Execution Strategies for LLM Inference on the Ryzen NPU

Curt John Bansil

Arizona State University

April 2026

Motivation

- Large language model (LLM) inference typically served in datacenters
  - Cost is immense, projected to rise as AI adoption increases
- One alternative is to run inference on an edge device
- Neural Processing Units (NPUs) are designed to be energy efficient in handling AI workloads, making it a strong candidate for local inference
- Chose Ryzen NPU for its architecture and open software stack

Problem Statement 1

- Many methods for executing LLM workloads on the NPU
  - Offload: Highly optimized execution of one LLM operation on the device
  - Runlist: Ordered dispatch of LLM operations to the device
  - Dataflow: Pipeline LLM operations through processing elements in the device

Problem Statement 2

- NPU has limited on-chip memory, LLM workloads don’t fit
  - Tiling becomes necessary
- With Dataflow method, pipelining operations in a tiled manner becomes difficult
  - Intermediate results required at a later point don’t stay on-chip long enough

Thesis Contributions

- Reusable methodology that improves reuse of intermediate results when pipelining dataflows
- Hybrid implementation combining dataflow and runlist methods for executing transformer encoder workloads
- Demonstration of implementation with BERT, and quantitative comparison to multiple baselines: naïve runlist (NPU) and iGPU

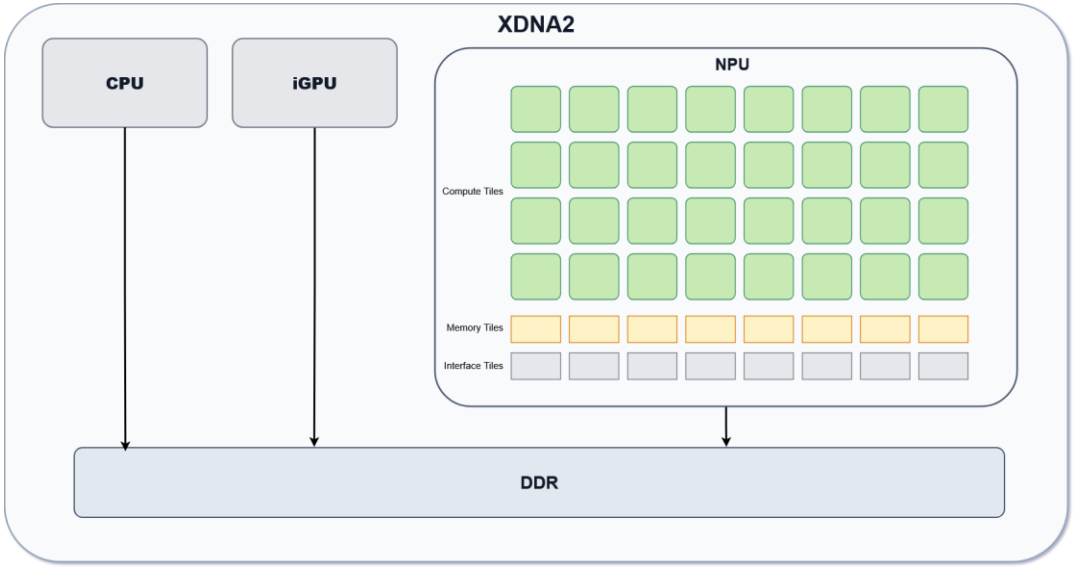
Agenda

- Background
- Related Work
- Execution Strategies
- Proposed Methodology
- Evaluation
- Takeaways

Background

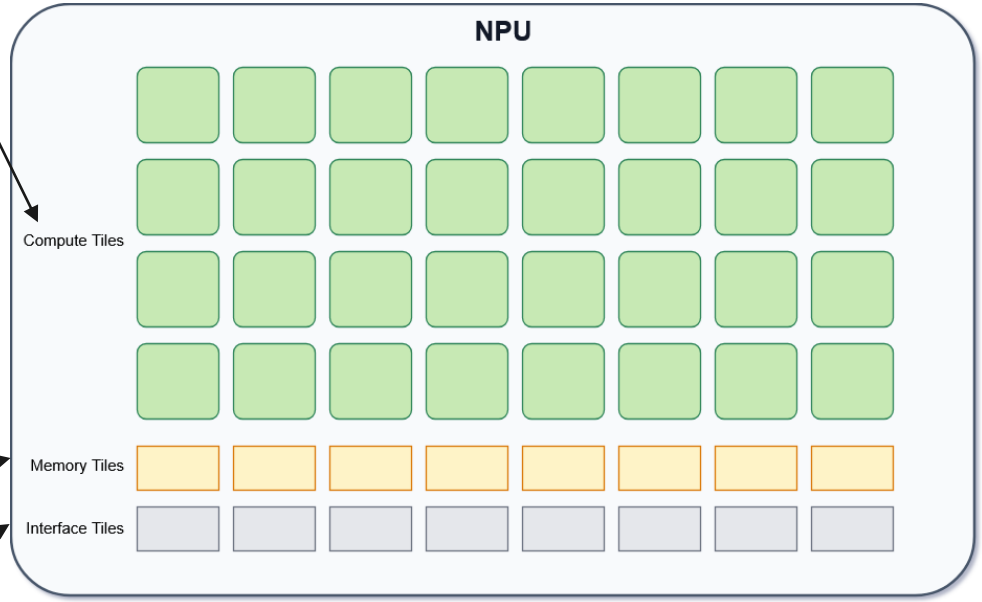
Ryzen AI

- Ryzen AI processor (XDNA 2) integrates CPU, GPU, and NPU in one chip
  - NPU only sees global memory, no cache-coherent view of data in the caches
  - LPDDR5X DDR: 7500 MT/s, 128-bit bus
    - 120 GB/s peak DRAM bandwidth



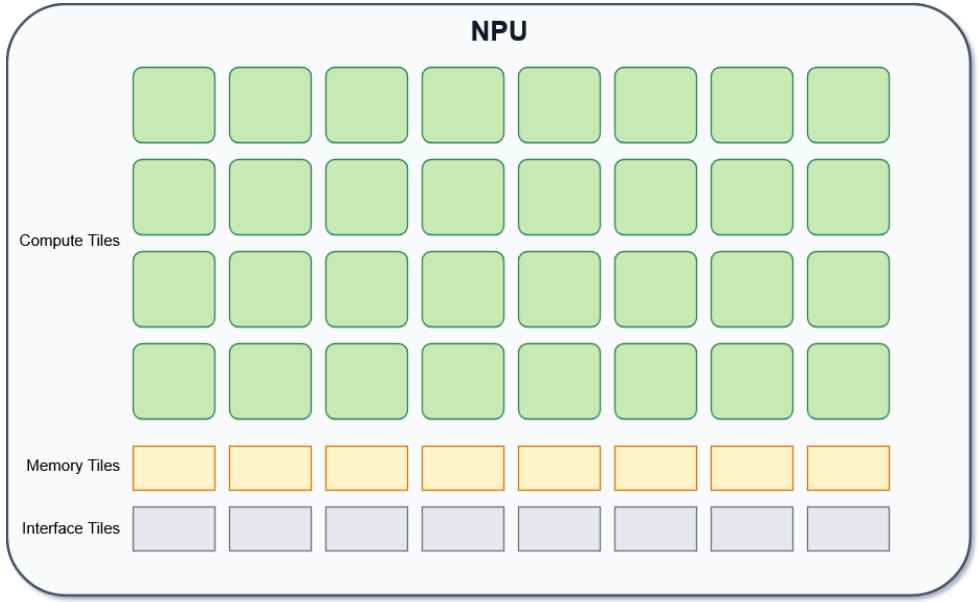
Ryzen NPU

- Spatially arranged compute tiles (CT)
  - 32 CTs for XDNA2
  - 7.4 TFLOP/s peak for bfloat16
    - bfloat16: 64 MACs/cycle/core
    - Clock frequency: 1.8 GHz
  - Local scratchpad memory (64 KB)
  - Neighboring memory access of tiles to its North, West, and South
- Row of memory tiles (MT)
  - Larger shared scratchpad (512 KB)
- Row of interface tiles (IT)
  - Writes to and reads from DDR



