Exploiting Activation Sparsity for Efficient LLM Inference on NPU

Harrison Zhang
Intern Presentation

AMD
together we advance_

Contents

About Me

- **Undergraduate at Harvard College**
  - Third year
  - AB/SM candidate in Computer Science and Mathematics
- **Researcher at MIT Computer Science and Artificial Intelligence Laboratory (CSAIL)**
  - Sparse approximate-NN search with ray tracing primitives
  - Quantized CPU-GPU heterogenous inference engine for RecSys
- **Intern at AMD**

Where does sparsity come from?

Traditionally, LLMs are not sparse…

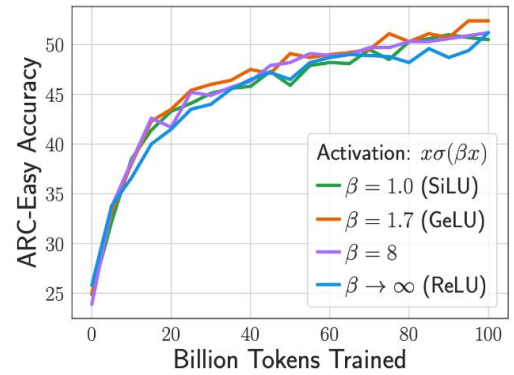
Research frontier: Introduce activation sparsity

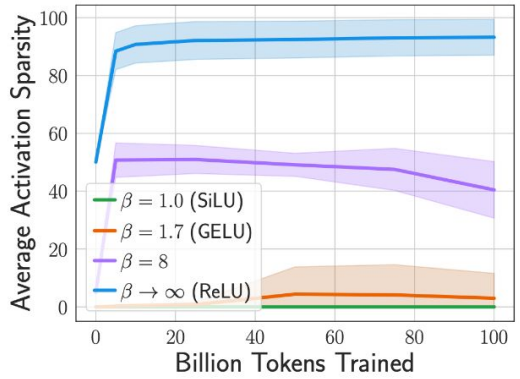
- **Swap activation map with ReLU** + additional training
- **Little drop in accuracy** on common benchmarks
  - ReLU Strikes Back (ICLR 2024)
  - ProSparse (arXiV 2024)
  - CATS (arXiV 2024)

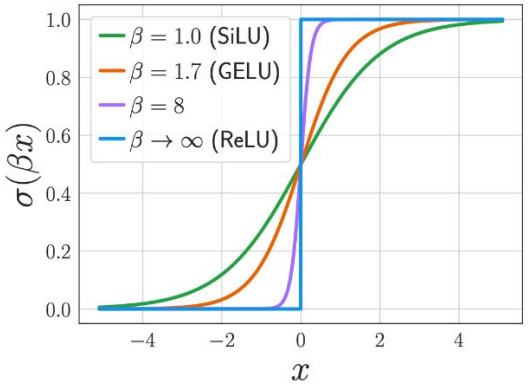
Sparse foundation models already available (Hugging Face)

- ProSparse-LLaMa-2-7B
- Meta AI OPT-6.7B
- Mixtral-8x7B
- Mixtral-8x22B
- ReluFalcon-40B
- And many more…

Training Meta AI OPT-6.7B







Figures from: Mirzadeh et al. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. ICLR 2024.

Where does sparsity come from?

ProSparse

- **Progressive sparsity regularization**
  - Gradually increases sparsity without degrading model performance
- **Activation threshold shifting**
  - Progressively increase sparsity in multiple stages to ensure adaptation to high sparsity

