arm

Accelerated LLM inference on Arm CPU

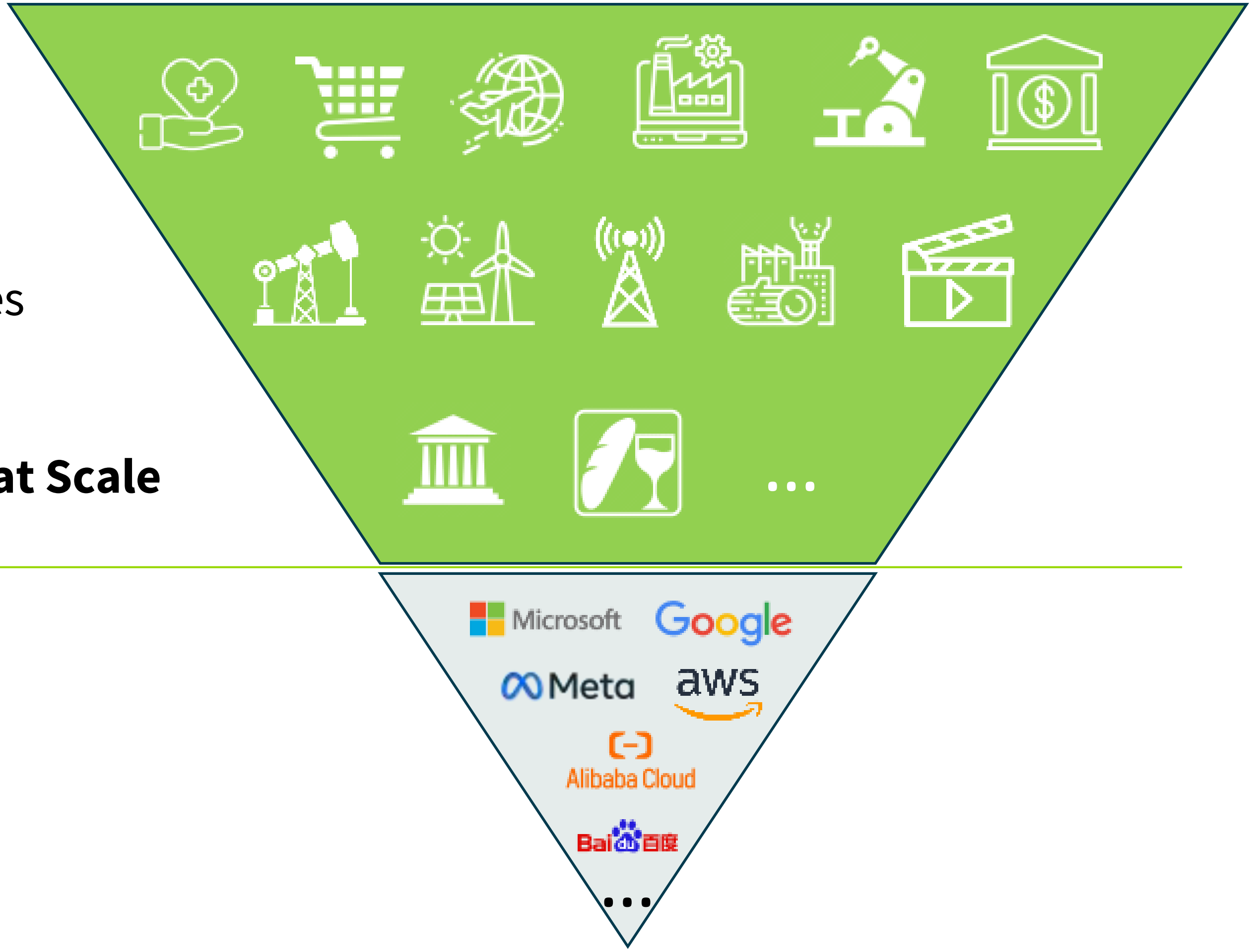
AArch64 Optimized GEMV and GEMM kernels for Llama.cpp Q4_0 Quantization