Ryzen NPU

- Data movement accelerators (DMA) at each tile to overlap data movement and compute
  - CT/IT: 2 DMA channels each to stream in and out
  - MT: 6 channels each to stream in and out
- Command processor can access all cores and stream switches to reconfigure at run time



Configuring the NPU

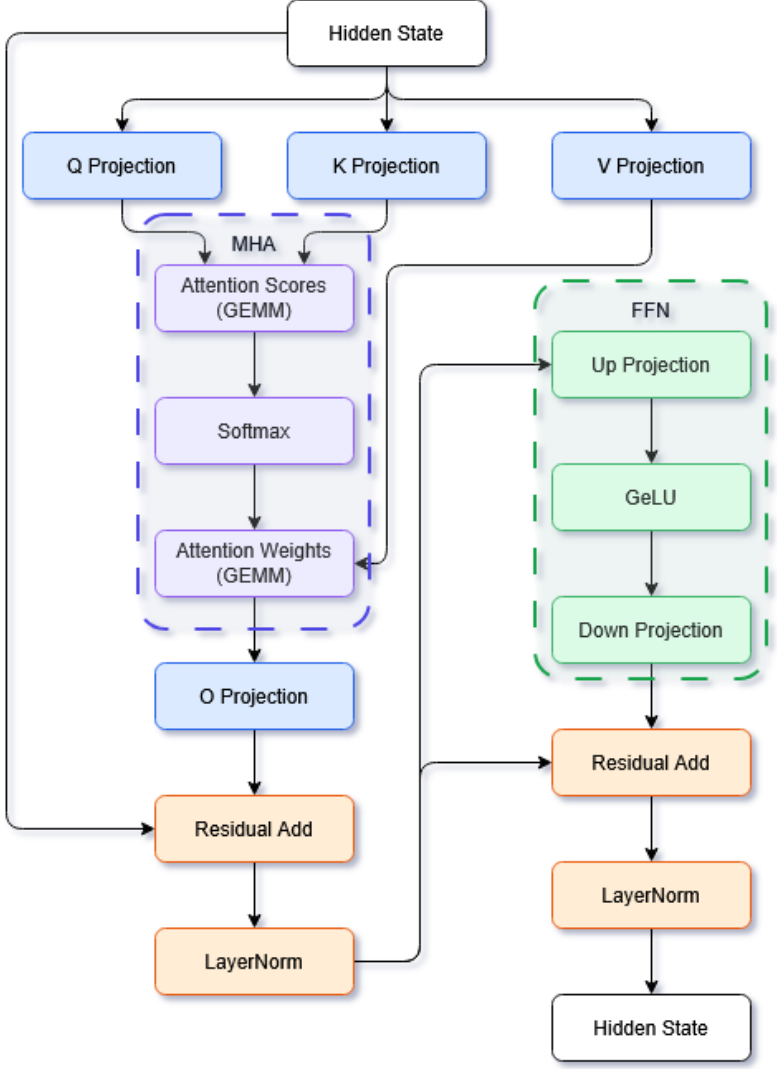
- Load computation kernels onto compute tiles
- Explicitly configure stream switches and DMA accesses for data movement
- Interface between host and NPU using Xilinx Run Time (XRT)
  - To initialize NPU configuration
  - Initialize host buffers in DRAM for passing input and output data between CPU and NPU
  - Write instructions to NPU's command processor

NPU Programming

- IRON toolkit
  - MLIR-based Python compiler
  - APIs for describing data movement and compute kernels on processing elements
- C++ used to write kernels that run on processing elements
- Python used to describe data movement on NPU
- IRON Python script generates MLIR, used to compile binaries loaded to the NPU

Workload Characterization

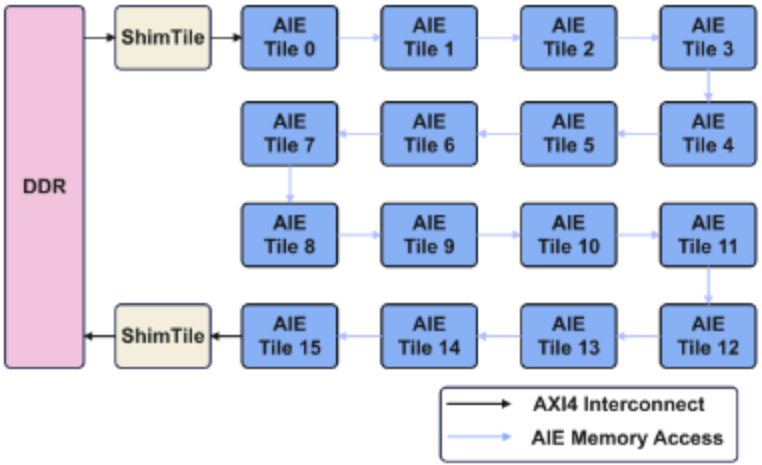
- Our Implementation processes a BERT workload
  - $S \coloneqq sequence\ length$
  - $d_{emb} \coloneqq embedding\ dim = num\ heads * head\ dim$
  - $d_{ffn} \coloneqq 4 * d_{emb}$
- Total MACs for layers:
  - Projections $= 4Sd_{emb}^2$
  - Multi-Head Attention (MHA) $= 2S^2d_{emb}$
  - Feed-Forward Network (FFN) $= 2Sd_{emb}d_{ffn}$



Related Work

Tile-Level Pipeline for Linear Scalable Stencil Computation on AMD AI Engines

- *Xu et al.* pipelines stencil computation on XDNA
- Able to achieve 14 GFLOPS by leveraging ping-pong buffering to overlap compute and data movement, and by keeping necessary data for compute within local scratchpad memory
- In our case, data necessary for compute can’t fit within local scratchpad



https://dl.acm.org/doi/10.1145/3706628.3708822

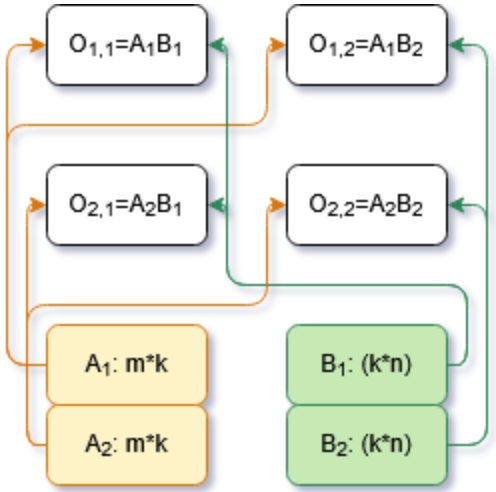
Mapping GEMMA3 onto an edge dataflow architecture

- *Du et al.* is a recent paper from Feb 2026
- Maps subgraphs of Gemma3's dataflow onto the NPU as separate kernels
- Runs inference end-to-end by sequentially executing those kernels
- Uses similar principles as our end-to-end implementation, but our work focuses on encoder models instead of decoder models
    - Decoders are mainly integrated into systems with user-facing interaction
    - Encoders are used for production AI systems that need semantic search, RAG retrieval, reranking, and/or classification

https://arxiv.org/abs/2602.06063

Unlocking the AMD Neural Processing Unit for ML Training on the Client Using Bare-Metal-Programming Tools