ReLU Strikes Back

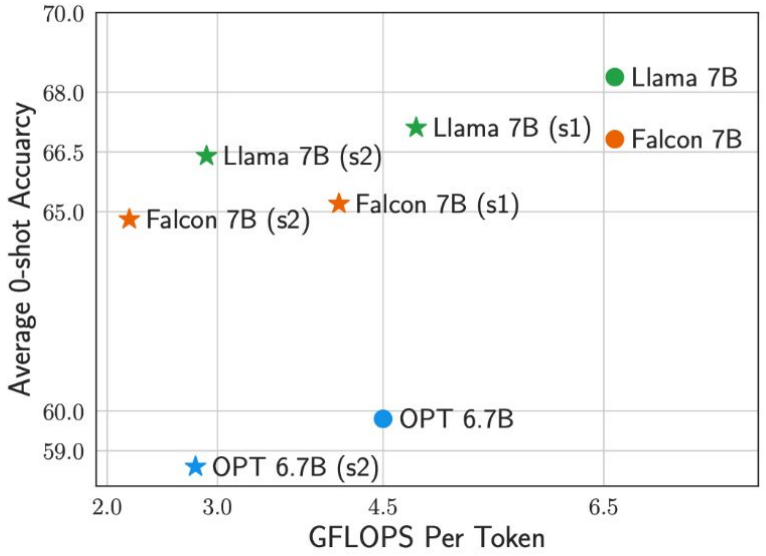
- **ReLU activation map** when training from scratch
  - Activation map has negligible impact on model performance
- **ReLUfication** for already-trained models
  - Change activation map to ReLU and fine-tune for a short time

More than 90% exploitable sparsity in FFN for performance gain

- Reduce compute (FLOPs)
- **Reduce I/O transfers**

Model	Input Sparsity (%)			FLOPS (G)	Avg 0-shot Acc %
	QKV	UpProj	DownProj		
Llama 7B	0	0	0	6.6	68.4
Llama 7B (relufied-s1)	0	0	62	4.8	67.1
Llama 7B (relufied-s2)	51	67	65	2.9	66.4
Falcon 7B	0	1	0	6.6	66.8
Falcon 7B (relufied-s1)	0	0	94	4.1	65.2
Falcon 7B (relufied-s2)	56	56	95	2.2	64.8
OPT 6.7B	0	0	97	4.5	59.8
OPT 6.7B (relufied-s2)	50	40	97	2.8	58.6

Exploitable sparsity



Figures from: Mirzadeh et al. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. ICLR 2024.

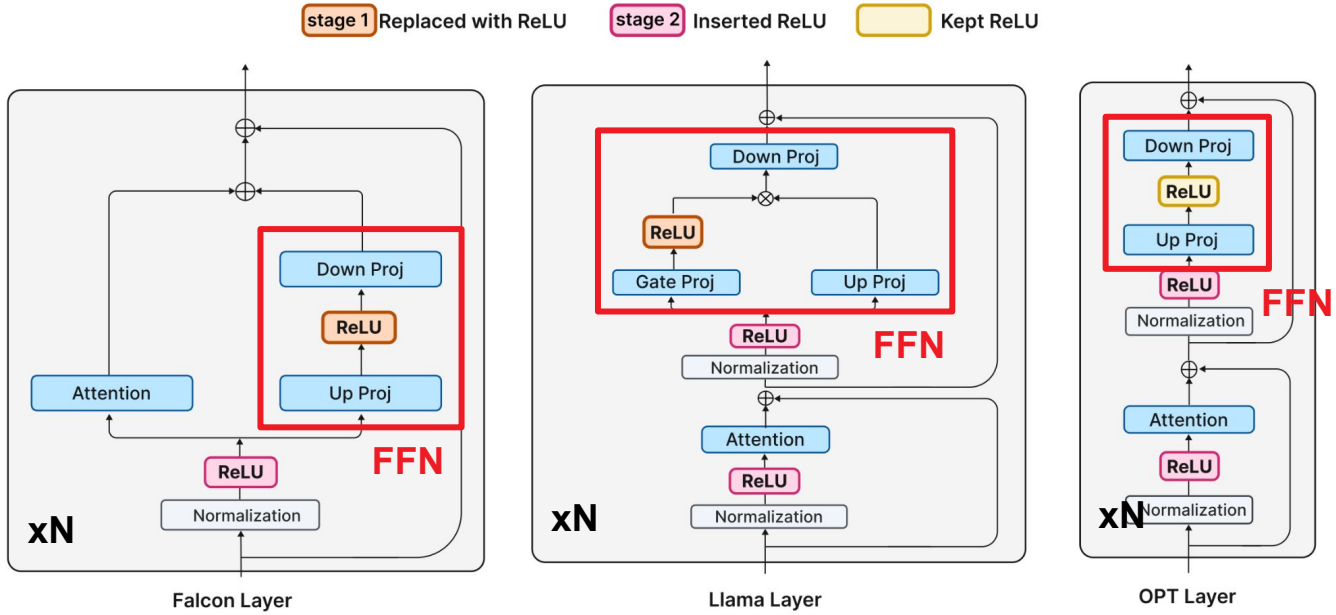
How to realize performance gains?

