Large Language Models on CPUs

Dibakar Gope, David Mansell, Ian Bratt

Outline

- Introduction to LLMs
- Optimized GEMV and GEMM kernels for 4b LLMs
- Single inference on AWS Graviton3
- Batched inference on Graviton3
- Conclusion and future work

Background

- LLMs have transformed the way we think about language understanding and generation
- Facilitating their efficient execution on Arm CPUs will expand their reach to billions of Arm devices
- LLMs are often BW bound and have a large weight memory footprint – CPU can achieve competitive performance
- CPU provides portability and flexibility – SW compression schemes, etc.
- Question: What is the potential performance of LLMs on Arm CPUs for single and batch inference cases?

Key results – LLMs on CPUs

- Focusing on LLaMA2 7B 4b quantized (Q4) model as a benchmark

- Llama.cpp (GGML) c/c++ runtime demonstrates performance on existing Arm platforms but fails to demonstrate the true potential of Arm CPUs

- Developed highly optimized blocked Q4 kernels for non-batched and batched inferences

- End-to-end LLaMA2 7B 4bit Speedup on Graviton3 (Neoverse V1):

  Single inference case: 35 tokens/s for 8 threads, 3.15x over GGML

  Batched inference case: 200+ tokens/s for batch size = 8,

  2.03x for BS=8 over optimized GEMV, 4.37x over GGML (BS=1)

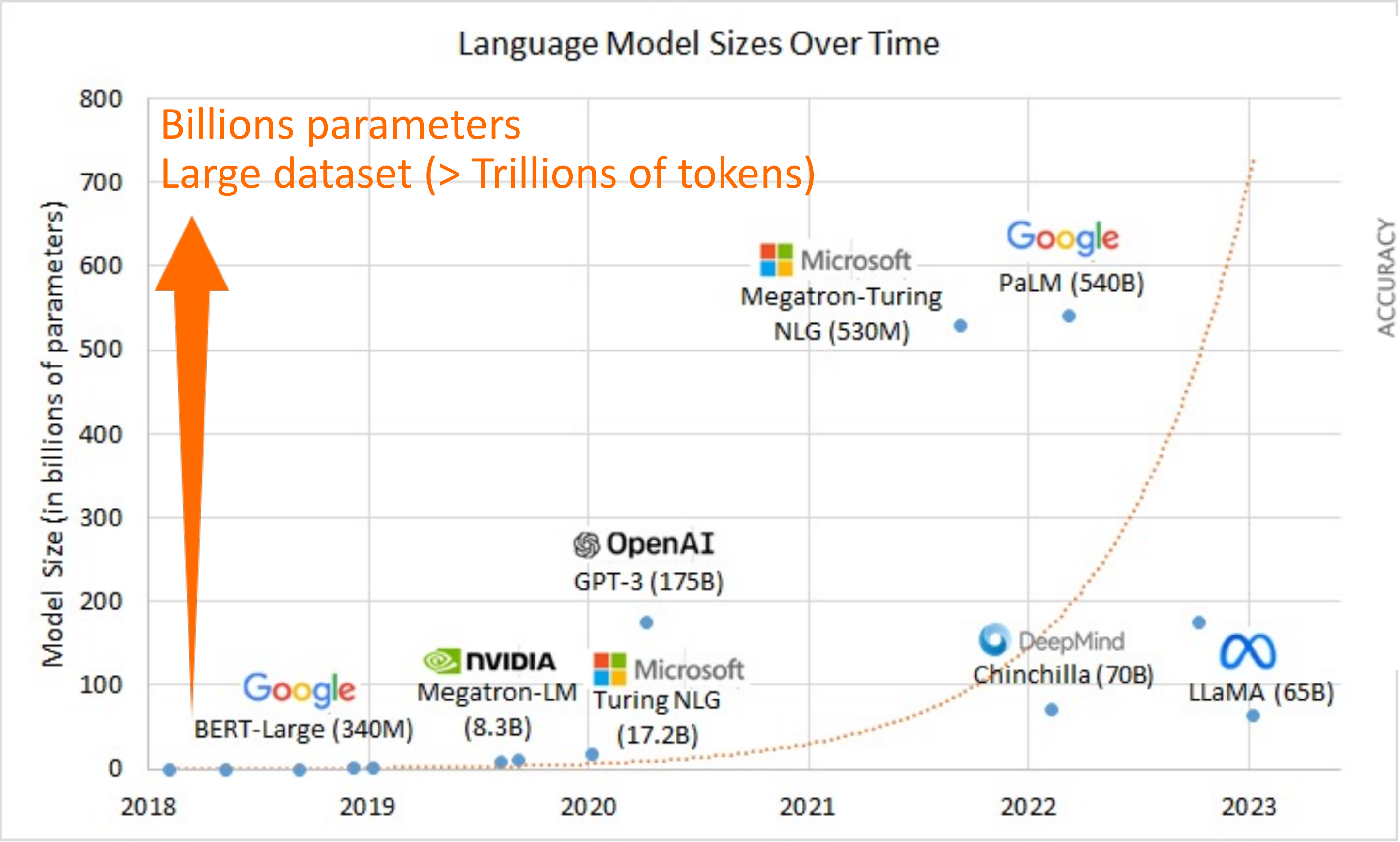
*BS = batch size

Evolution of Large Language Models

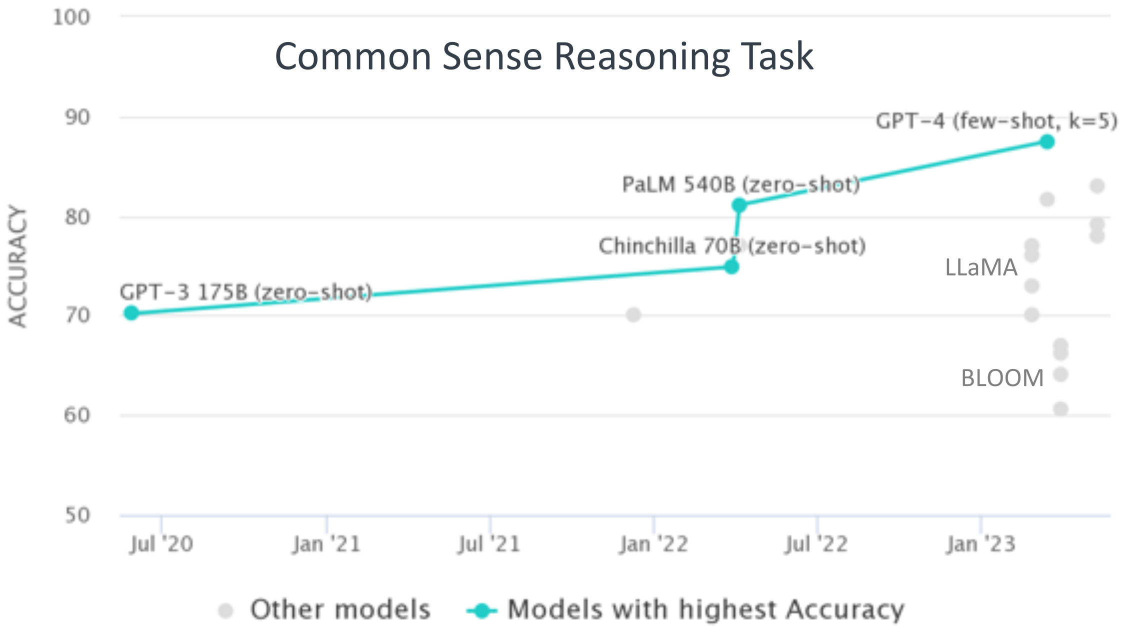
What are large language models (LLMs)?

- A language model can predict the next word given a context or a question.
- Large language models are trained with massive amounts of data to learn language patterns
- Perform tasks ranging from summarizing and translating texts to responding in chatbot conversations
- Basically, anything that requires language analysis of some sort.