- *Rosti et al.* maps matrix multiplication (GEMM) to the NPU
- Demonstrates efficient way to offload GEMM operations end-to-end by adjusting GEMM kernel loop bounds dynamically at run time
- Our implementation builds upon their mapping to fuse Q/K/V projections



https://ieeexplore.ieee.org/abstract/document/11008965

Striking the Balance: GEMM Performance Optimization Across Generations of Ryzen™ AI NPUs

- Work by *Taka et al.* shows that efficient processing of GEMM workloads requires tiling and mapping decisions based on the workload's characteristics
- Demonstrates that memory placement must be configured precisely to achive good performance

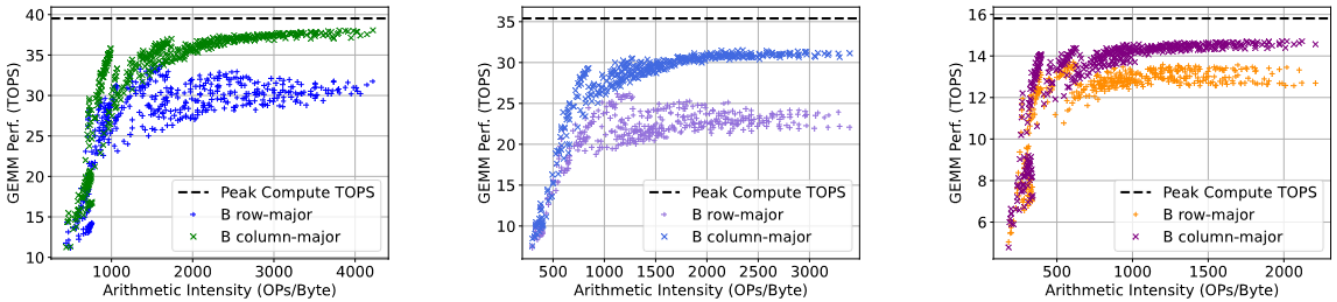


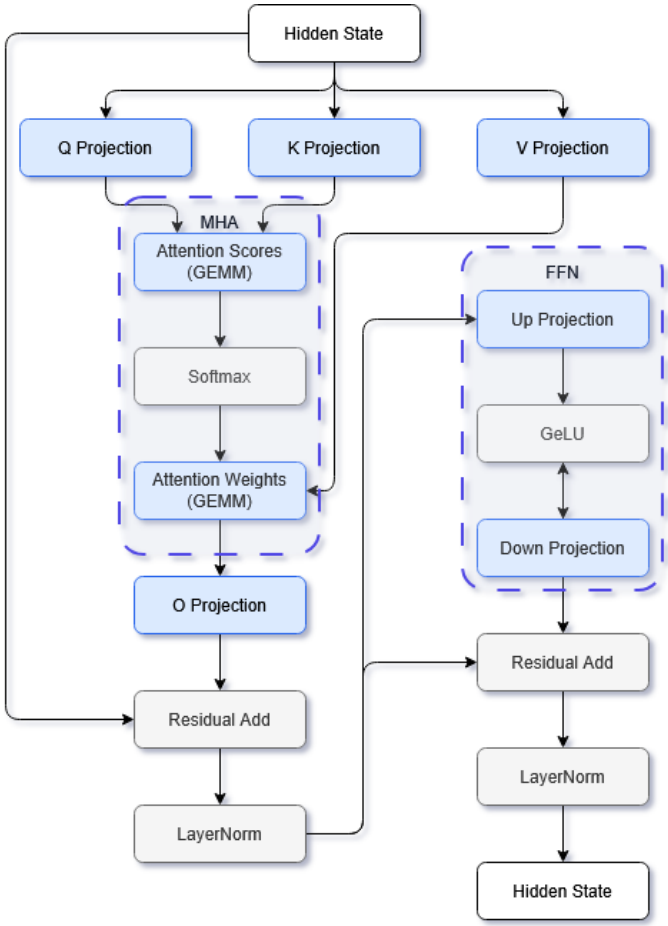
Figure 8: Roofline GEMM performance sweeps for various matrix sizes on XDNA2.

https://arxiv.org/abs/2512.13282

Execution Strategies

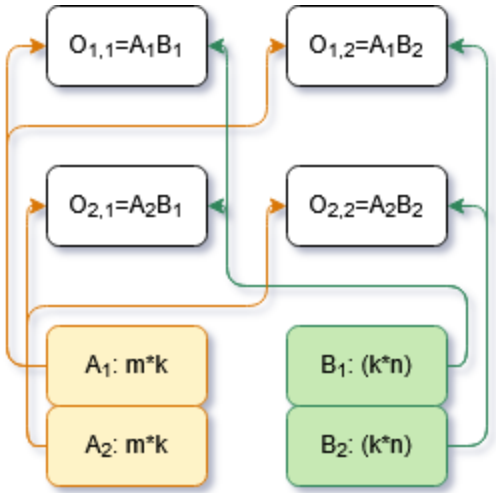
Offload Strategy

- Most of compute in LLMs are GEMMs
- One reasonable way to improve end-to-end performance with LLMs is to offload only GEMM operations to NPU



GEMM Offload Example

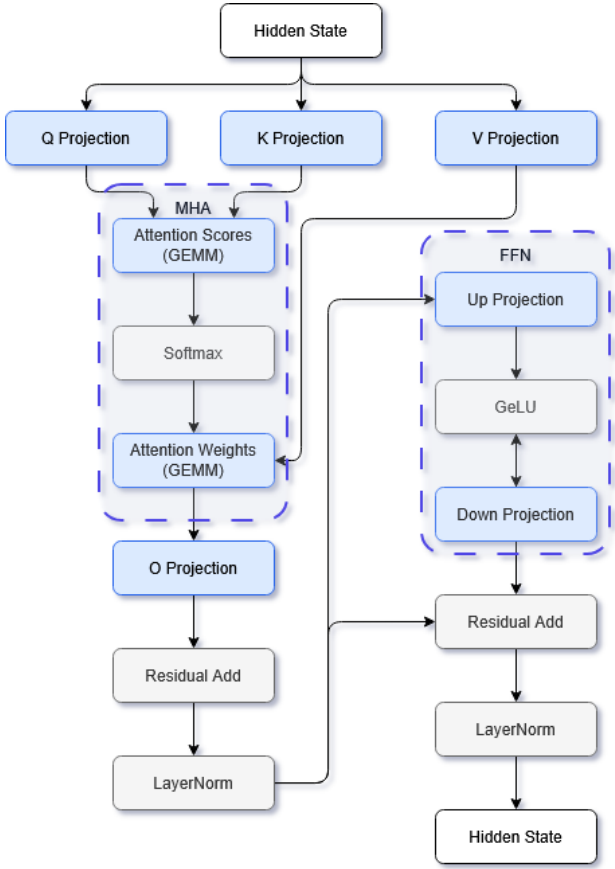
- Mapping: accumulate-in-place strategy
  - Allows for reuse of tiles across rows and columns
  - $Total\ Data\ Transfer = \frac{MKN}{cn} + \frac{MKN}{rm} + MN$
    - $M, K, N \coloneqq GEMM\ workload\ size$
    - $m, k, n \coloneqq Tiling\ parameters$
    - $r, c \coloneqq number\ of\ rows\ and\ columns\ of\ AIEs\ used$



https://ieeexplore.ieee.org/abstract/document/11008965

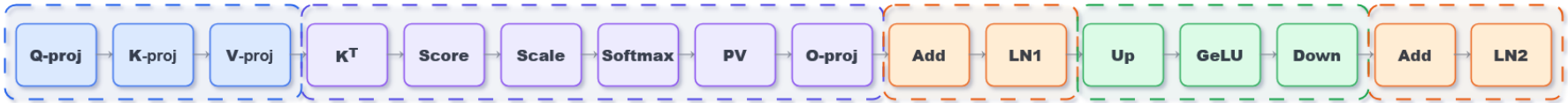
Offload Strategy

- Benefits:
  - All 32 tiles available to execute one operation, e.g. GEMMs
  - Can focus on highly optimizing that operation on NPU
- Drawbacks:
  - Executing non-linear operations on host (CPU) increases energy consumption
  - Switching between host and NPU compute incurs synchronization and launch overheads
  - Quadratic scaling of Multi-Head Attention matrices leads to data movement bottleneck



