb}^2 + 4Sd_{emb}d_{ff} + 4S^2d_{emb}, \tag{6.1}$$

where S is the sequence length, d_{emb} is the embedding dimension, and d_{ff} is the FFN dimension.

Each factor in Equation 6.1 corresponds to one dense component of the transformer layer. The first term, $8Sd_{emb}^2$, counts the four $S \times d_{emb}$ by $d_{emb} \times d_{emb}$ linear operators: the Q, K, and V projections and the output projection. Each contributes $2Sd_{emb}^2$ FLOPs when one multiply-accumulate is counted as two floating-point operations. The second term, $4Sd_{emb}d_{ff}$, counts the two FFN linear operators, up projection and down projection, with two FLOPs per multiply-accumulate. The third term, $4S^2d_{emb}$, counts the two dense attention products, the attention score product and the attention weights-value product, each contributing $2S^2d_{emb}$ FLOPs.

This numerator excludes softmax, GeLU, residual add, and layer normalization, so latency remains the primary metric for comparison. For the NPU experiments, power is measured from package power reported by `turbostat_pkgwatt`. The iGPU comparison uses a ROCm power path. The resulting efficiency comparison evaluates effective throughput per watt using the same effective throughput calculation.

CHAPTER 7

EVALUATION RESULTS

7.1 Block-Level Characterization

The block study identifies which parts of the runlist dominate. Figure 7.1 shows that the dominant block depends on both family and sequence length.

The strongest trend is the FFN-to-MHAO crossover. At short sequence lengths, FFN is often the slower isolated block because the graph execution is still dominated by linear-in-S dense work. As sequence length grows, MHAO takes over because the attention block scales quadratically with sequence length. TinyBERT crosses over earliest, the 768- and 1024-wide encoder families cross later, and the two GPT-2 decoder families follow the same qualitative progression. The exact crossover point varies by family, but the result is consistent: long-sequence latency is dominated by MHAO.

7.2 End-to-End Hybrid Runlist Versus Naive Runlist

Now we compare the the hybrid runlist+dataflow implementation with the naive runlist implementation using the complete workloads. Figure 7.2 shows the latency comparison, and Figure 7.3 shows the same comparison in terms of effective throughput.

Across the measured surface, the hybrid runlist implementation outperforms the naive runlist baseline in both latency and throughput. The runlist-to-hybrid latency and throughput ratios range from 1.27x to 2.63x.

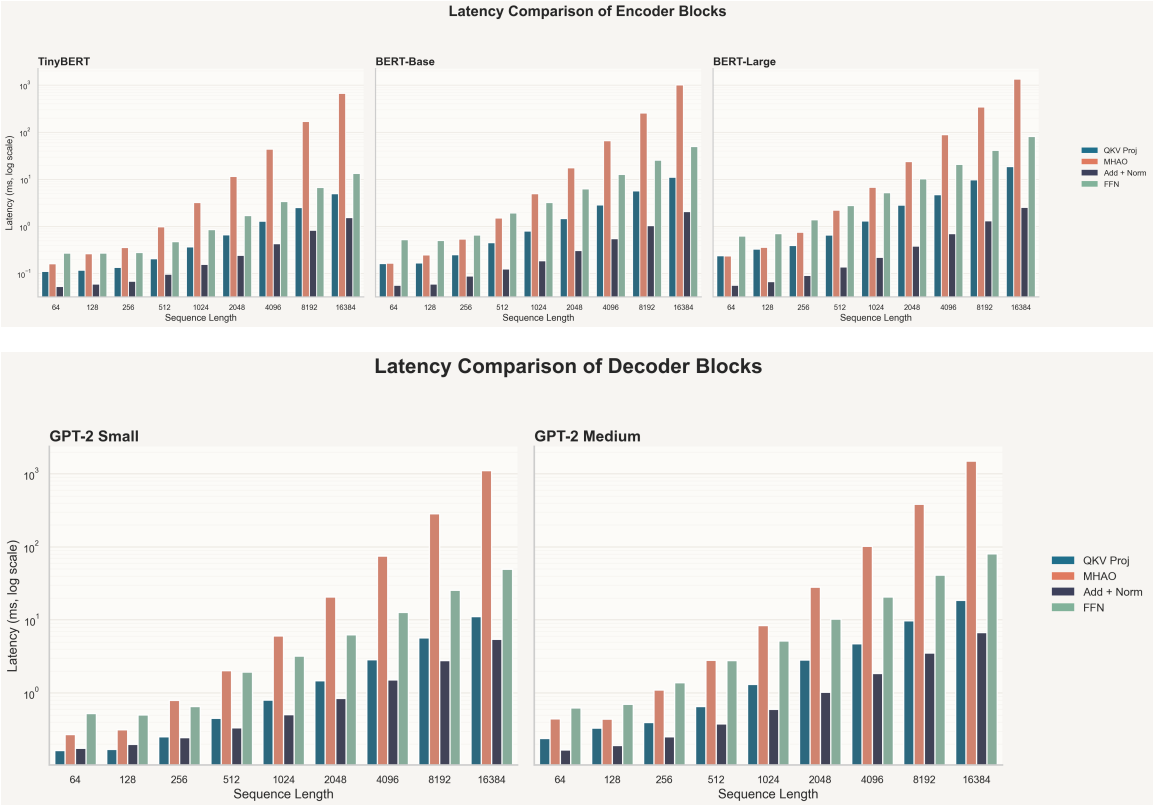

Figure 7.1: Best block-level latencies across sequence length for the encoder (top) and decoder (bottom) families. The y-axis is logarithmic.