Evolution of large language models



Billions parameters
Large dataset (> Trillions of tokens)

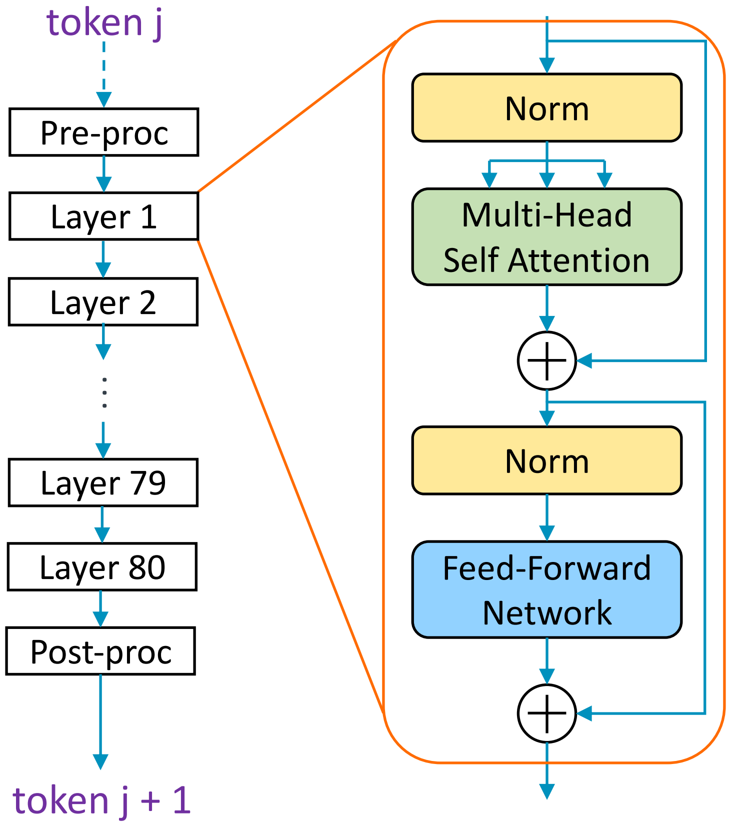


Why Scale Language Models?

- ✓ Bigger is Better!
- ✓ Performance continues to improve

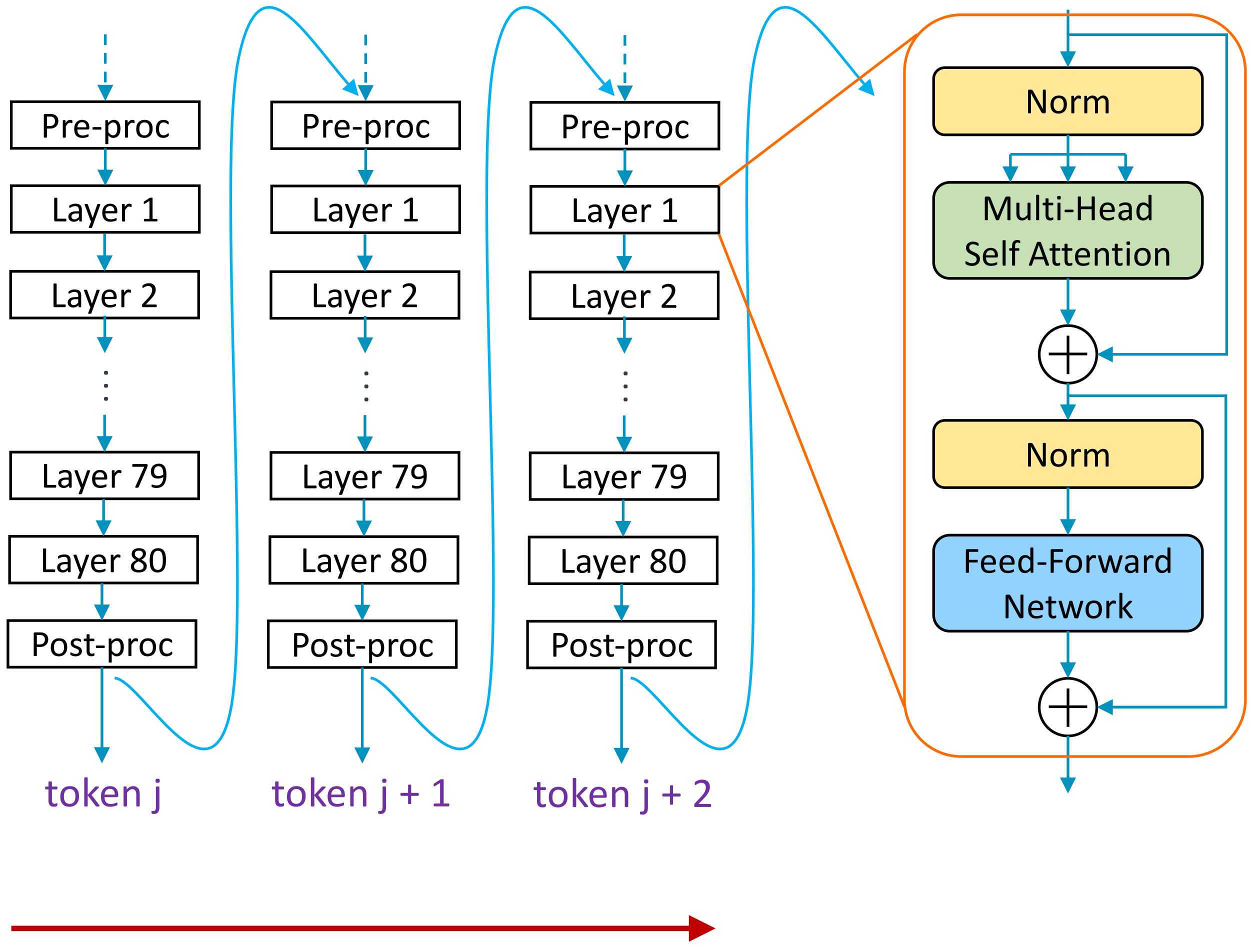
How does LLM work?

Meta’s 7B-70B parameter LLaMA 2 LLM



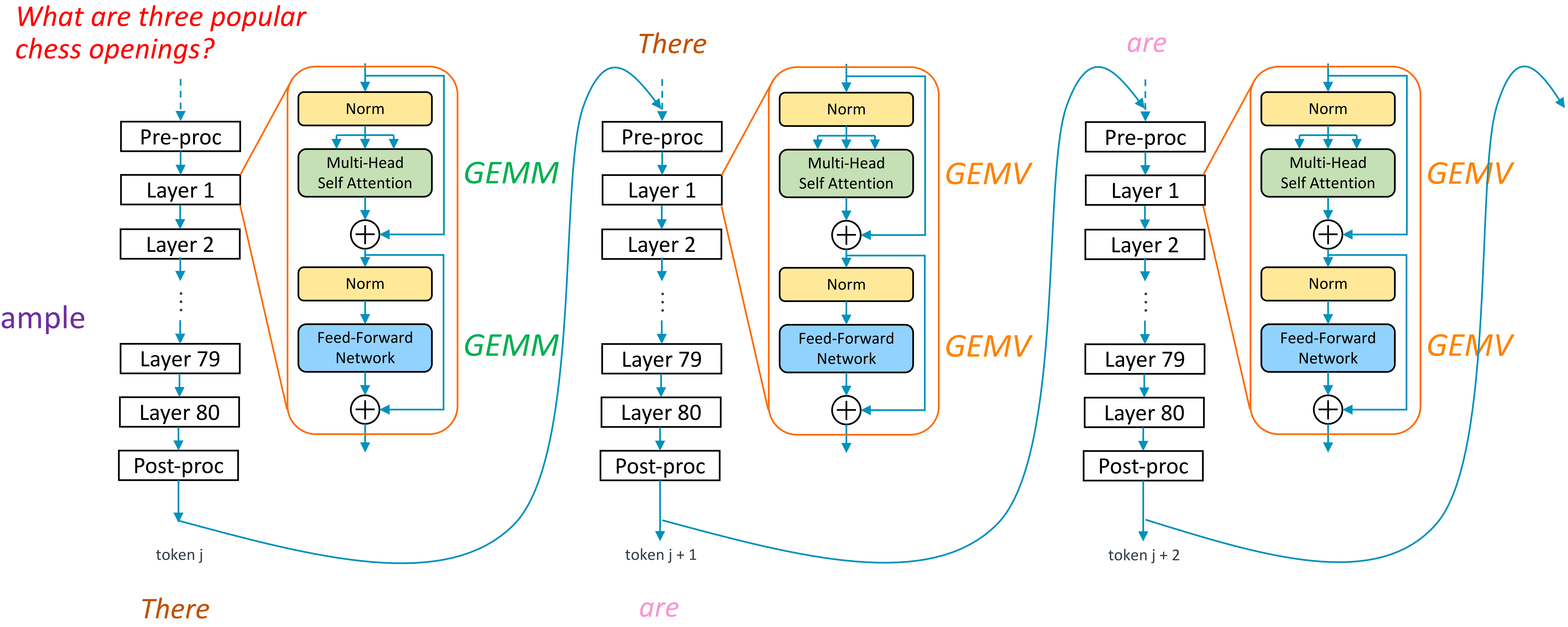
- LLaMA 2 is a well-known open-source LLM released by Meta
  - Llama.cpp is a popular open-source framework for quantizing and running it.
- A stack of self-attention transformer layers
- 80 layers in cascade for LLaMA 2 70B model,
- 32 layers for LLaMA 2 7B model