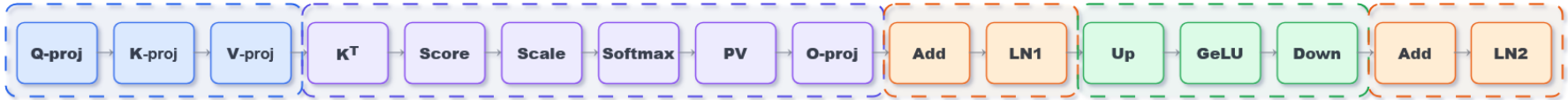
Runlist Strategy

- Method to dispatch nodes of transformer graph from host to NPU—sequentially executes operations in order submitted
- No need for synchronization between host and NPU with intermediate nodes



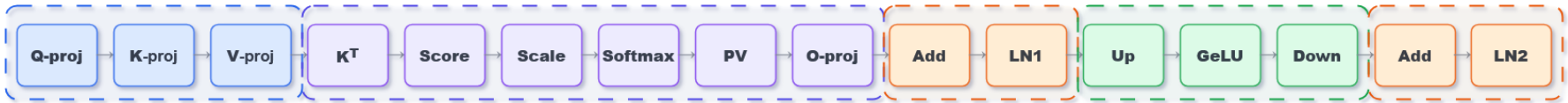
Runlist Strategy

- Benefits:
  - All 32 tiles available to execute each operation
  - Can focus on highly optimizing each operation on NPU
- Drawbacks:
  - Each operation in dispatch list incurs reconfiguration overhead
  - Quadratic scaling of Multi-Head Attention matrices leads to data movement bottleneck
  - Some operations have much higher arithmetic intensity compared to others, i.e. GEMMs vs Add—may not be worth overheads to run latter operations individually



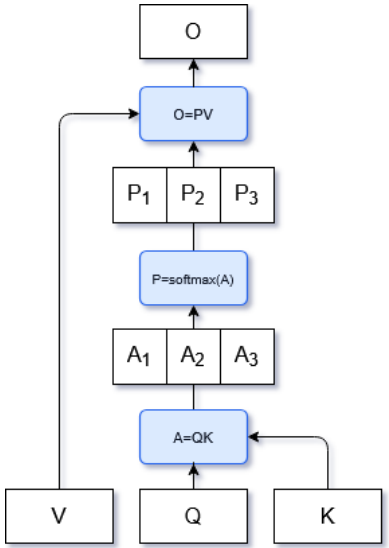
Transformer Runlist

- A naïve runlist implementation of transformer graph is used as one baseline
  - For matrix multiplications, we configure tiling and mapping with all 32 cores using accumulate-in-place strategy
  - For other operations, we maximize interface tile utilization for data movement between NPU and external memory, since arithmetic intensity compared to matrix multiplications is much lower



Dataflow Strategy

- Method to map nodes in transformer graph through processing elements in NPU
  - DMAs at each tile allows for overlap between compute and data movement
  - Combined with double buffering allows for pipelining between nodes
- Example: Pipelined attention, QK→softmax→PV
  - $O = PV$ (GEMM operation) needs 3 tiles for reduction, $P_1 - P_3$
  - Without pipelining, 1 output tile is generated every nine time steps
  - With pipelining at steady state, 1 output tile is generated every three time steps

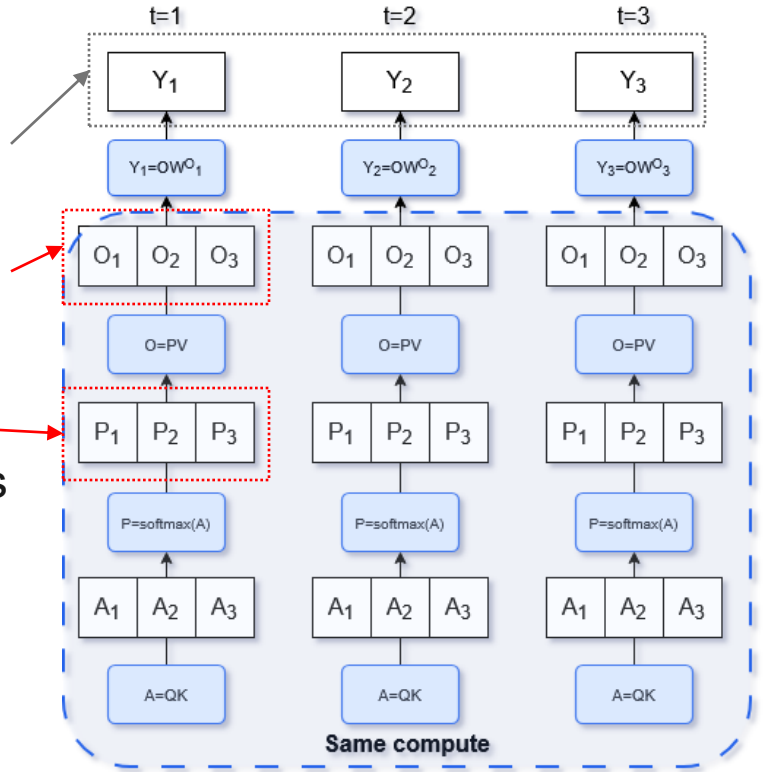


Dataflow Strategy

- Benefits
  - Fusing operations in the graph reduces reconfiguration overhead
  - Operations can be pipelined through NPU, which can increase throughput
- Drawbacks
  - Operations don't have access to all 32 cores for compute
  - Imbalanced pipeline leads to increased latency
  - Limited on-chip memory means pipelining needs careful mapping to avoid repeated compute