ReLUfication

- Replace or insert ReLU in FFN
- Sparse input to down projection

Selective loads

- Inference usually bottlenecked by I/O
- Down projection step
- Weights in DDR, activations in L2 (from previous layer operations)
- **Core idea: Load only rows that correspond to activated elements in *up projection* for I/O performance gain**



$$\mathrm{FFN}(x) = \boxed{\mathrm{ReLU}(xW_1 + b_1)W_2} + b_2$$

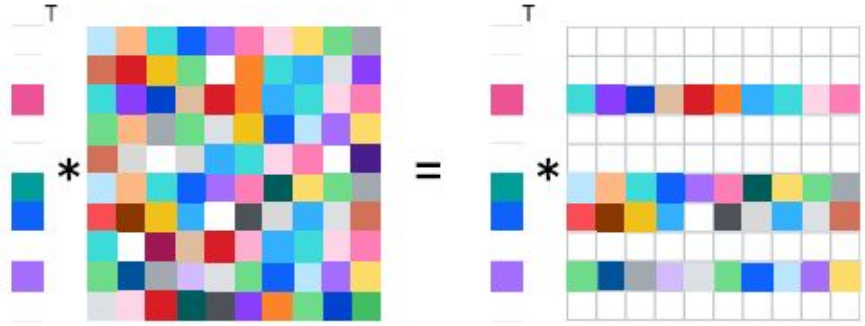
ReLUfied up projection:
Sparse-vector GEMV
down projection

Figures from: Mirzadeh et al. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. ICLR 2024.

Methodology

Our enhancement

- Given a DMA descriptor (BD) for transferring the weight matrix, transfer weights from DDR to L2
  - **1. Read activation vector** from L2
  - **2. Build row transfer bitmap**
  - **3. Dynamically create DMA descriptors** for row transfers with nonzero corresponding activations
- Leads to lower average latency for token generation



Only transfer row of weight matrix if corresponding activation vector element is activated…

Output is equivalent, with less required I/O

Model	Avg Acc.	Params	Act. Params	Gains
SparseLLM/ReluLLaMA-7B(fp16), Dense	55.517	6.74 B	6.74 B	N/A
SparseLLM/ReluLLaMA-7B(fp16), Sparse	55.517	6.74 B	5.63 B	16.5%
SparseLLM/ReluLLaMA-13B(fp16), Dense	57.688	13.02 B	13.02 B	N/A
SparseLLM/ReluLLaMA-13B(fp16), Sparse	57.688	13.02 B	10.94 B	16.3%

Fewer effective parameters by considering sparsity

Timeline

✅ **Phase 0: Program AIE array to identify ideal speedup**

- Passthrough with Xilinx Runtime overhead, synchronization, etc.

✅ **Phase 1: Develop fixed-sparsity NPU firmware prototypes**

- Co-design passthrough synchronization logic and NPU firmware
- Firmware images have fixed sparsity patterns