Meta’s 7B-70B parameter LLaMA 2 LLM



- Each round through the network generates a new token.
- The new token is fed into the network's next round.
- The "state" gradually builds up and is carried from left to right in the figure (through LLM's Key Value cache).
- A typical LLM inference involves going through the network multiple times and generating many tokens.

Example



LLM's key value cache:

1^{st} round: -

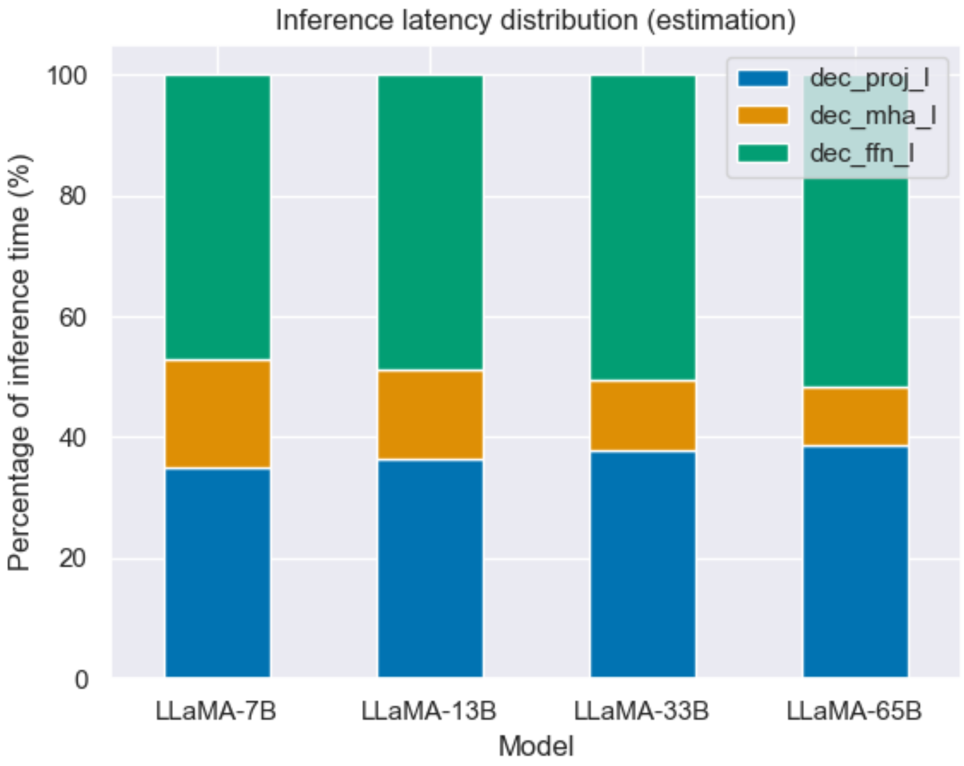
2^{nd} round: What are three popular chess openings?

3^{rd} round: What are three popular chess openings? There

…

Nth round: What are three popular chess openings? There are …

Non-batched inference of LLMs – generative phase

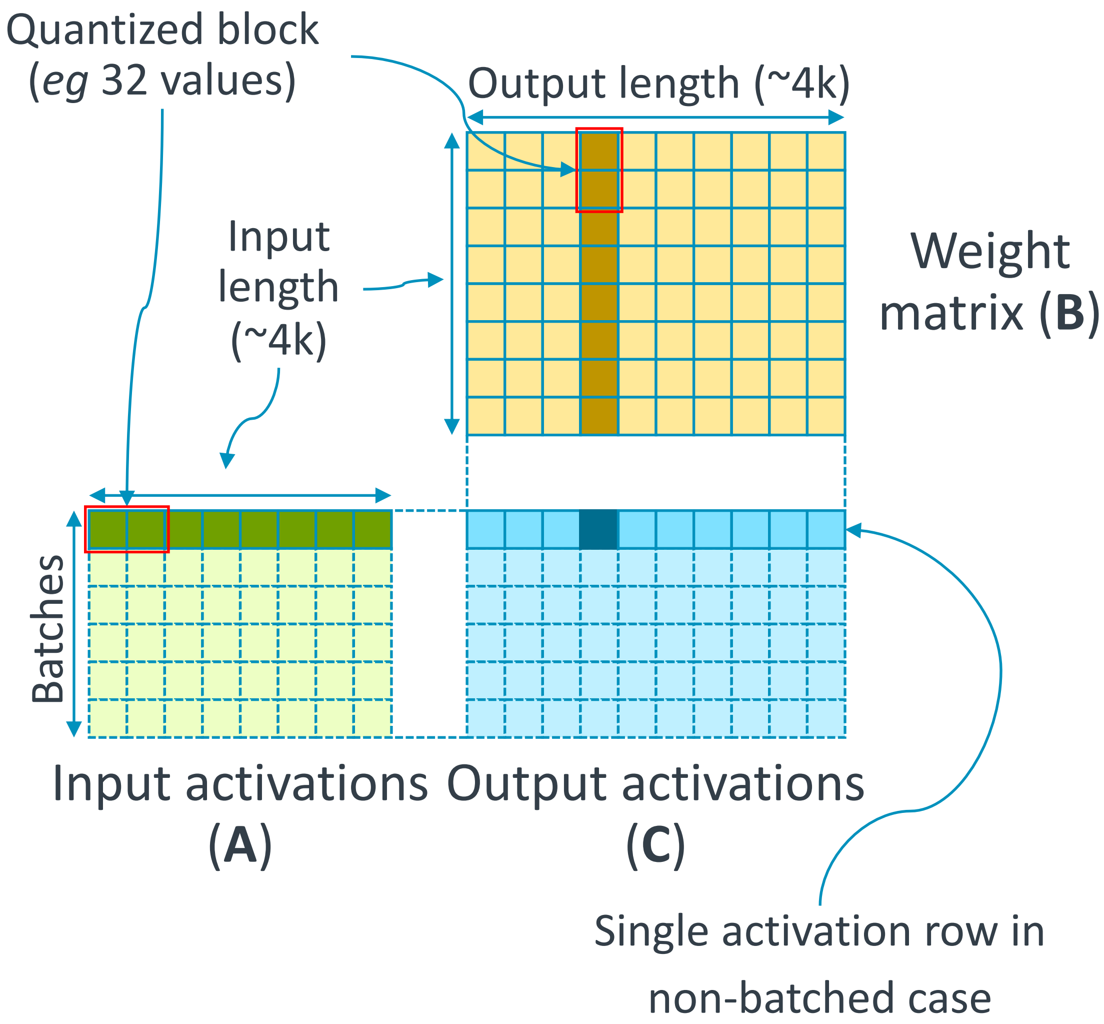


- Runtime dominated by the projection and feed forward layers
- Projection and feed forward layers are GEMVs for non-batched single inference, MHA is GEMM
- All layers are GEMMs for batched inference
- Memory-bound problem with memory accesses dominated by the weights