Pipelining Problem

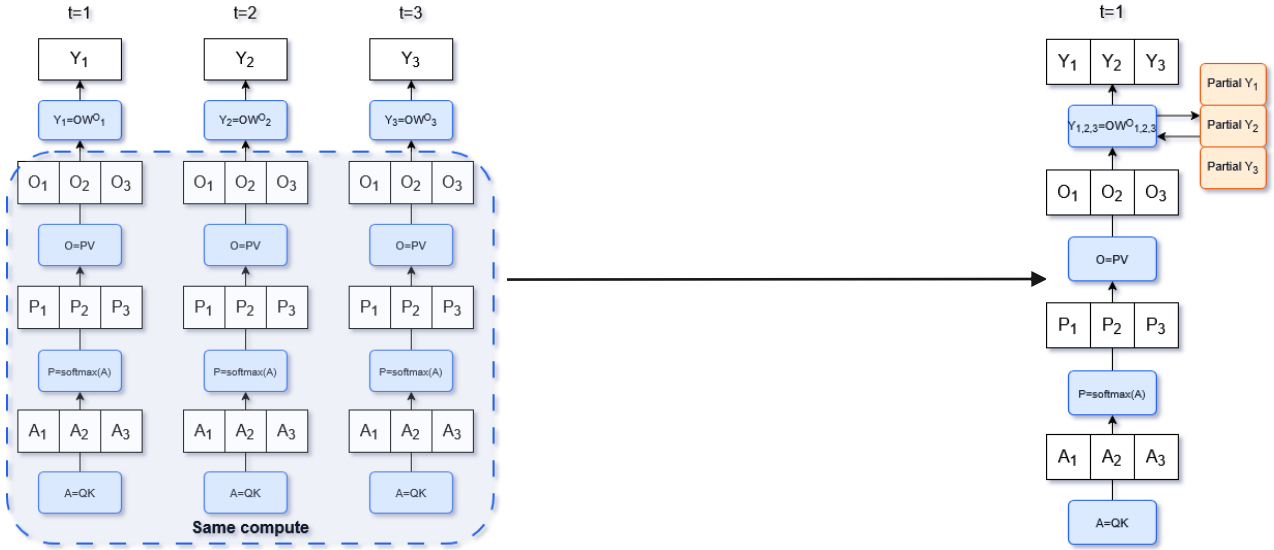
- Example: Generating Output Projection (GEMM) tiles $Y_1 - Y_3$
- Pipelining MHA to Output Projection requires data across the full reduction dimension, tiles $O_1 - O_3$
  - $O = PV$ still also needs $P_1 - P_3$ to generate output
- Assuming tiles can't stay on-chip, MHA stages need to recompute same data in order to generate Y_2 and Y_3
- Pipelining is appealing, but only useful when data can be utilized fully to avoid recompute



Proposed Implementation and Methodology

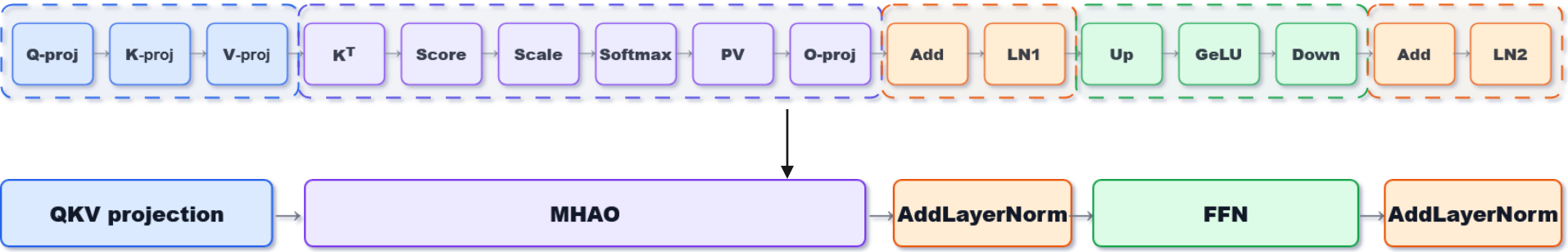
Proposed Solution to Pipelining Problem

- Preserve the accumulations of operations in larger shared scratchpad to improve reuse of intermediate data
- By placing partial accumulations in shared scratchpad, tiles O_1-O_3 can be reused such that tiles Y_1-Y_3 can be generated by the end of t=1
  - Takes advantage of last two stages being GEMMs that need tiles to loop across reduction dimension



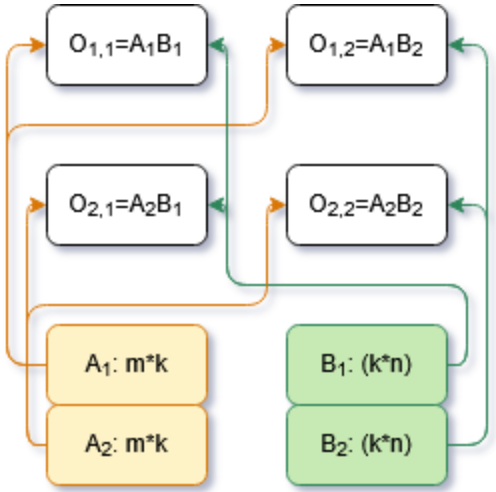
Hybrid Implementation For End-to-End Execution

- Combining the partial accumulation staging strategy with the runlist allows for fewer dispatches while reusing intermediate data as much as possible
- MHAO: Multi-Head Attention + Output Projection
- FFN: Feed-Forward Network



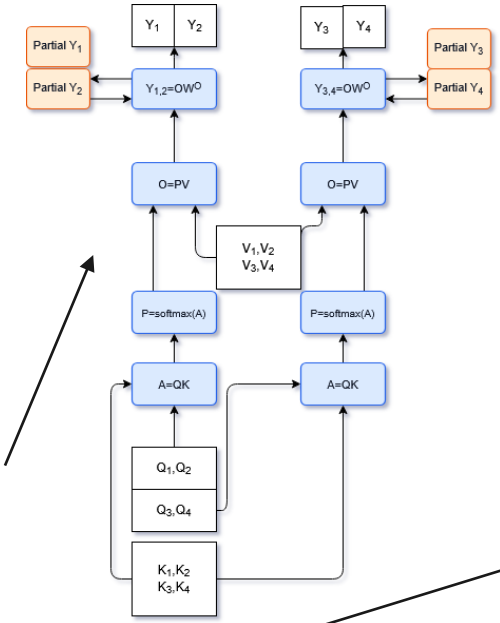
QKV Projection Mapping

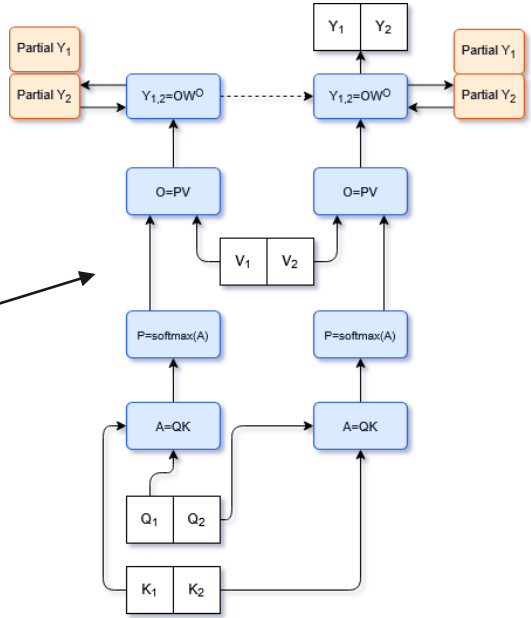
- Instead of individual GEMM executions, projections can be fused by treating them as a larger GEMM workload using accumulate-in-place:
  - $\mathrm{M} = seq\ length, K = embedding\ dim, N = 3 * embedding\ dim$
- Parallelized across sequence length *M* and embedding dimension *N*