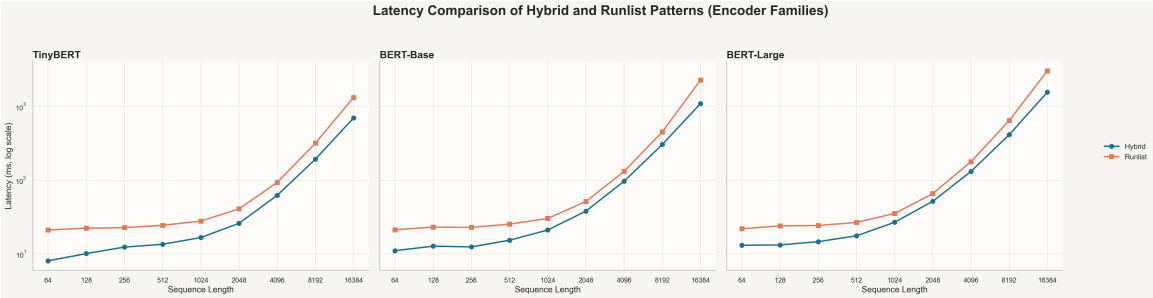

Figure 7.2: End-to-end NPU latency for the hybrid runlist implementation versus the naive runlist baseline in the encoder (top) and decoder (bottom) families. The y-axis is logarithmic.

7.3 Reconfiguration Overhead

By utilizing the runlist method, we incur a reconfiguration overhead per kernel used in the list. This overhead is large at short sequence lengths. For the encoder families, the hybrid implementation uses five unique `xclbin` artifacts, whereas the naive runlist baseline uses 12 unique `xclbin` artifacts. For the decoder families, the hybrid implementation uses six unique `xclbin` artifacts, whereas the naive runlist baseline uses 13 unique `xclbin` artifacts.

Figure 7.4 quantifies the reconfiguration overhead by comparing the aggregate latency of the measured blocks against the full hybrid end-to-end latency. At short sequence lengths, the end-to-end latency remains much larger than the aggregate block latencies, which indicates that NPU reconfiguration and launch overhead makes runlist impractical for shorter sequence lengths. As sequence length grows, that ratio collapses toward 1x as those costs become amortized over the full execution.

7.4 Partial-Sum Staging Improvement

We aim to quantify the performance improvement obtained by utilizing our partial-sum staging methodology. Figures 7.5 and 7.6 show the MHAO sweep for the encoder and decoder families, and Figures 7.7 and 7.8 show the same trend for FFN. Staging depth represents the amount of reuse obtained by staging more tiles within the Memory Tiles. In all four plots, using the highest staging depth of 8 improved MHAO by up to 7.72x to 7.80x, and FFN by up to 4.67x to 6.99x.

7.5 GPU Comparison

We compare with an iGPU implementation of the BERT and GPT-2 layers to place the NPU results in the context of another accelerator on the same Ryzen platform. Figure 7.9 compares raw effective throughput across embedding sizes and sequence lengths.

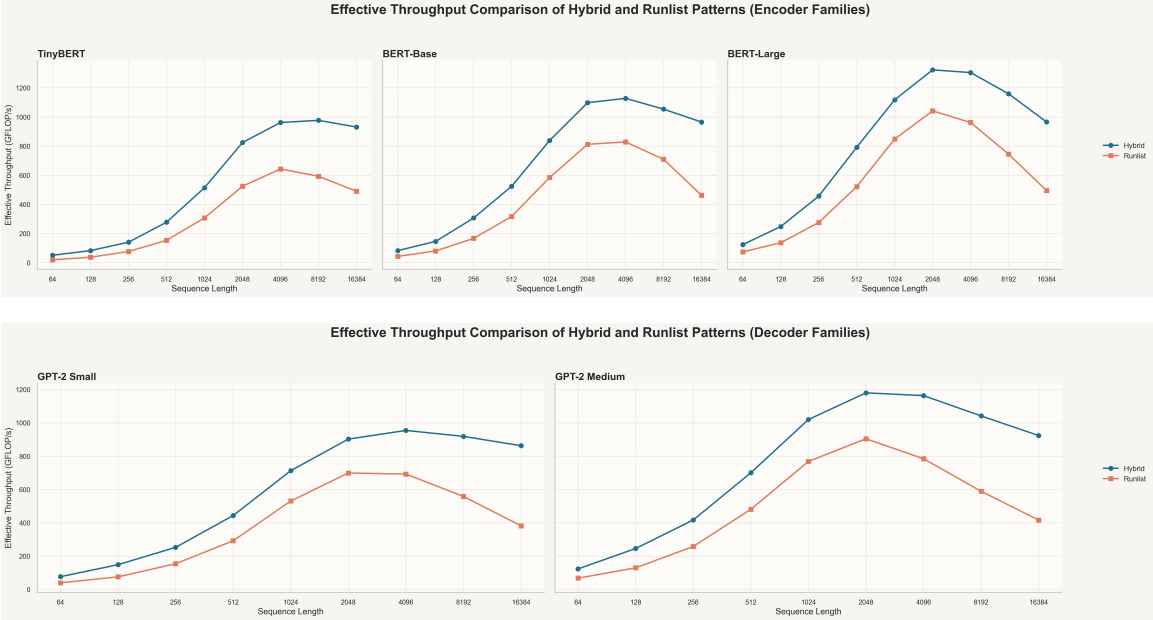

Figure 7.3: End-to-end NPU throughput for the hybrid runlist implementation versus the naive runlist baseline in the encoder (top) and decoder (bottom) families