Phase 2 (future work): Build end-to-end NPU firmware image

- Activation vector loading and scanning with dynamic BD generation

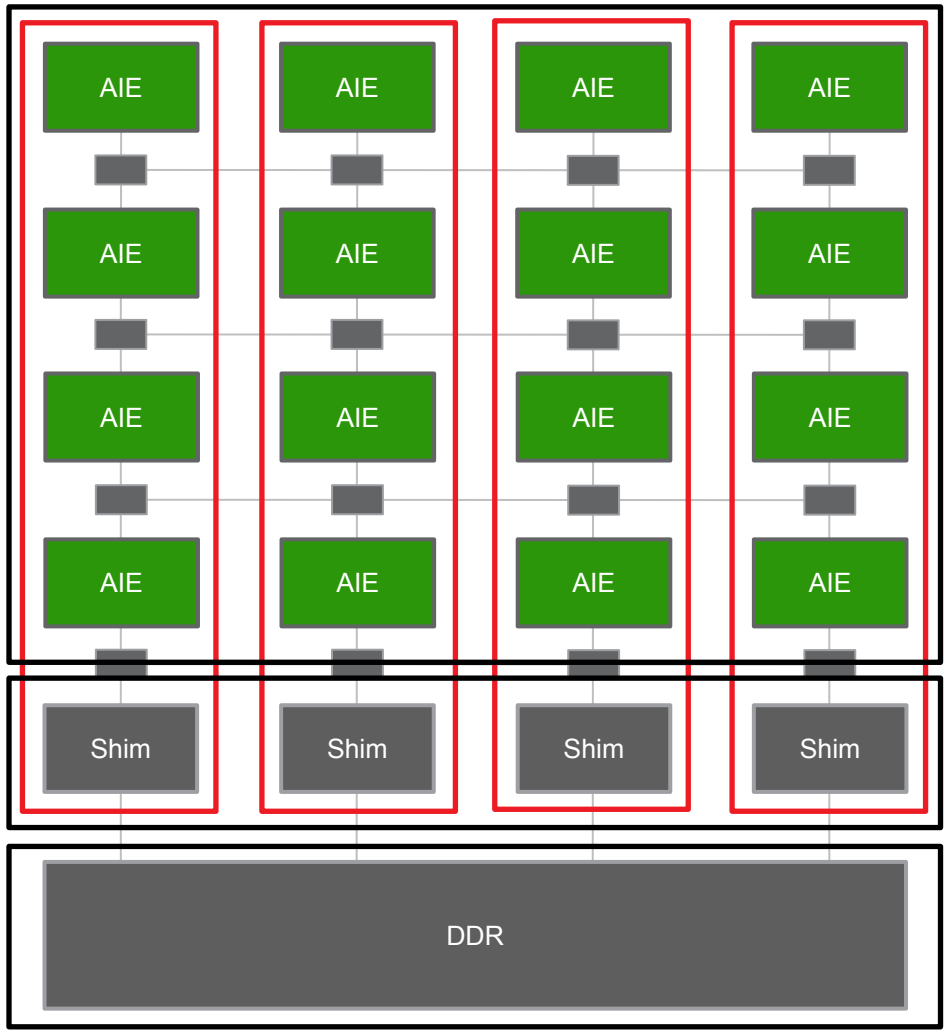
Figures from: Guanchen Li, Dong Li. AIG-Models.

Methodology

Tools

- **MLIR-AIE / IRON toolchain (Research and Advanced Development)**
  - Accelerator programming via Python interface
  - Compilation flow enables faster testing and prototyping
  - Co-design application synchronization with sparse BD selection, for experimentation only
  - Testbench provides built-in data integrity checks
- **NPU firmware source**
  - Production firmware hacking in C, conforming to underlying programming model
  - DDR runtime patching: Select, enqueue, execute certain BDs
  - Phoenix → Strix