Multi-Head Attention + Output Projection Mapping

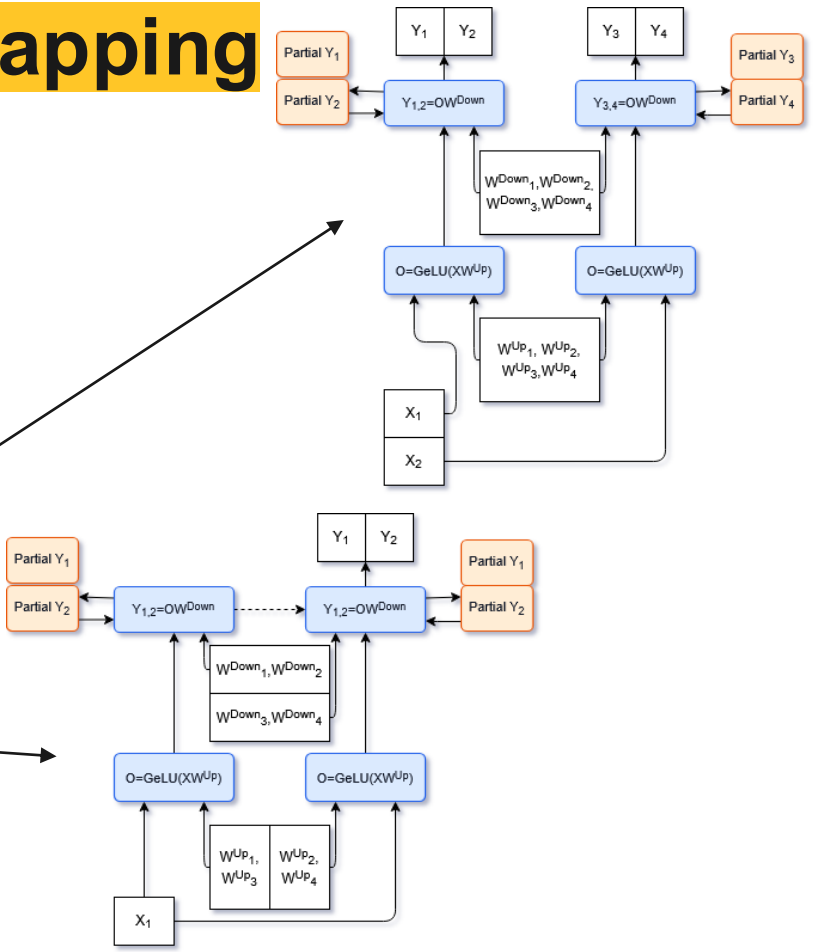
- Pipeline $Y = MHA * W^O$ in 4 stages, placing partial-accumulation in shared scratchpad
- Parallelized across sequence length and heads, can mix between the two
  - Parallelizing across sequence length for longer sequence lengths—better reuse of K/V/W^O tiles
  - Parallelizing across heads for sequence lengths shorter than tile size in sequence dimension—better NPU utilization, but requires reduction step across output projection tiles





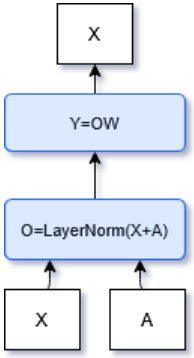
Feed-Forward Network Mapping

- Pipeline $Y = \left(GeLU(XW^{Up})\right)W^{Down}$ in 2 stages
  - Up projection uses accumulate-in-place
  - Down projection places partial-accumulation in shared scratchpad
- Parallelized across sequence length and FFN dimension
  - Parallelizing across sequence length for longer sequence lengths—better reuse of weight tiles
  - Parallelizing across FFN dimension for sequence lengths shorter than tile size in sequence dimension—better NPU utilization, but requires reduction step across down projection tiles



Residual Add + LayerNorm Mapping

- Pipelined Add and Weighted LayerNorm such that stage 1 is Residual Add + LayerNorm, and stage 2 is Elementwise Multiplication
  - Compile with LayerNorm weights (vector) written into local scratchpad
- Implemented algorithm to calculate maximum tile size possible that can be stored in local scratchpad based on sequence length and embedding dimension
- In general, the latency of this block is much less compared to other blocks



Evaluation

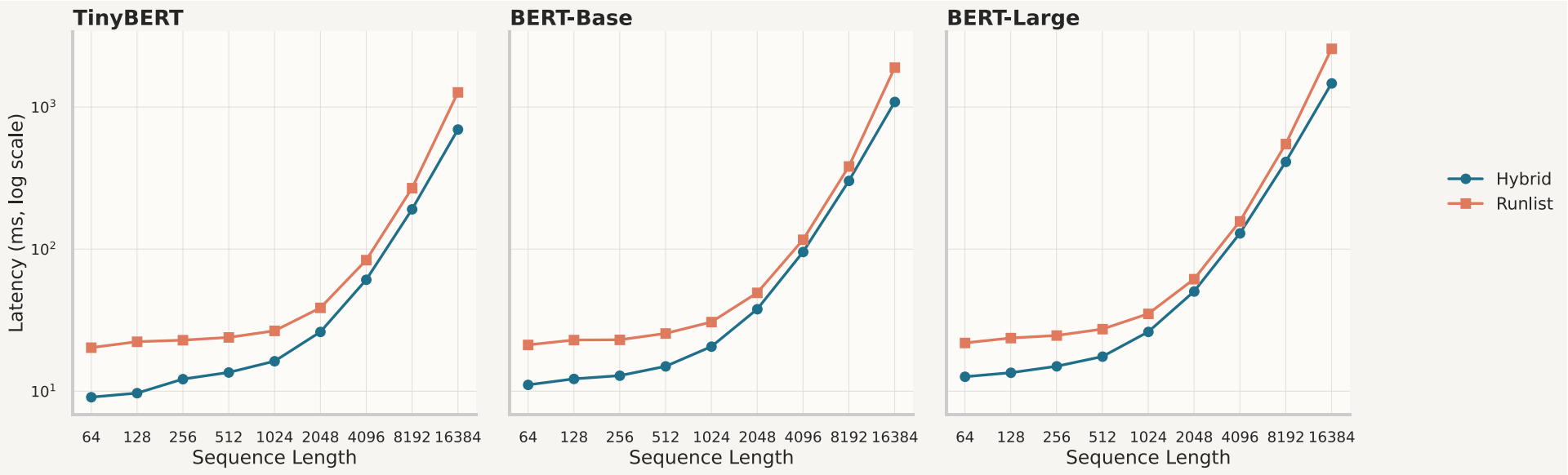
Evaluation Surface

Parameter	Evaluated surface
Model families	TinyBERT: $d_{emb} = 512$ BERT Base: $d_{emb} = 768$ BERT Large: $d_{emb} = 1024$
Sequence lengths	64 to 16384
Baselines	Naïve runlist, iGPU
Data type	bfloat16

- Hardware: Ryzen AI 9 HX 370
  - NPU: XDNA2
  - iGPU: Radeon 890M
- Power measurement
  - turbostat for NPU
  - ROCm for iGPU
- Sensitivity studies
  - Varying sequence length from 64 to 16384
  - Varying embedding dimension (different BERT sizes)
- Ablation studies
  - Evaluate reconfiguration overhead
  - Evaluate partial-accumulation staging improvement