Tianyu Li
Jun 2024

Generative AI Adoption

Inferencing

- **~80-85%** of AI workloads
- Customize to industry verticals & enterprises
- Hundreds of startups launched since 2023
- An evolving AI software stack
- **Needs to be done at low TCO (Perf/Watt) at Scale**



Training Frontier Models

- ~20-15% of AI workloads
- Led by handful of hyperscalers
- Will remain cost and power intense for now

Rise of Smaller Specialized LLMs

‘Democratizes’ LLMs by bringing them to a wider set of developers

“Small” LLMs – 2-70B parameters

LLaMa3 Qwen GLM
Baichuan Phi3

.. and 1000s of others on e.g. Hugging-Face

- Typically open-source
- Efficient at focused tasks, data-sets
- Can be easily fine-tuned, augmented
- Runs on wide variety of platforms – CPUs & GPUs
- Lower security risk, better privacy

“Large” LLMs – 180B – 1T+ parameters

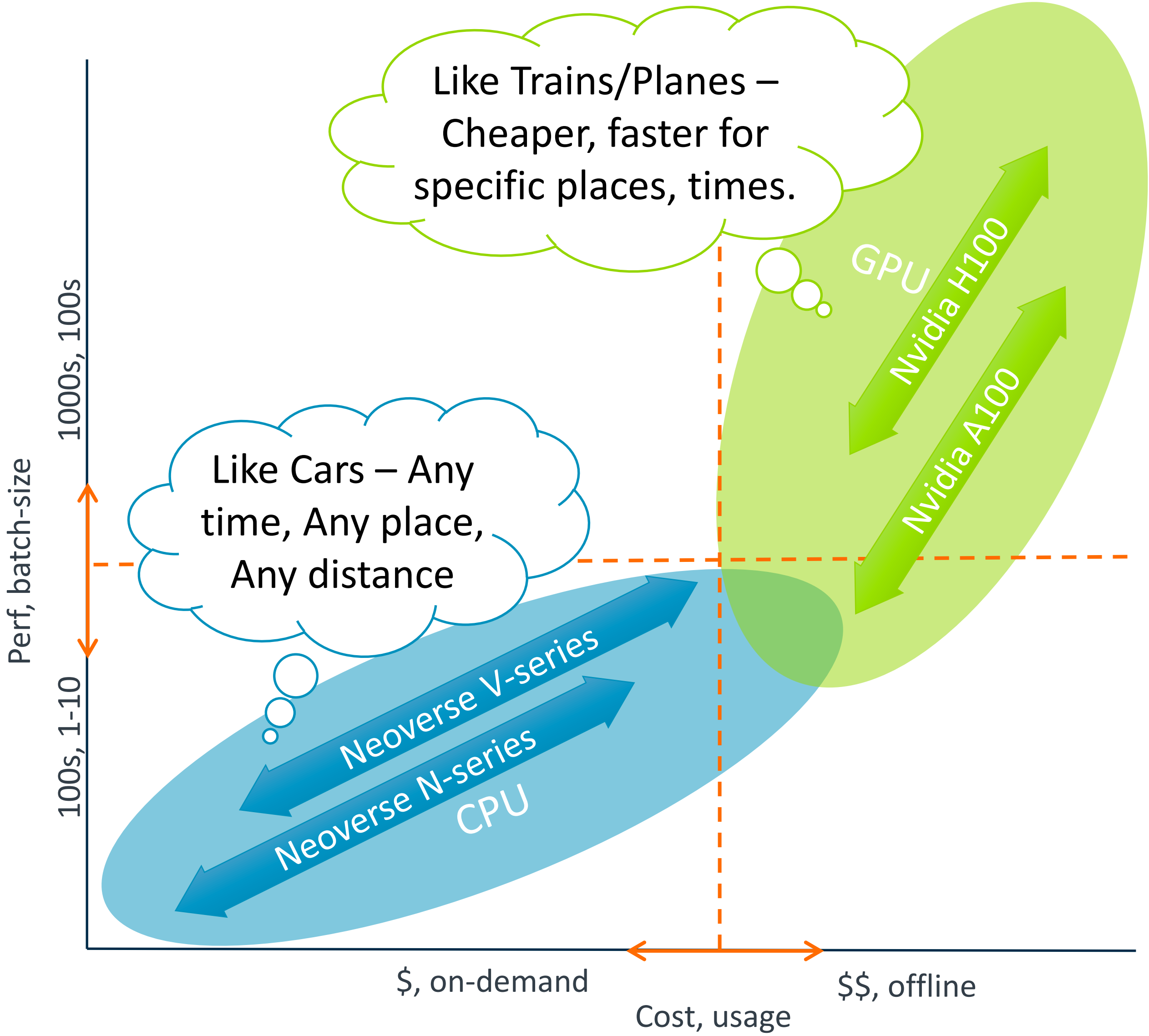
GPT4 Claude
ERNIE

… just a handful

- Typically closed-source
- Efficient at variety of generic tasks
- Limited fine-tuning, augmentation
- Requires large cluster of GPUs/accelerators
- Privacy, security can be a challenge

CPUs vs. GPUs – Cars vs. Trains/Planes

- GPUs – compute-heavy and expensive
  - Require a certain threshold of ML inferencing to justify cost, integration.
  - Incur an accessibility tax – added latency.
- CPUs – compute-decent and scalable
  - Can scale with ML inferencing needs.
  - Great for on-demand inferencing.
- Adoption trends
  - Large-scale ML users – start with GPUs for offline, mix-match with CPUs for on-demand operations.
  - Entry-level ML users - start with CPUs, and then decide if scale justifies GPUs.



* Above chart is based on LLaMa2-7B performance – other models have similar characteristics, but different crossover points.