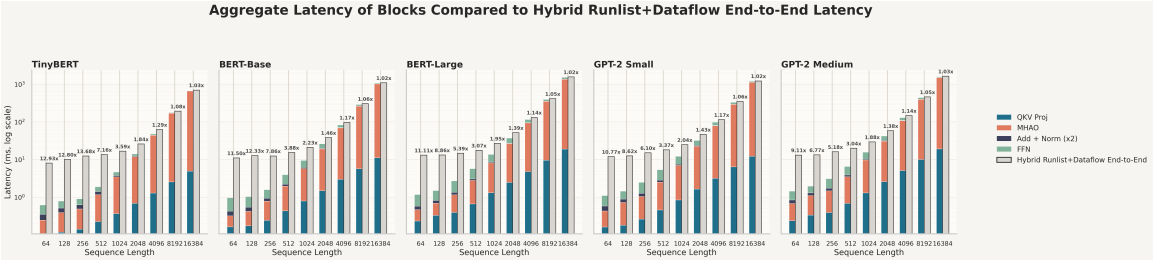

Figure 7.4: Ablation study for the reconfiguration overhead. The stacked bars on the left show the aggregate latency of the kernels used in the hybrid implementation. The bars on the right show the end-to-end latency of the hybrid implementation itself. The difference between the two bars outline the reconfiguration overhead incurred when using the runlist strategy.

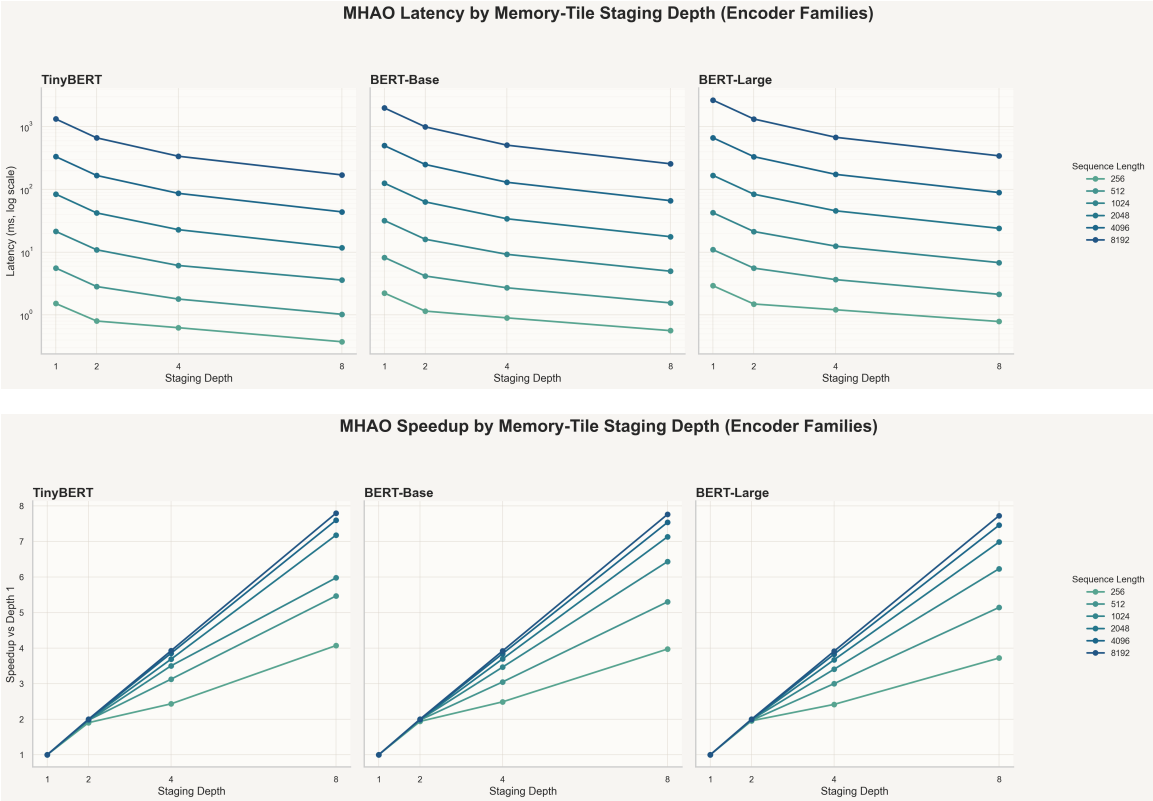

Figure 7.5: Encoder-family MHAO partial-sum-staging results: latency (top, logarithmic y-axis) and speedup (bottom)