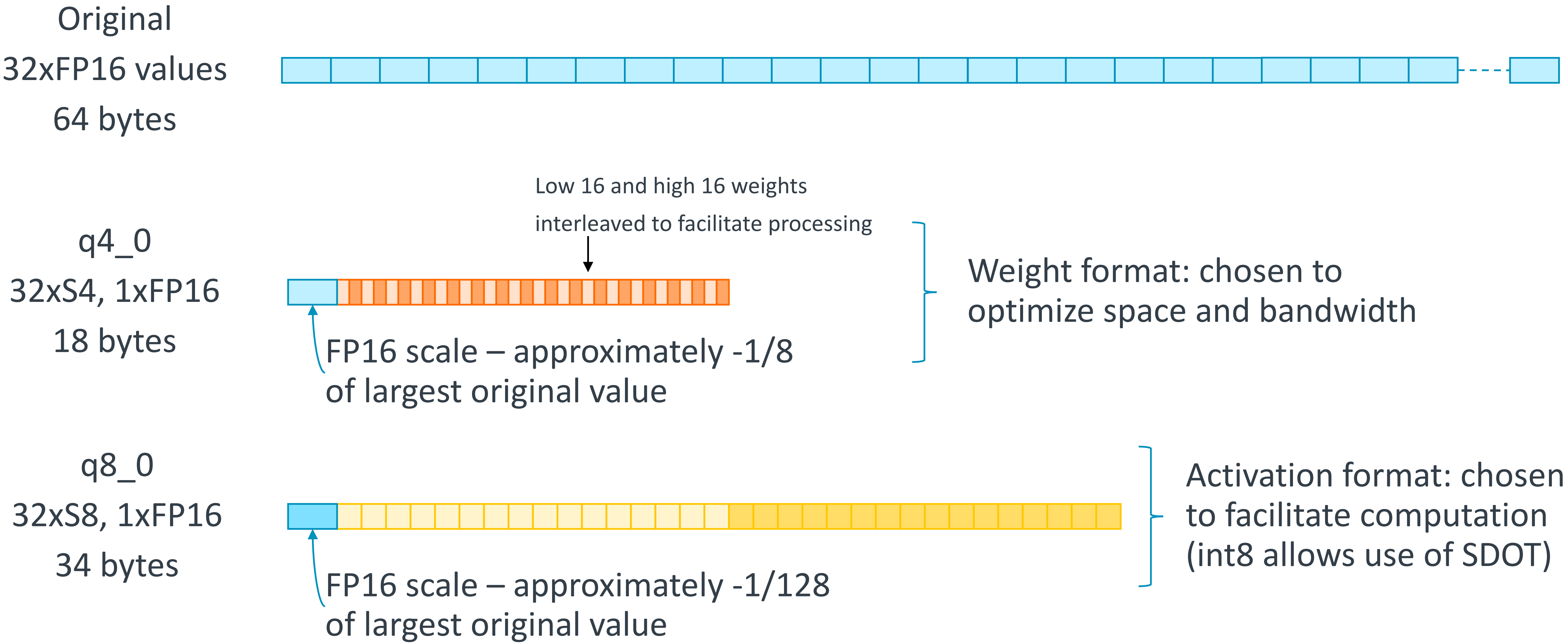
LLaMA inference – CPU runtime on Graviton3 - Neoverse V1

GEMM/GEMV background

- For typical operators in LLMs, weight matrix (B) is much larger than the input (A) and output (C).
- Compression of weight matrix is key to reducing memory and bandwidth consumption.
- llama.cpp/GGML use block quantized formats to store chunks of weight columns and activations
- In GGML, a dot-product kernel computes a single result – it's called at each point to populate the whole of C.



Block Quantized Formats



Original

32xFP16 values

64 bytes

q4_0

32xS4, 1xFP16

18 bytes

Low 16 and high 16 weights interleaved to facilitate processing

FP16 scale – approximately -1/8 of largest original value

Weight format: chosen to optimize space and bandwidth

q8_0

32xS8, 1xFP16

34 bytes

FP16 scale – approximately -1/128 of largest original value

Activation format: chosen to facilitate computation (int8 allows use of SDOT)

Block processing steps – (original GGML/llama.cpp)

Expand low weights to 8b (AND, SUB)

Expand high weights to 8b (SHR, SUB)

Initialize integer accumulator (MOV)

Multiply low part (DOT)

Multiply high part (DOT)

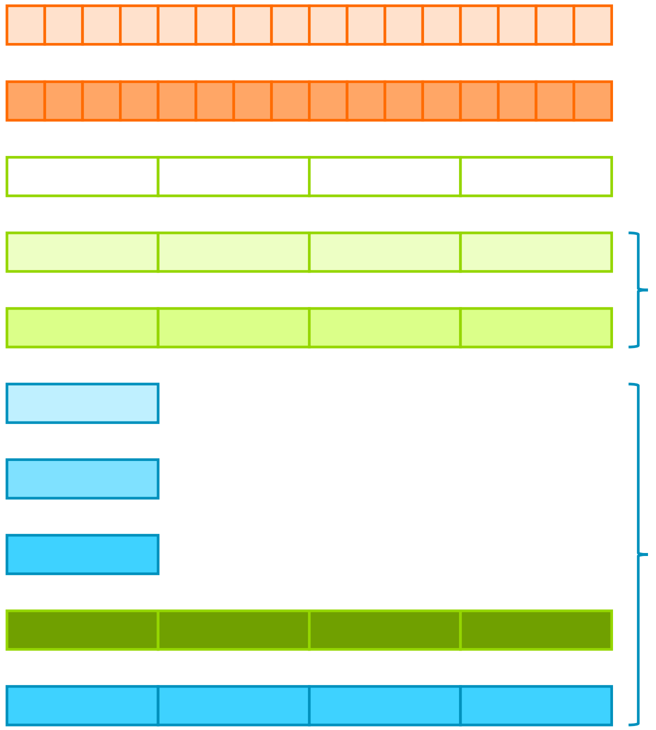
Convert LHS scale to FP32 (FCVT)

Convert RHS scale to FP32 (FCVT)

Combine scales (FMUL)

Convert integer sum to FP32 (SCVTF)

Scale + Accumulate (FMLA)



(-) No reuse of activations – redundant loads

(-) No reuse of activations scale

(-) No use of vector instructions for weights scales

(-) Pseudo-scalar ops

“real work”

scalar/pseudo-scalar ops

- 12 operations, of which 2 are doing the “real” MAC work (17%)
  - Plus 5 load ops (not shown) – comfortably compute bound at instruction level.
- 50% (6/12) are scalar/pseudo-scalar (work on a vector that is later reduced)