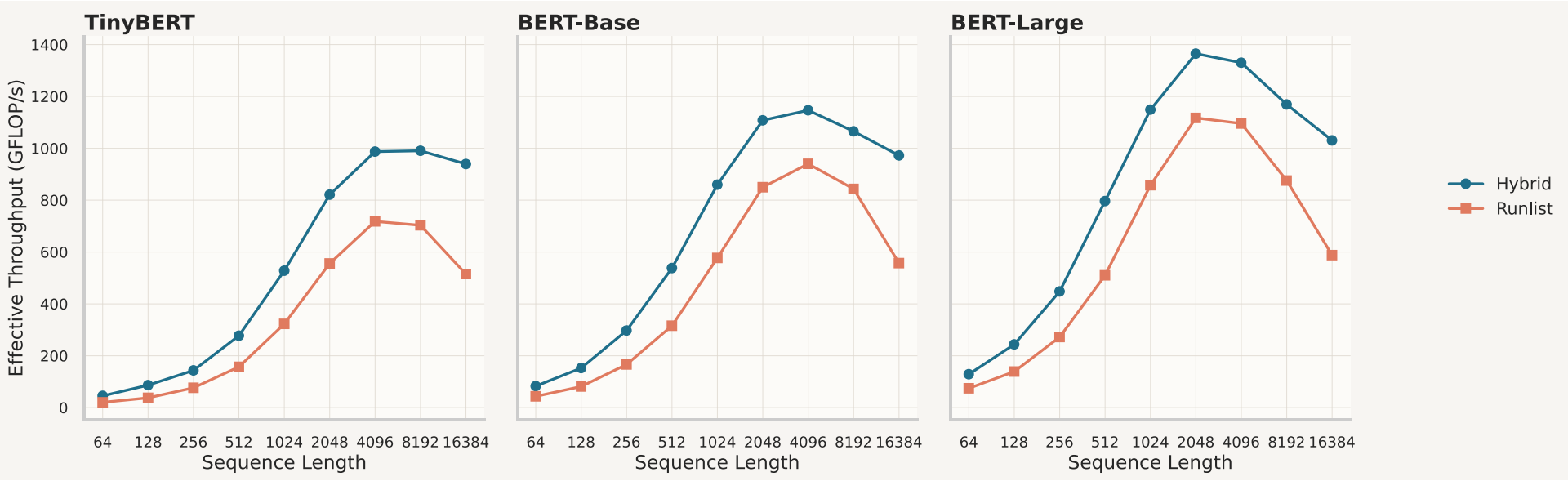
End-to-End Improvement Over Naïve Runlist

- Comparing latency of hybrid dataflow implementation with naïve runlist
- Takeaway: Hybrid implementation beats runlist over all measured points



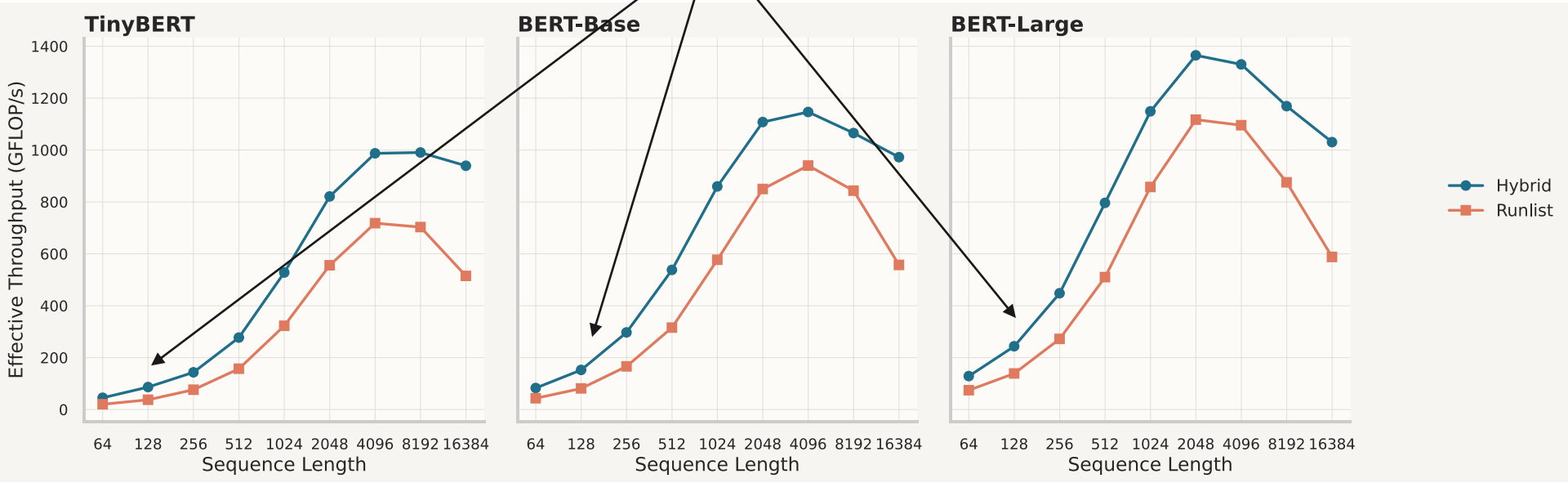
End-to-End Improvement Over Naïve Runlist

- Comparing throughput of hybrid dataflow implementation with naïve runlist
- Takeaway 1: Hybrid implementation beats runlist over all measured points



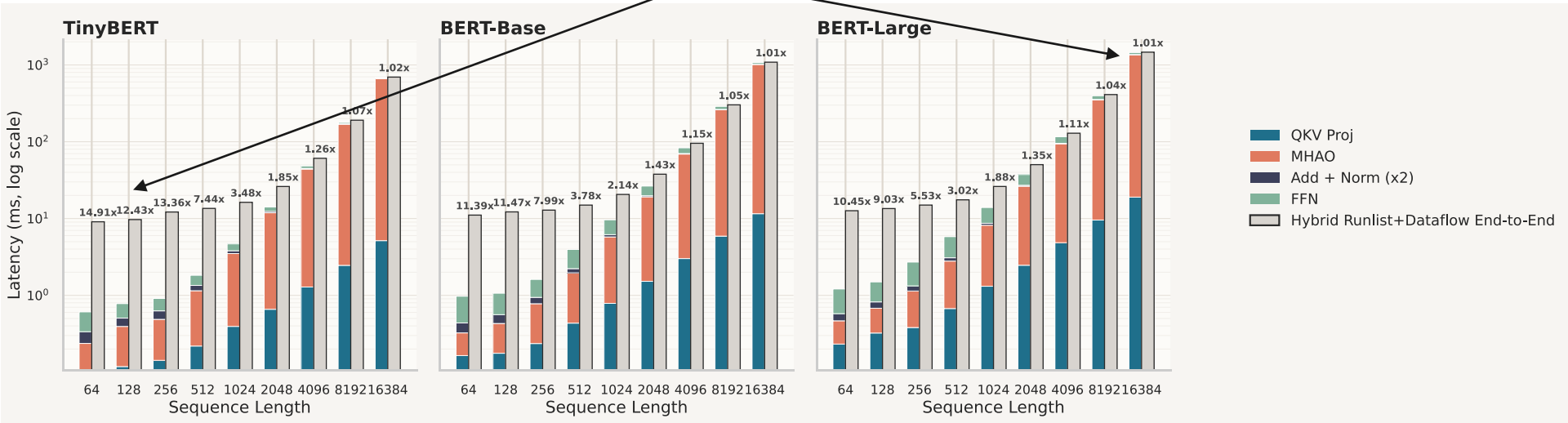
End-to-End Improvement Over Naïve Runlist

- Takeaway 2: Modest improvement seen at lower sequence lengths because reconfiguration overhead is not amortized over the whole execution