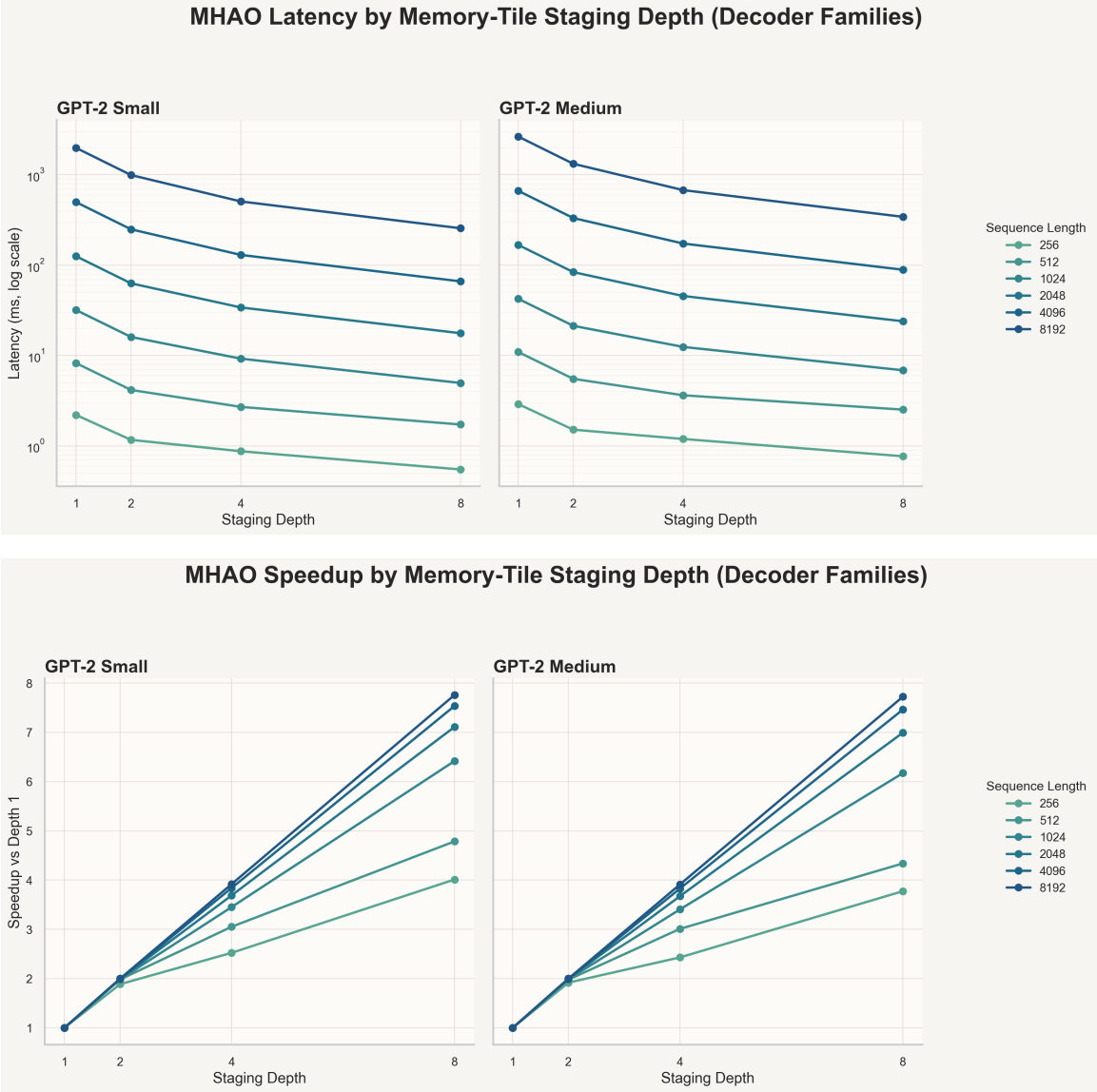

Figure 7.6: Decoder-family MHAO partial-sum-staging results: latency (top, logarithmic y-axis) and speedup (bottom)