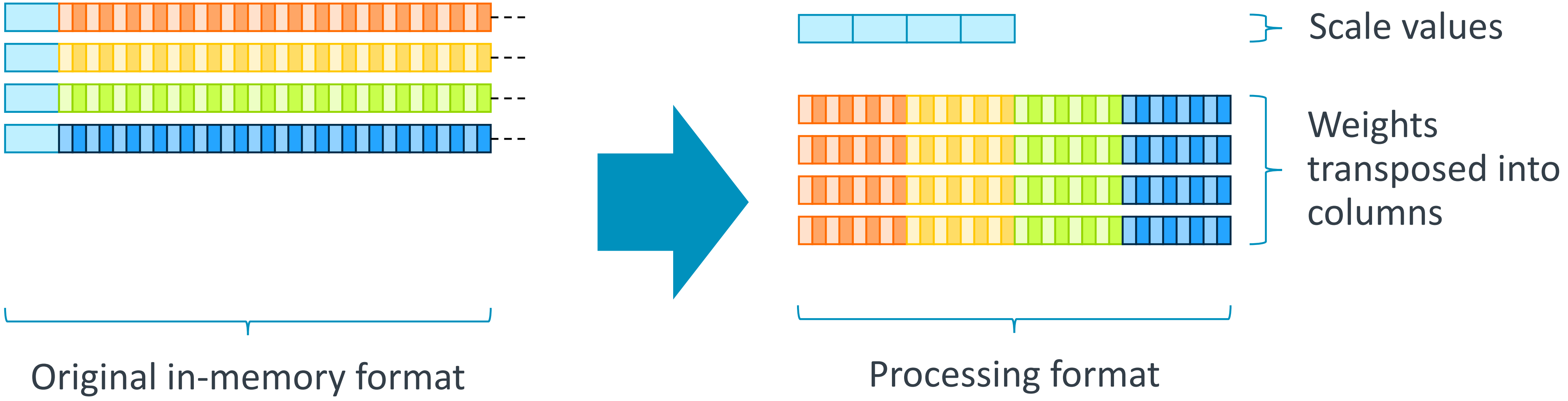
Avoiding pseudo-scalar operations

- Half the operations in original code are scalar or "pseudo-scalar" – operating on a vector of values which is really one true value split across lanes.
  - This technique reduces the number of reduction operations (sum across lanes) needed.
  - Still less efficient than ”true” vector operations.

- Using true vector operations should improve performance by around 60%.
  - Runtime of 50% (already vectorized) + (50%/4 = 12.5%) = 62.5%.
  - 62.5% runtime = 1.6x performance.

- => Vector lanes need to accumulate different results rather than multiple parts of the same result.
- => Compute more than one result at once – for non-batched case this must be different output points.

Transformed block layout



- To avoid pseudo-scalar ops, need to arrange than each lane is working on unique result.
- This means moving data into the relevant lane (transposing).
- Lane loads can assemble vector of scale values.

Block processing steps – 4 simultaneous blocks

Transpose weights into columns (8x ZIP)

Expand low weights to 8b (4x AND, SUB)

Expand high weights to 8b (4x SHR, SUB)

Initialize integer accumulator (MOV)

Multiply low parts (4x DOT)

Multiply high parts (4x DOT)

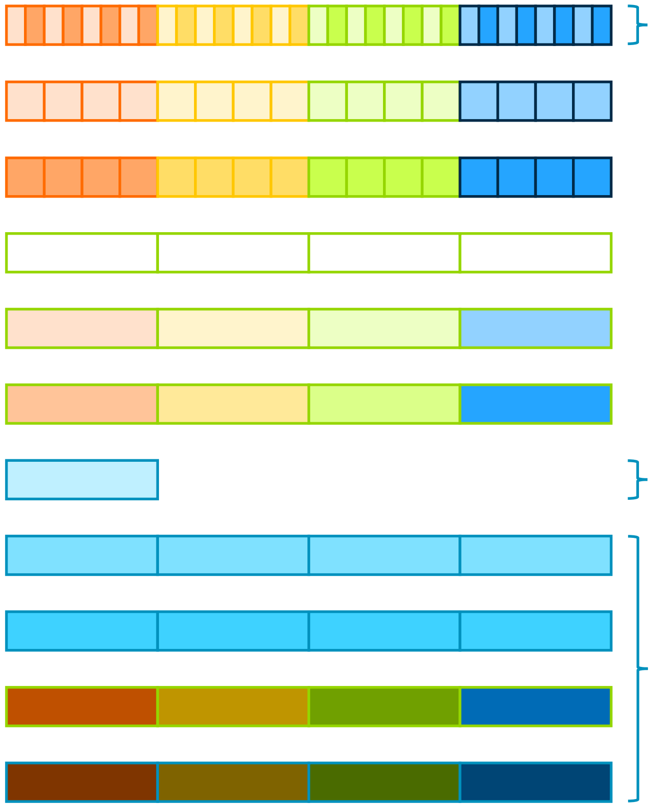
Convert LHS scale to FP32 (FCVT)

Convert RHS scales to FP32 (FCVT)

Combine scales (FMUL)

Convert integer sum to FP32 (SCVTF)

Scale + Accumulate (FMLA)



Extra operations added!

(+) Reuse of activations – No redundant loads

(+) Reuse of activations scale

(+) Use of vector instructions for weights scales

(+) No pseudo-scalar ops

Still scalar, but amortised

Now doing useful vector work

- 38 operations, computing 4 blocks => 9.5 operations per block (21% MAC)
- 26% speedup

Optimizing in-memory format

- Instead of permuting weights each time, store in memory in blocked format instead.
  - Space neutral – same data in a different order.
  - Improved alignment characteristics (no more 18-byte structures).
  - Scale factor handling easier (don't need to assemble vector from multiple locations)
  - Could go full "structure of arrays"; we just went for "array of more useful structures".

- Extra saving available on 4->8 bit unpacking:
  - Current scheme stores signed 4-bit values as unsigned (+8 bias) to avoid sign extension problems.
  - Need to subtract 8 to restore true signed value and sign bits.
  - Turns out it's more efficient to store signed values directly:
    - Top nibble can achieve sign extension with single signed shift op.
    - Bottom nibble can be recovered with 2 shifts, which should cost the same as AND and SUB.

Block processing steps – optimized memory format

~~Transpose weights into columns (8x ZIP)~~

Expand low weights to 8b (4x MUL, SHR)

Expand high weights to 8b (4x SHR~~, SUB)~~

Initialize integer accumulator (MOV)

Multiply low parts (4x DOT)

Multiply high parts (4x DOT)

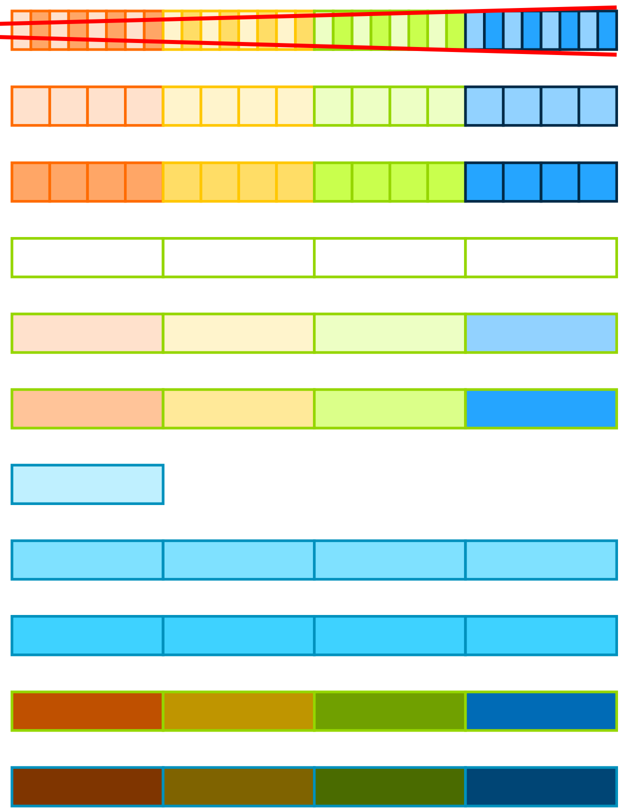
Convert LHS scale to FP32 (FCVT)

Convert RHS scales to FP32 (FCVT)

Combine scales (FMUL)

Convert integer sum to FP32 (SCVTF)

Scale + Accumulate (FMLA)



- 26 operations, computing 4 blocks => 6.5 operations per block (31% MAC)
- 85% speedup over original code