AIE2 Architecture



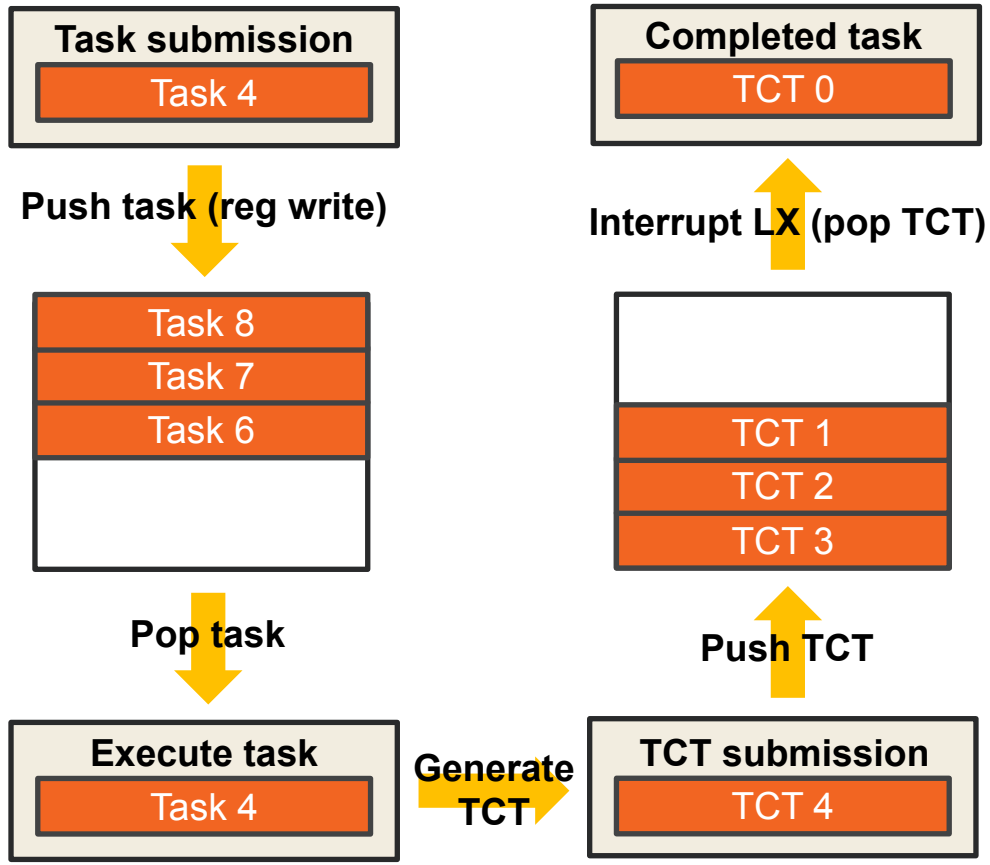
Data parallel

L1

L2

L3

Hardware FIFO Queue



Experiments

Performed on AIE2: Phoenix, Strix

Phase 0 (done)	**Program AIE array using IRON** to identify speedup	Perfect BD utilization	BD execution is statically determined
Phase 1 (done)	**Develop fixed-sparsity-pattern firmware images.** Creates one BD per row statically, then filters for desired row execution at runtime	Sparse BD (under)utilization – create BDs for each row	BD execution is dynamically determined – only desired BDs are enqueued
Phase 2 (future work)	**Develop firmware image that scans activation pattern**, then dynamically generates BDs for desired rows	Perfect BD utilization – create BDs only when needed	BD execution is dynamically determined

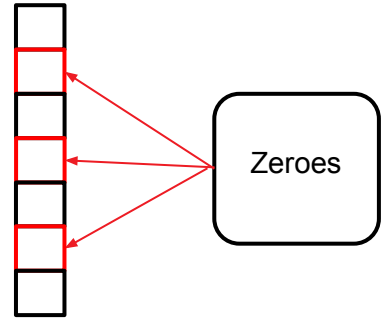
Phase 0

Objective: Show I/O performance gain with increasing sparsity and **perfect BD execution** (including associated overheads)

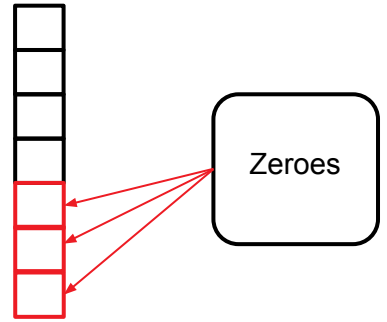
End-to-end passthrough kernel flow

- **Define the instruction sequence in application code**
  - DMA tiles to use
  - Source and destination tensor addresses
  - Rows to transfer, size of transfers
  - Synchronization on task completion token (TCT)
- **Setup**
  - **Worse-case pattern:** alternatingly transfer one row for every *s* rows
  - Create one BD **for each desired row**, at compile-time

Details: Static activation pattern, perfect BD utilization and execution



Worst case: Zeroes are uniformly distributed within activation vector



Best case: Zeroes are contiguous on either end of activation vector

Phase 1

Objective: Show I/O performance gain with increasing sparsity and **online-selective BD execution** (including associated overheads)

End-to-end passthrough kernel flow

- **Define the instruction sequence in application code**
  - (same as before)
- **Setup**
  - Create one BD **for each row**, at compile-time
- **Core BD firmware execution loop**
  - Patch runtime tensor address for source and destination
  - Create task completion token (TCT)
  - Submit this BD to the task queue and execute
  - Synchronize with TCT

Our enhancement: online conditional BD execution on a (fixed) activation pattern

Details: Fixed activation pattern (per firmware image), BD underutilization: statically generate BDs for each row, selectively enqueue BDs at runtime