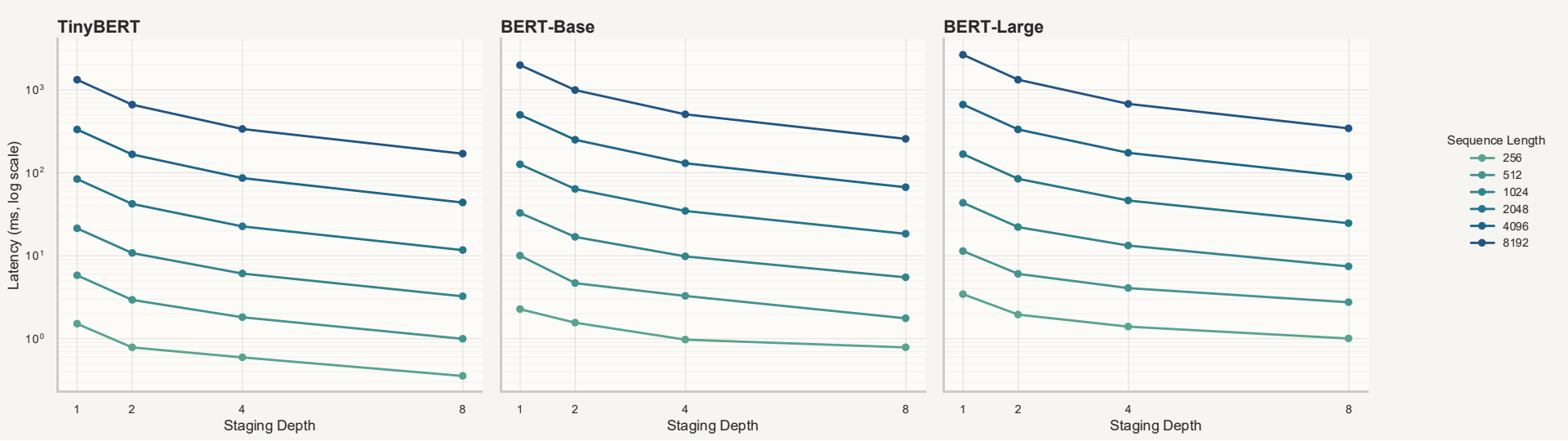
Reconfiguration Overhead

- Comparing the sum of individual kernel latencies with hybrid runlist + dataflow end-to-end latency
- Takeaway 1:
  - Reconfiguration overhead ranges from 15x to 1.01x across sequence lengths



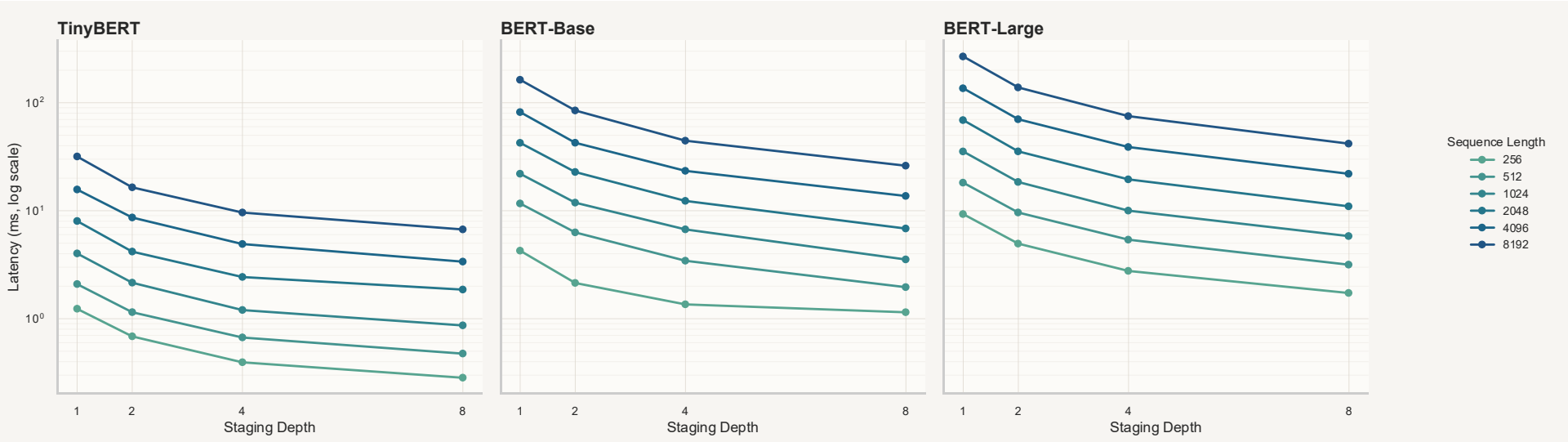
Partial-Accumulation Staging Improvement

- Comparing latency with varying MHA output reuse using partial-accumulation staging method
- Takeaway 1:
  - Better reuse (larger staging depth) of MHA output tiles to Output Projection improved latency by a factor of up to 8x



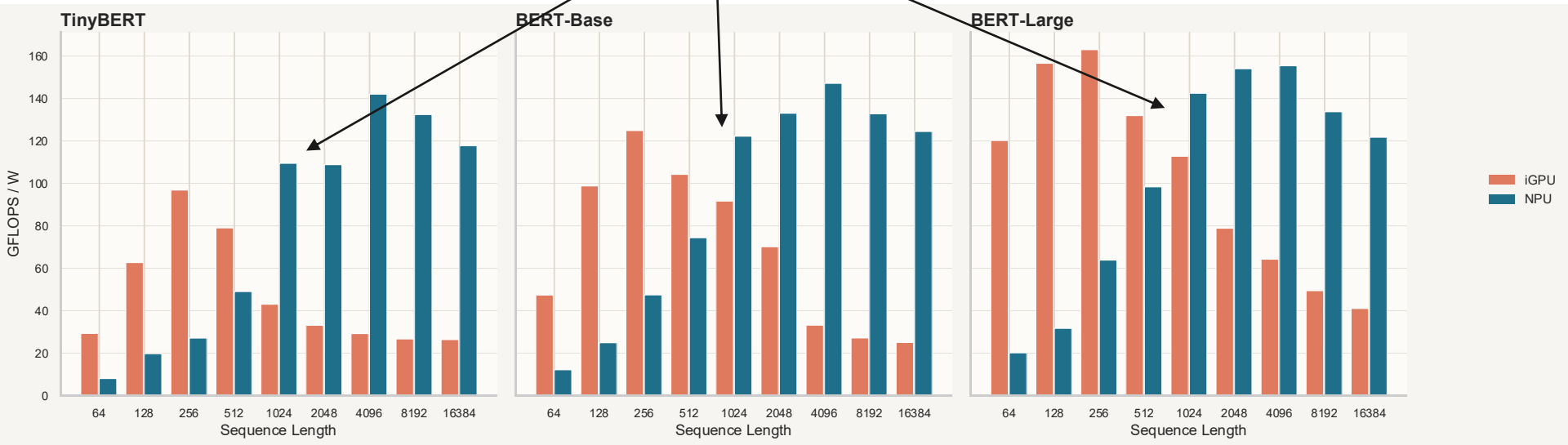
Partial-Accumulation Staging Improvement

- Comparing FFN with and without reuse using partial-accumulation staging method
- Takeaway 2:
  - Better reuse (larger staging depth) of Up Projection output tiles to Down Projection improved latency by a factor up to 7x



Energy Efficiency Compared to iGPU

- Comparing energy efficiency of hybrid implementation with iGPU
- Takeaway: NPU is more energy efficient than iGPU after 1024 tokens
  - Due to better throughput at longer sequence lengths and lower energy consumption
  - Reconfiguration overhead makes it less efficient than iGPU at lower sequence lengths



Conclusion

- Rising datacenter costs makes local inference an appealing alternative, and NPUs are a strong target for energy-efficient inference
- Partial-accumulation staging strategy is an effective strategy for improving reuse in pipelined implementations, showed latency improvement by 3x-8x
- BERT execution on NPU with hybrid implementation consistently beat naïve runlist in throughput and beat iGPU in energy efficiency outside of short sequence lengths
- Future work will be to evaluate dataflow and runlist patterns with offload pattern to determine tradeoffs between three execution strategies on the NPU