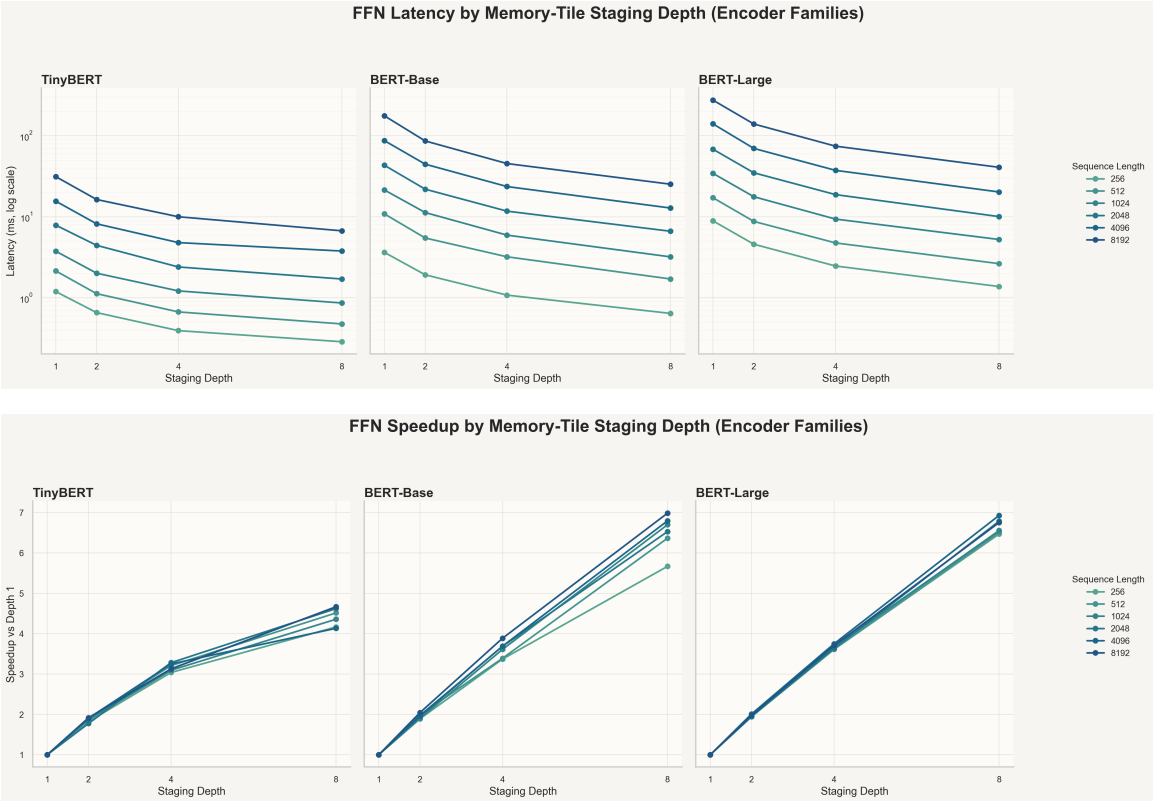

Figure 7.7: Encoder-family FFN partial-sum-staging results: latency (top, logarithmic y-axis) and speedup (bottom)

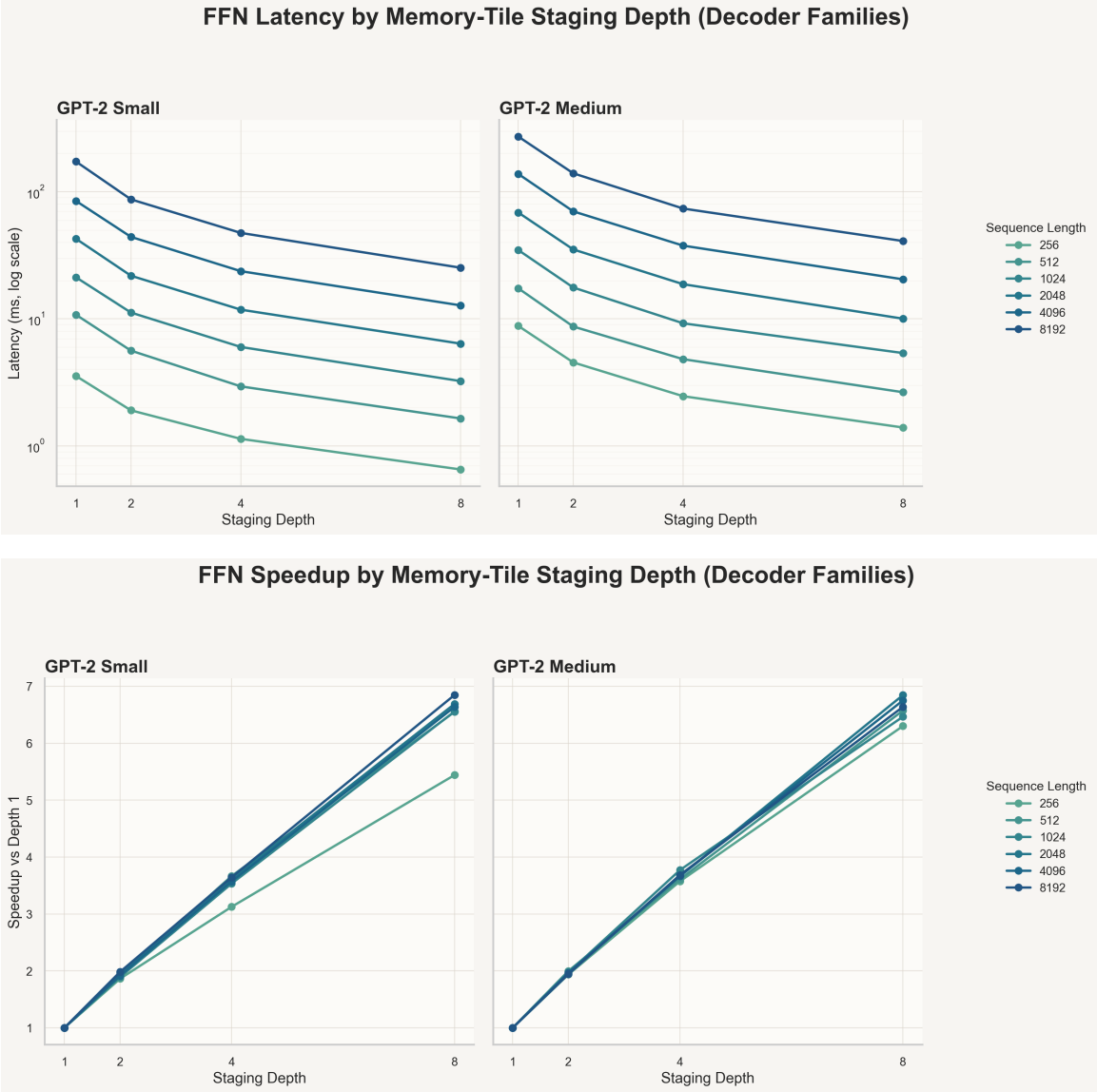

Figure 7.8: Decoder-family FFN partial-sum-staging results: latency (top, logarithmic y-axis) and speedup (bottom)