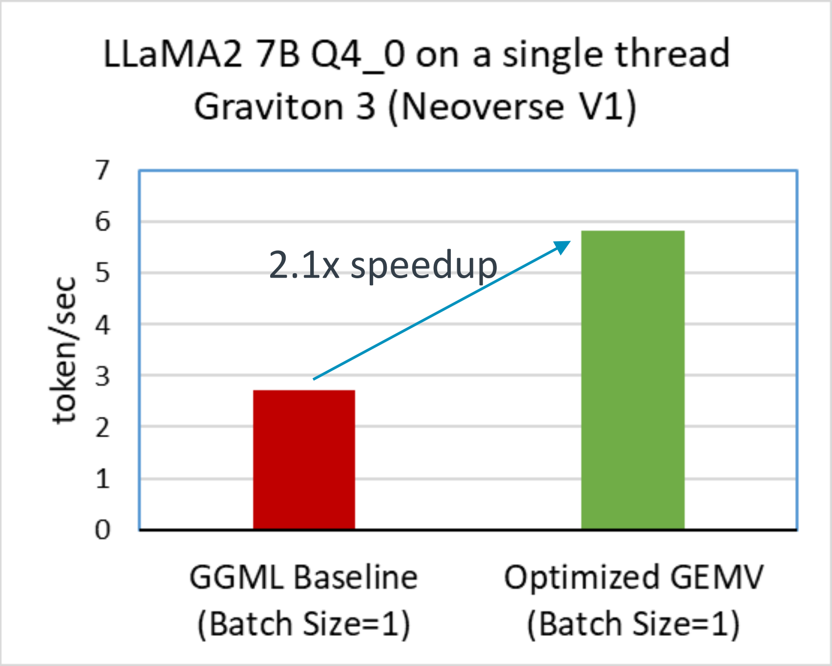
Non-batched inference on Graviton3 server CPUs (Neoverse cores)

Disclaimer

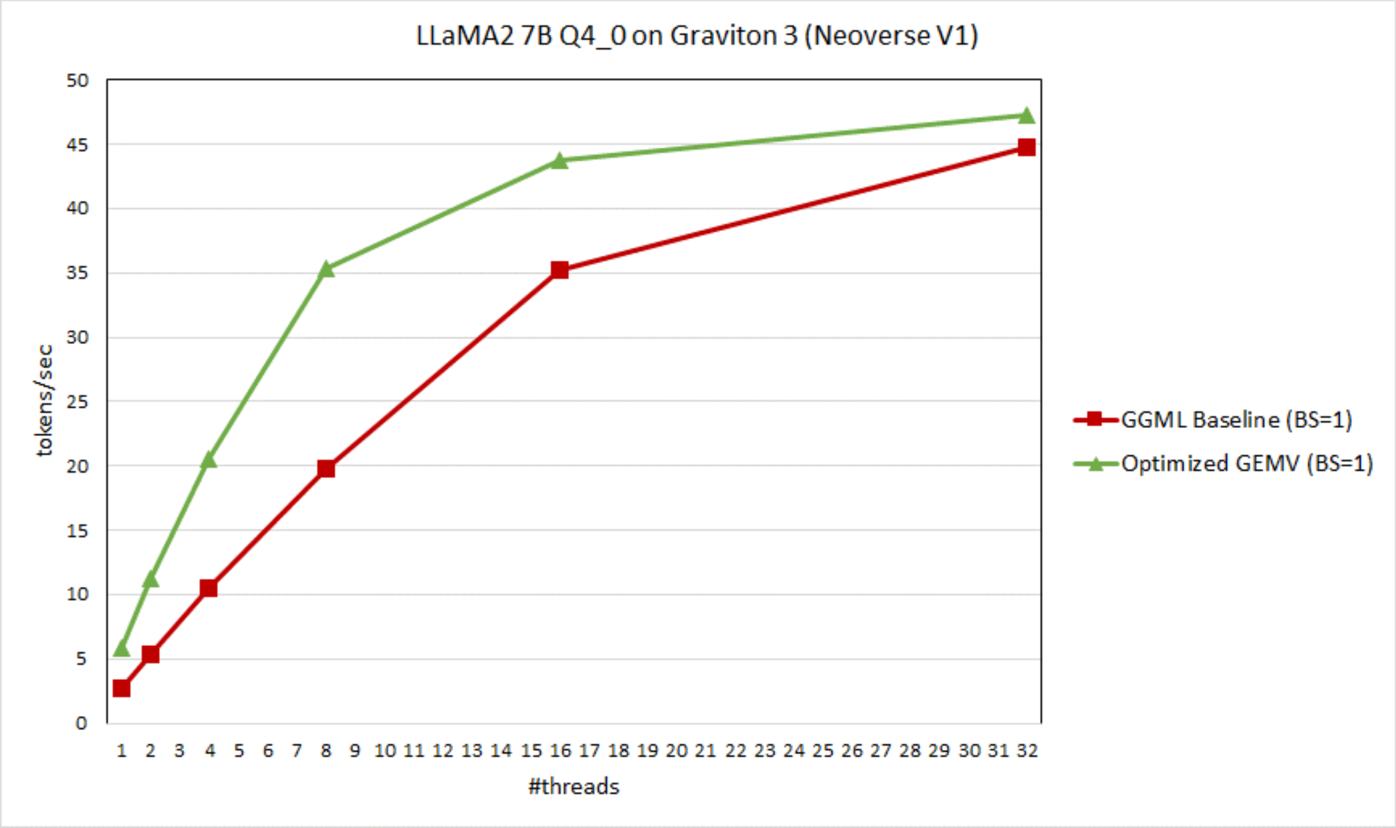
- This benchmark presentation made by Arm Ltd and its subsidiaries (Arm) contains forward-looking statements and information. The information contained herein is therefore provided by Arm on an "as-is" basis without warranty or liability of any kind. While Arm has made every attempt to ensure that the information contained in the benchmark presentation is accurate and reliable at the time of its publication, it cannot accept responsibility for any errors, omissions or inaccuracies or for the results obtained from the use of such information and should be used for guidance purposes only and is not intended to replace discussions with a duly appointed representative of Arm. Any results or comparisons shown are for general information purposes only and any particular data or analysis should not be interpreted as demonstrating a cause and effect relationship. Comparable performance on any performance indicator does not guarantee comparable performance on any other performance indicator.
- Any forward-looking statements involve known and unknown risks, uncertainties and other factors which may cause Arm’s stated results and performance to be materially different from any future results or performance expressed or implied by the forward-looking statements.
- Arm does not undertake any obligation to revise or update any forward-looking statements to reflect any event or circumstance that may arise after the date of this benchmark presentation and Arm reserves the right to revise our product offerings at any time for any reason without notice.
- Any third-party statements included in the presentation are not made by Arm, but instead by such third parties themselves and Arm does not have any responsibility in connection therewith.

LLaMA2 7B Q4_0 for single inference case on Graviton3 (Neoverse V1)

Single thread behavior



Thread scaling behavior

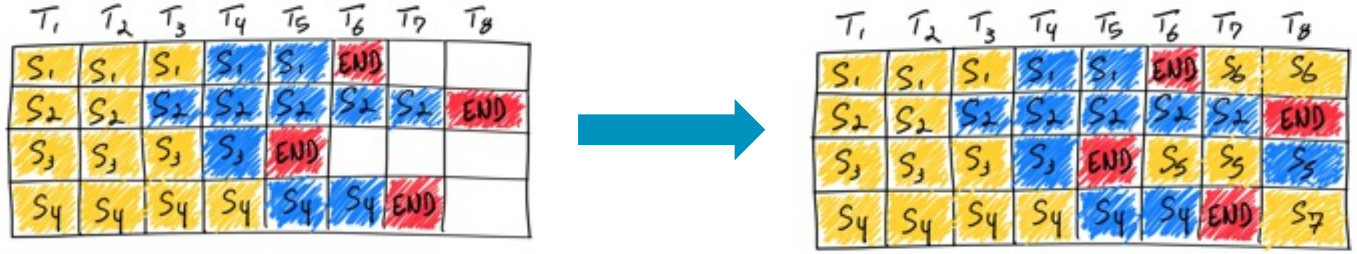


Batched inference

Batched inference of LLMs

- Typically seen in server setting

- Larger batch of inferences possible when serving multiple requests from different users
  - Difference in term of sequence lengths can be handled with techniques like dynamic batching