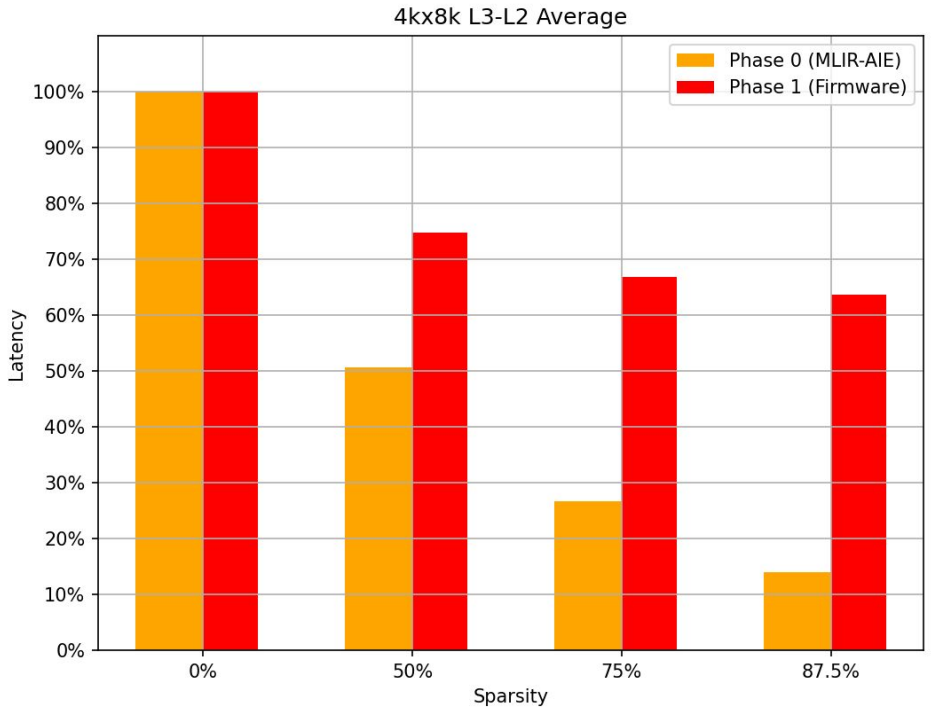
Results

- **Phase 0: Near-perfect linear speedup** in FFN
- **Phase 1: Up to 37% performance gain** in FFN using online selective BD execution, even with BD underutilization

- Scales linearly for sufficiently large column dimensions and sufficiently filled BDs
- Enqueuing overhead is small

AIE4

- Expect much better results due to per-column control processors
- Easier to manage task queue to maximize throughput and performance



Prototype (Phase 1) Limitations

BD underutilization – one for each row at compile-time

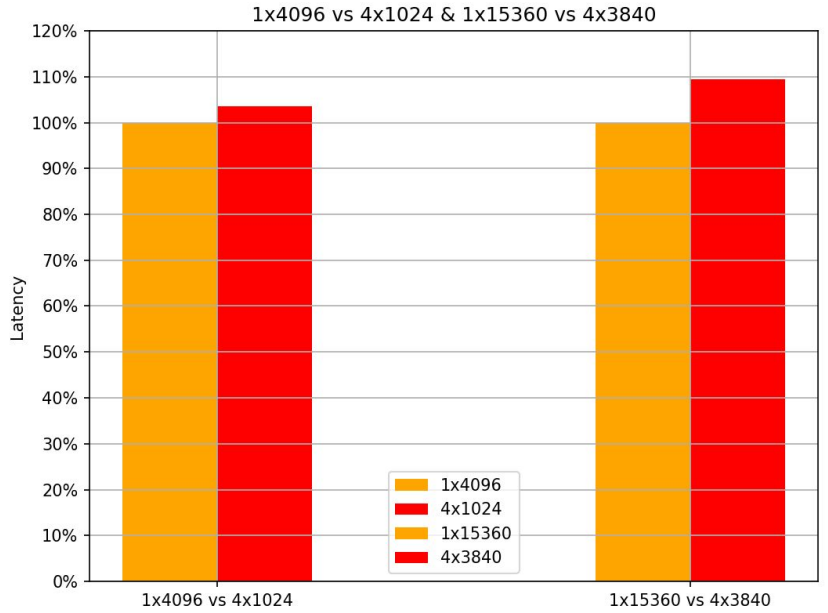
- Still admits performance gains, despite overhead
- For sufficiently small adjacent rows, batch into "island" transfers

Fixed activation sparsity patterns in firmware image

- Addressed in Phase 2 (dynamic BD generation)

Excessive overhead

- Run through the entire BD payload buffer (one BD for each row at compile-time) to decide enqueues
- Addressed in Phase 2