Figure 7.9: Raw throughput comparison for the iGPU and hybrid NPU in the encoder (top) and decoder (bottom) families

The iGPU provides higher throughput throughout the measured surface compared to the NPU. This is because the iGPU has significantly higher peak throughput compared to the NPU (as mentioned in Chapter 2). The hybrid NPU reaches up to 74.6% of raw effective throughput of the iGPU, with the strongest relative results appearing at longer sequence lengths.

7.6 Energy Efficiency

Figure 7.10 compares the energy efficiency of our hybrid NPU implementation to the iGPU.

The efficiency comparison is more favorable to the NPU. The hybrid path becomes more energy efficient than the iGPU from 1024 tokens onward in BERT-Base, BERT-Large, GPT-2 Small, and GPT-2 Medium, and from 2048 tokens onward in TinyBERT. Overall, the throughput-per-watt comparison shifts in favor of hybrid at longer sequence lengths.

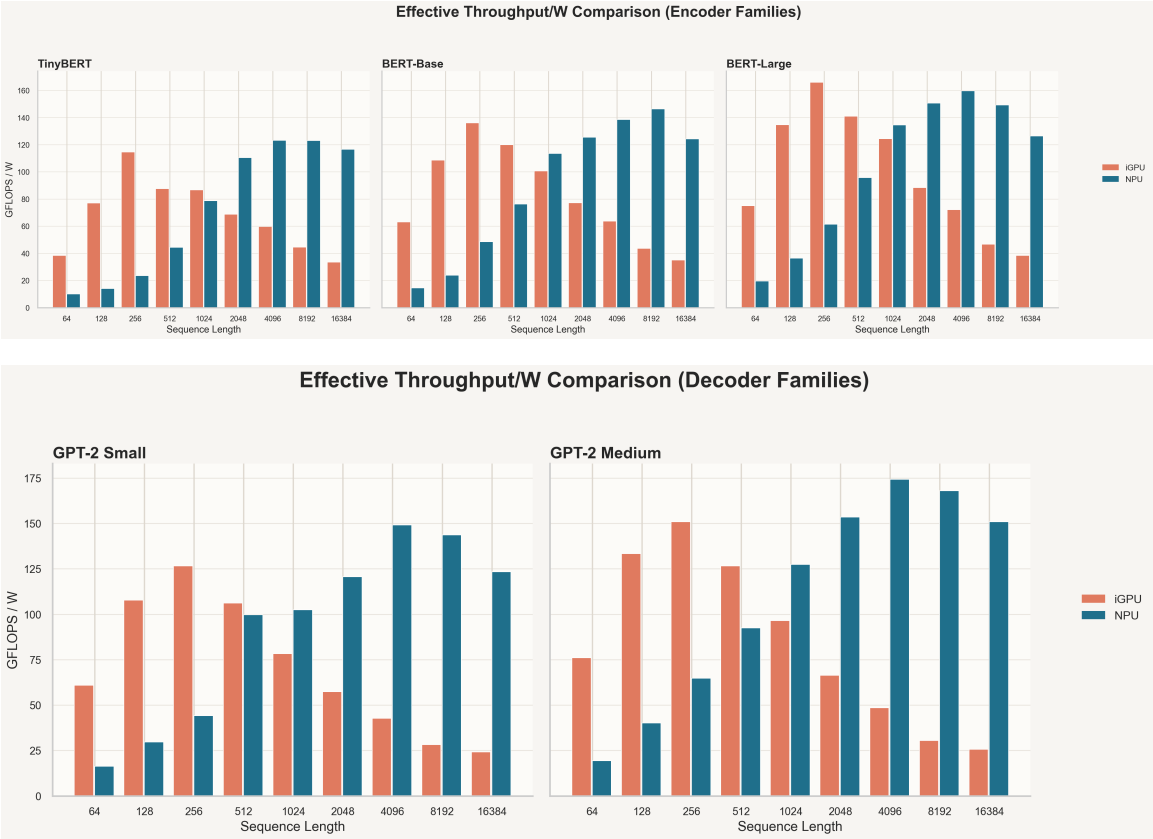

Figure 7.10: Efficiency comparison for the iGPU and hybrid NPU in the encoder (top) and decoder (bottom) families

CHAPTER 8

DISCUSSION

8.1 Deployment Implications

The measured results support a clear deployment interpretation. Within the runlist strategy, the hybrid implementation is stronger than the naive runlist baseline as a result of a reduced number of operator dispatches and the staging of partial sums in Memory Tiles for the dataflow-mapped kernels.