T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1	S_1	END		
S_2	S_2	S_2	S_2	S_2	S_2	S_2	END
S_3	S_3	S_3	S_3	END			
S_4	S_4	S_4	S_4	S_4	S_4	END	

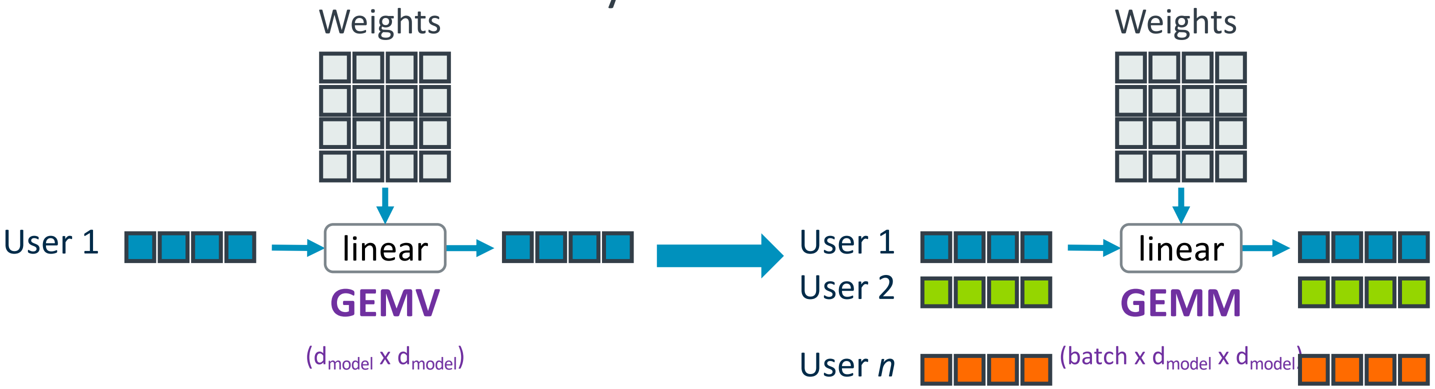
T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1	S_1	END	S_6	S_6
S_2	S_2	S_2	S_2	S_2	S_2	S_2	END
S_3	S_3	S_3	S_3	END	S_5	S_5	S_5
S_4	S_4	S_4	S_4	S_4	S_4	END	S_7

- Limited batching of inference for a single user enabled by speculative decoding

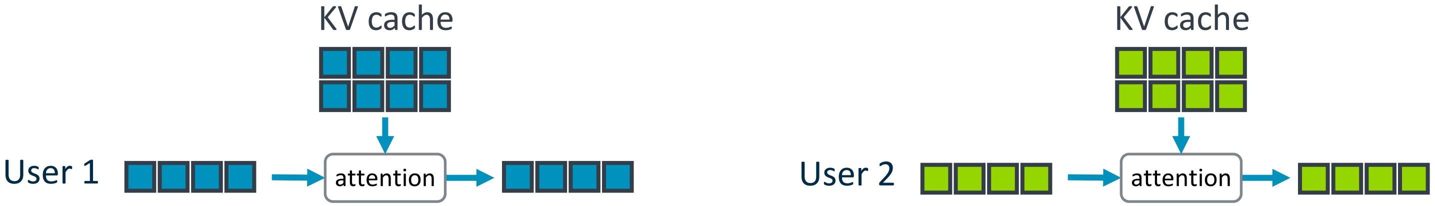
- A throughput optimisation problem within constraints of latencies and limited resources
  - Memory footprint is a challenging constraint for large models (LLAMA2 up to 65 billion parameters, PALM up to 540 billions)

Batching of operators

- Linear projections and the Feed Forward Network can be easily batched
  - GEMV operations will become GEMM operations
  - Size of matrices and arithmetic intensity will increase with batch size


Weights
User 1
linear
GEMV
(d_{model} x d_{model})
Weights
User 1
User 2
User *n*
linear
GEMM
(batch x d_{model} x d_{model})

- But attention mechanism cannot be batched
  - Even if multi-query attention & grouped-query attention can be processed with GEMM operations



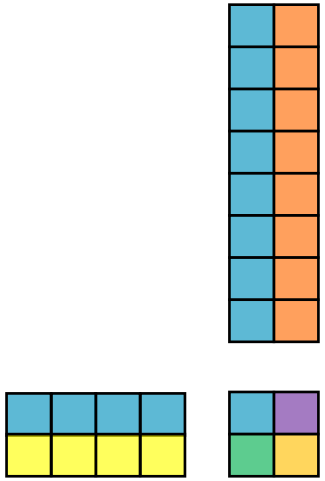
Optimized GEMM for batched inference

- Uses the same concepts as GEMV:
  - Weights in blocks prearranged ready for processing.
  - Apply the same to activations (as we now process multiple rows of activations).

- Uses SMMLA instruction (double the MAC count of SDOT, requires multiple output rows).

- Reduced bandwidth consumption as each input is processed by several SMMLA instructions.

- Batch 8: 79 vector ops, 32 output points => 2.5 ops/point (40% MAC)
  - 4.8x speedup vs original GGML code, 2.6x speedup vs optimized GEMV code



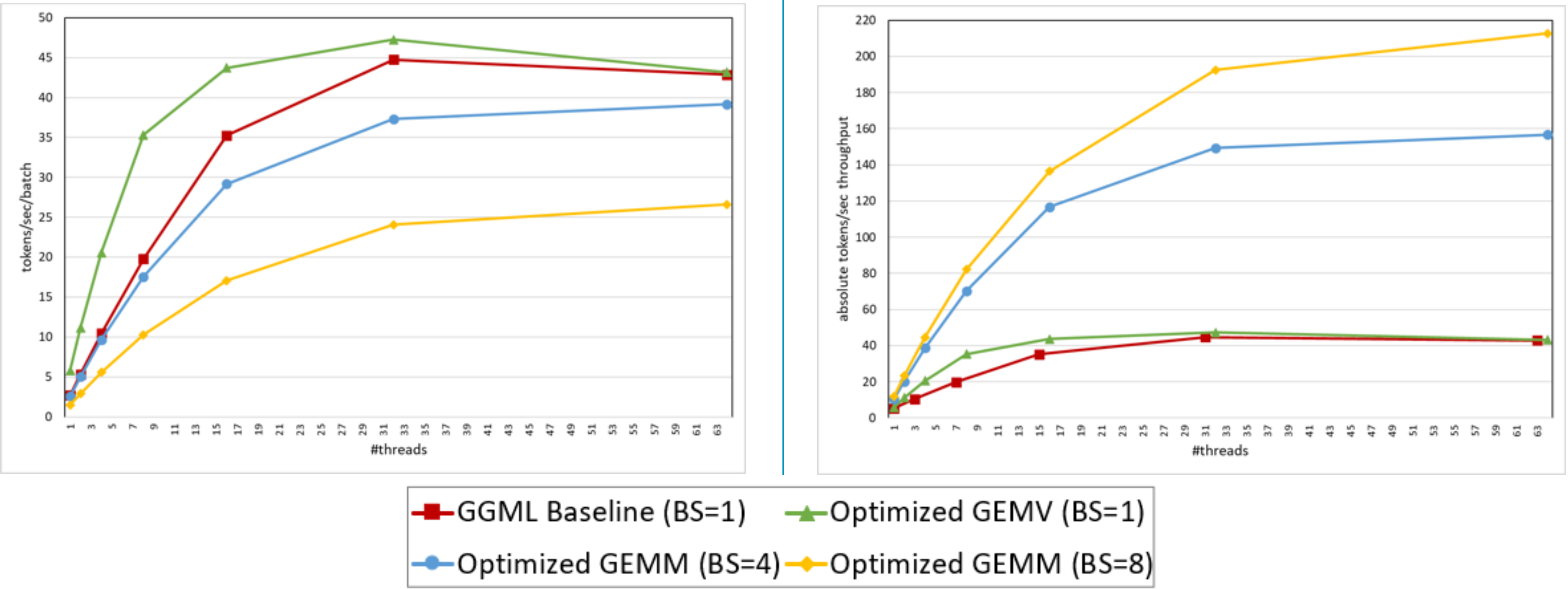
SMMLA

Batched inference on Graviton3 server CPUs (Neoverse cores)