Equivalent transfers with one BD and multiple BDs on one AIE array column…

Transferring with row-wise BDs admits small overhead

Phase 2 (future work)

Proposed algorithm

1. **Store activation vector location, length**
   - Pack within the first runtime patch instruction
2. **Receive original BD in the first patch instruction**
   - This encodes a weight transfer for down projection
3. **Read from activation vector location**
   - Build bitmap of row transfer indicators
4. **Do until weight transfer is complete (in batches of 16 BDs):**
   - Synchronize transfers on respective TCTs
   - Encode desired row transfer information in the BDs
   - Enqueue and execute the BDs
5. **Skip the original BD transfer**

Expected Phase 2 Performance is closer to Phase 0

- Only overhead is scanning activation vector and constructing transfer bitmap
- Optimized: batch reads of 16 bytes from L2 to inform construction of 16 BDs (per-column maximum)

Experimental Takeaways

- **Even with BD underutilization, prototypes show performance gain**
  - Diminishing returns with increasing sparsity due to overhead
- **Performance is a function of number of row transfers**
  - Significant gains by exploiting I/O sparsity
- **Weight dimensions are re-interpretable**
  - Rows can be transferred in batches
  - E.g. batch-transfer rows regardless of activations, if faster than executing 2+ transfers
  - Optimal row-wise tiling approach is needed
- **Optimized BD utilization**
  - Pack BDs to ensure they are fully utilized
  - Initialize multiple BD transfers, then synchronize separately
- **Our enhancement is independent and modular – implementable with…**
  - Speculative decoding
  - Accumulator aware quantization
  - And many more… (see NPU Tech Forum)
- **Exploit activation sparsity to maximize dataflow throughput and performance**

Challenges

- Learning AIE architecture, IRON tooling
- Scattered access permissions for submodules
- Lack of documentation
- Long build processes (slower prototyping)

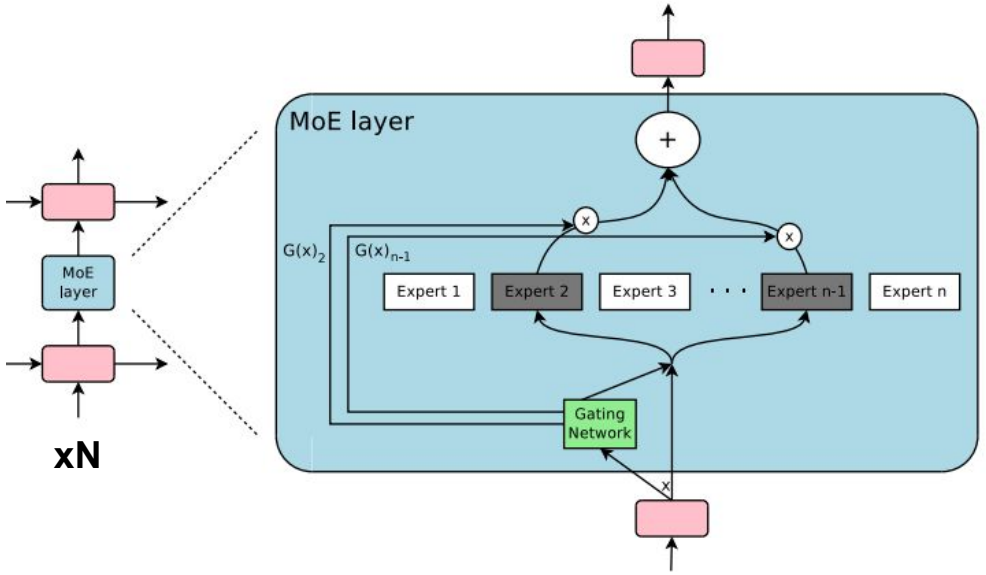
Impact at Research Frontier

Scalable to multiagent ensembles

- Intra-agent optimization
  - Performance gains within each node of agentic network
  - Reduces span
- Significant performance gain with increasing introduction of sparsity in FFNs

Scalable to Mixture of Experts

- Experts are hierarchical MoEs, which use FFNs at the core
- Lower-level weight fetching is usually on-demand
  - Need dynamic BD generation
  - Little model gating or routing in lower levels (weights are not preloaded)



Figures from Hugging Face: https://huggingface.co/blog/moe


AMD