The results also indicate that execution strategy should remain sequence-sensitive. Shorter sequence lengths are dominated by reconfiguration overhead. At longer sequence lengths, the reuse of tiles inside pipelined dataflows such as MHAO and FFN are important for achieving efficient performance. A plausible deployment split for LLM inference is therefore fuller end-to-end pipelining at shorter sequence lengths where activation tiles are smaller, and more aggressively optimized runlist implementations at longer sequence lengths where the benefits of coarse packaging and staged partial sums are larger.

Decoder models also present a related opportunity. Partial-sum staging may be especially effective in the decode stage, because activations are smaller than in compared to the prefill phase. The main remaining obstacle is layer normalization: end-to-end decode stage pipelines still require the full rows to bee visible, which makes end-to-end pipelining difficult. Decode stage pipelines are therefore a natural next target, but not a trivial extension of the current implementation.

The offload viewpoint also remains relevant for deployment. A natural offload direction

is dynamic workload partitioning around GEMM-heavy operations. The runtime sequence can be implemented such that the GEMM workload can be partitioned evenly across cores, and any leftovers that don't partition evenly leave as little cores idle as possible. With dynamic partitioning, the NPU configuration used for offloading GEMM can use larger tiles instead of being constrained by the minimum GEMM workload dimensions to process end-to-end for LLM inference.

8.2 Software and Toolchain Implications

The current methodology exposes both a software-stack opportunity and a mapping opportunity. In MLIR-AIE, high-level movement constructs are ultimately lowered into buffers, locks, DMA configuration, and stream connectivity (AMD Xilinx, 2025a; AMD, Inc., 2025b). Deeper staged pipelines therefore consume not only storage in memory tiles, but also descriptor and DMA programming state. In practice, staging depth is bounded by both memory capacity and the finite buffer-descriptor budget available to support the required transfers.

That constraint implies a sharper software trade-off than "stage more" alone. One practical direction is to maximize staging depth where descriptor pressure remains manageable, but accept selective recompute when descriptor pressure becomes the limiting factor. The objective is not to preserve every intermediate tile at any cost, but to preserve the partial state that is most valuable while keeping the DMA program within feasible limits.

The same limitation suggests a toolchain opportunity. The measured implementation uses objectFIFOs and related movement constructs to express staged partial accumulations in memory tiles. A more direct abstraction for "store this larger partial state in memory tiles and stream it back in tiles to the same or a neighboring core" could make this pattern easier to express and automate. Such an operator would better match the actual data-movement structure of partial-sum staging than the current indirect composition of FIFOs and DMA

configuration.

Automation of the hybrid runlist+dataflow implementation is another possible direction. A compiler/runtime flow could allow the programmer to mark the start and end of a pipelineable region, automatically stitch supported kernels into a runlist, and retain the option to fuse pipelineable regions without hand-writing a bespoke fused kernel for every new pattern. Under such a flow, the runlist mechanism begins to resemble an inference engine whose supported operations can be stitched dynamically, while pipelined dataflow mappings provide better throughput when the toolchain can identify them.

8.3 Hardware Implications

One obvious hardware direction is more support for partial-sum staging. Larger memory-tile capacity, richer DMA support, or a less restrictive descriptor budget would all make deeper staging easier to sustain. Reduced precision is another likely lever. The present study uses BF16, but INT8 or FP8 would allow larger tiles and potentially deeper staging by reducing storage pressure for the same mapped region. That would not remove the descriptor constraint by itself, but it would relax one side of the trade-off and could make fuller on-chip pipelines practical for larger workloads.

The tiling itself could also be experimented with. Because activations are streamed through neighboring connections while weights are streamed through DDR, a taller sequence tile together with a wider embedding tile may better balance those two streams.

CHAPTER 9

CONCLUSION

LLM inference on edge devices is an attractive option for reducing costs of serving LLM inference through datacenters. NPUs are designed to process AI workloads efficiently. However, there are many ways to execute LLM inference on NPUs. Our work discussed and explored three broad categories for executing LLM inference: offload, runlist, and dataflow. We proposed a hybrid implementation combining the runlist and dataflow strategies to improve upon a naive runlist implementation.

Moreover, a partial-sum staging method was proposed to improve the performance of pipelined dataflow implementations. This method improved reuse of intermediate data by leveraging the larger capacity of shared scratchpad. Using this method improved the latency of our MHAO block by up to 7.80x in MHAO and FFN block by up to 6.99x.

Across all sequence lengths and embedding sizes, our hybrid implementation beat naive runlist in both latency. Against the integrated GPU, the hybrid NPU implementation was more energy-efficient at moderate and long sequence lengths across all embedding sizes.

Our work shows that practical edge inference on the Ryzen NPU depends on the choice of execution strategy, and how dataflow pipelines are mapped such that reuse is maximized under tiled execution. A next steps for this work includes the evaluation of the offload strategy built around dynamic workload partitioning for GEMM operations.

BIBLIOGRAPHY

AMD, Inc., *Versal AI Engine ML v2 Architecture*, Advanced Micro Devices, Inc., URL https://docs.amd.com/r/en-US/am027-versal-aie-ml-v2, document No. AM027 (2024).