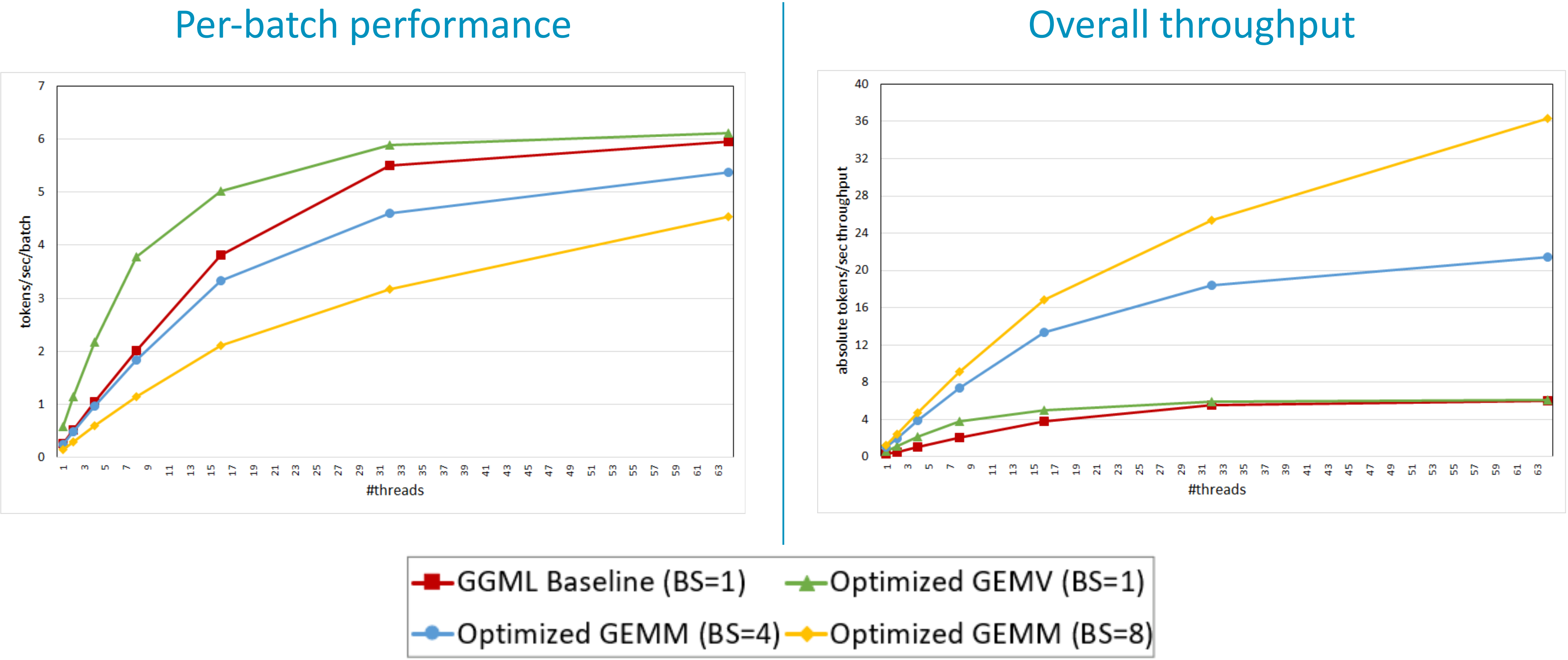
LLaMA2 7B Q4_0 for batched inference case on Graviton3 (Neoverse V1)

Per-batch performance

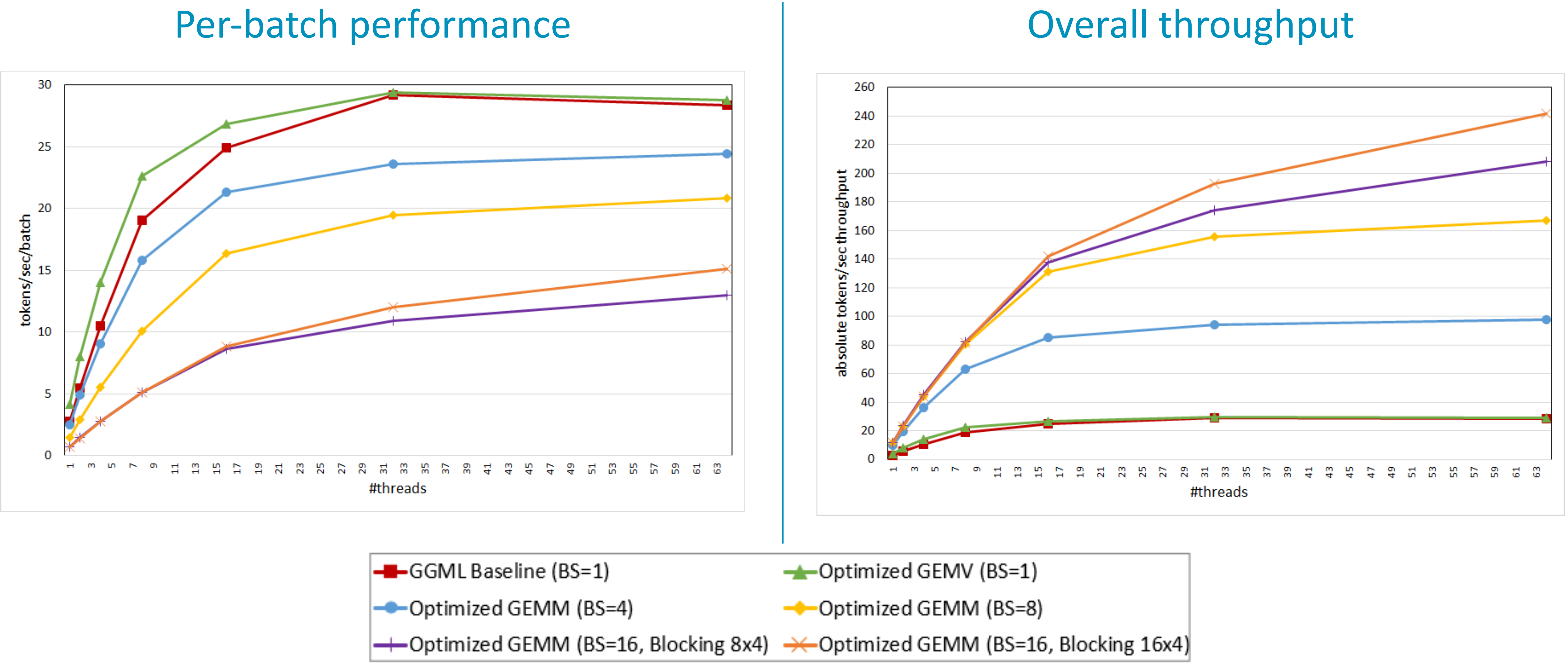
Overall throughput



LLaMA2 70B Q4_0 for batched inference case on Graviton3 (Neoverse V1)



LLaMA2 7B Q8_0 for batched inference case on Graviton3 (Neoverse V1)



Per-batch performance

Overall throughput

Limitations and future work

- Q4_0 is the simplest blocked quantization format.
    - Results are lower quality than other, more complex, schemes; performance is much higher.
    - Need to understand more about this tradeoff – is q4_0 good enough for some use cases?

- Future work to understand and optimize more of the llama.cpp schemes (e.g. Q4_K).

Conclusions

- CPUs are a viable platform for LLM inference.

- Llama.cpp has reasonable implementations for Arm CPUs but further optimization is possible.
  - 2.1x speedup per thread on non-batched case.

- High scaling of overall throughput with batch size shows promise for future platforms with more lpddr bandwidth

Thank You

Danke

Gracias

Grazie

谢谢

ありがとう

Asante

Merci

감사합니다

धन्यवाद

Kiitos

شكرًا

ধন্যবাদ

תודה