AMD, Inc., "Ai engine development environment for versal ai edge series gen 2 device", https://docs.amd.com/r/2025.1-English/ug1304-versal-acap-ssdg/AI-Engine-Development-Environment-for-Versal-AI-Edge-Series-Gen-2-Device (2025a).

AMD, Inc., "Ai engine-ml kernel and graph programming guide", https://docs.amd.com/r/en-US/ug1603-ai-engine-ml-kernel-graph (2025b).

AMD, Inc., "Accelerator and gpu hardware specifications", https://rocm.docs.amd.com/en/latest/reference/gpu-arch-specs.html (2026a).

AMD, Inc., "Amd ryzen ai 9 hx 370", https://www.amd.com/en/products/processors/laptop/ryzen/ai-300-series/amd-ryzen-ai-9-hx-370.html (2026b).

AMD, Inc., "Npu management interface", https://ryzenai.docs.amd.com/en/latest/xrt_smi.html (2026c).

AMD Xilinx, "mlir-aie: An mlir-based toolchain for amd ai engine-enabled devices", https://github.com/Xilinx/mlir-aie (2025a).

AMD Xilinx, "mlir-aie: An mlir-based toolchain for amd ai engine-enabled devices", https://github.com/Xilinx/mlir-aie/tree/main/mlir_exercises/tutorial-2/tutorial-2a (2025b).

AMD Xilinx, "vector_scalar_add_runlist programming example", https://github.com/Xilinx/mlir-aie/tree/main/programming_examples (2026a).

AMD Xilinx, "Xrt runlist api documentation", https://github.com/Xilinx/XRT (2026b).

ASUSTeK Computer Inc., "Asus vivobook s 15 (m5506)", https://www.asus.com/laptops/for-home/vivobook/asus-vivobook-s-15-oled-m5506/ (2026).

Deshmukh, A., V. Y. Raparti and S. Hsu, "Zen-attention: A compiler framework for dynamic attention folding on amd npus", URL https://arxiv.org/abs/2508.17593 (2025).

Devlin, J., M.-W. Chang, K. Lee and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding", URL https://arxiv.org/abs/1810.04805 (2018).

Du, S., M. Yu, Z. Ni, J. Cai, Q. Yang, T. Wei and Z. Xu, "Mapping gemma3 onto an edge dataflow architecture", URL https://arxiv.org/abs/2602.06063 (2026).

Harada, T. and A. Yoshimura, "How to accelerate ai applications on rdna 3 using wmma", https://gpuopen.com/learn/wmma_on_rdna3/ (2024).

Hunhoff, E. *et al.*, "Efficiency, expressivity, and extensibility in a close-to-metal npu programming interface", in "Proceedings of the International Symposium on Field-Programmable Custom Computing Machines", (2025).

International Energy Agency, "Energy and ai", Tech. rep., IEA, URL https://www.iea.org/reports/energy-and-ai, published April 10, 2025 (2025).

Jiao, X., Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang and Q. Liu, "Tinybert: Distilling bert for natural language understanding", URL https://arxiv.org/abs/1909.10351 (2019).

Melber, J., K. Denolf, G. Singh, P. James-Roxby, M. Awad, S. Bayliss, J. Fifield, E. Hunhoff, A. Khodamoradi, J. Lo, S. Neuendorffer, A. Bisca, Y. Papadopoulos, E. Richter, A. Rösti, E. Wang, P. Vasireddy and R. Wittig, "Leveraging the iron ai engine api to program the ryzen ai npu", https://github.com/Xilinx/mlir-aie/blob/main/docs/Presentations.md (2024).

Noffsinger, J., M. Patel and P. Sachdeva, "The cost of compute: A $7 trillion race to scale data centers", https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-cost-of-compute-a-7-trillion-dollar-race-to-scale-data-centers, mcKinsey Quarterly, published April 28, 2025 (2025).

Radford, A., J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language models are unsupervised multitask learners", https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf (2019).

Rösti, A. and M. Franz, "Unlocking the amd neural processing unit for ml training on the client using bare-metal-programming tools", URL https://arxiv.org/abs/2504.03083 (2025).

Taka, E., A. Rösti, J. Melber, P. Vasireddy, K. Denolf and D. Marculescu, "Striking the balance: Gemm performance optimization across generations of ryzen ai npus", URL https://arxiv.org/abs/2512.13282 (2025).

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention is all you need", in "Advances in Neural Information Processing Systems", (2017).

Wang, C., W. Pang, X. Wu, G. Jun, L. Romero, E. Taka, D. Marculescu *et al.*, "Can asymmetric tile buffering be beneficial?", URL https://arxiv.org/abs/2511.

16041 (2025a).

Wang, E., S. Bayliss, A. Bisca, Z. Blair, S. Chowdhary, K. Denolf, J. Fifield, B. Freiberger, E. Hunhoff, P. James-Roxby, J. Lo, J. Melber, S. Neuendorffer, E. Richter, A. Rösti, J. Setoain, G. Singh, E. Taka, P. Vasireddy, Z. Yu, N. Zhang and J. Zhuang, "From loop nests to silicon: Mapping ai workloads onto amd npus with mlir-air", URL https://arxiv.org/abs/2510.14871 (2025b).

Xu, Z., 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", pp. 172–178 (2025).