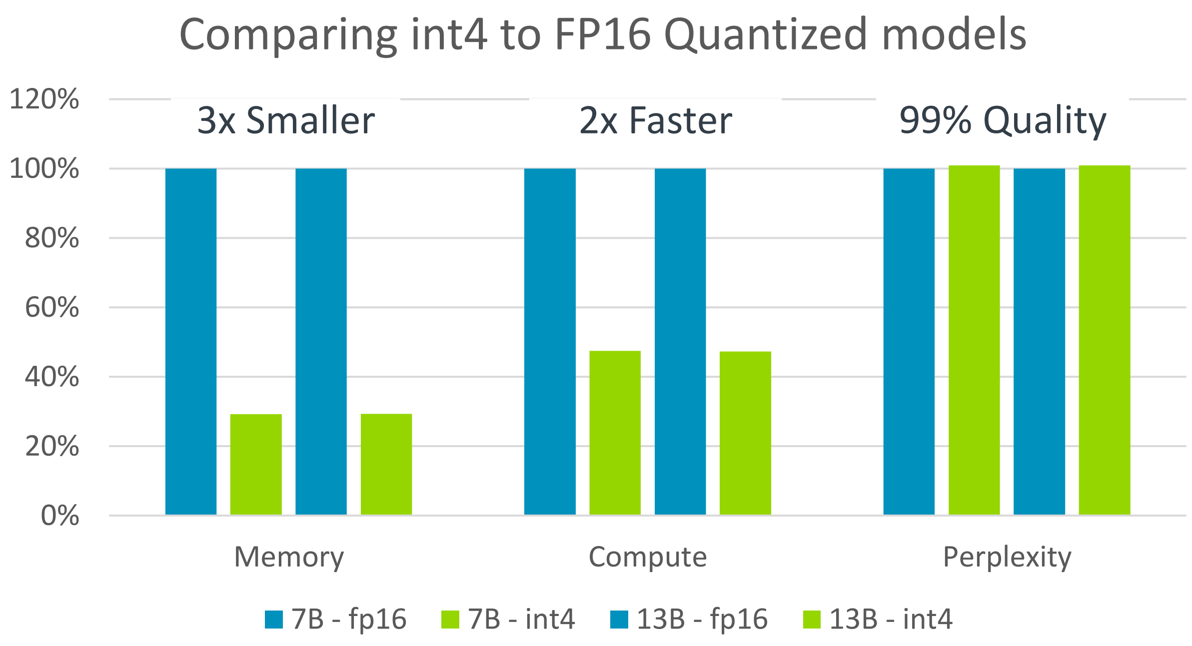
Optimization Techniques - Quantization

- Higher Precision formats (FP32) preferred for Training.
- Lower Precision formats (FP16, FP8, INT8, INT4) preferred for Inferencing

- Lower Precision formats like INT4 significantly decrease memory + compute footprint.
    - Comes with a minor (~ 1%) trade-off in quality of output.

- Different types of Quantization techniques
    - Post-training Quantization
    - Quantization aware training

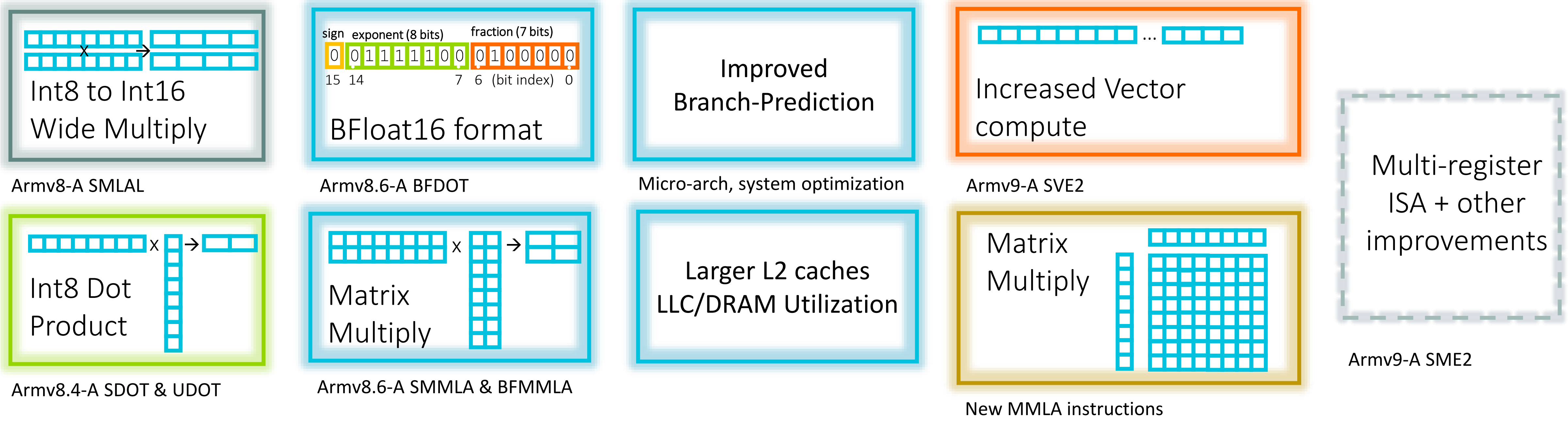
- Different Quantization formats
    - GGUF – CPU and GPU support
    - GPTQ/NF4 – focused on GPUs



* Using LLaMa2 with llama.cpp perplexity benchmark

On-CPU ML architecture evolution

Arm has made several architectural improvements to improve on-CPU ML performance.
These improvements span across Neoverse (Infrastructure) and Cortex-A (Client/IoT/Auto) families.



Neoverse N1
AWS Graviton2
Ampere Altra

Neoverse V1, N2
AWS Graviton3
Alibaba Yitian
Microsoft Cobalt

Neoverse V2, V3, N3
AWS Graviton 4
Google Axion
Nvidia Grace

Upcoming Neoverse cores

ML Software Stack on Arm Neoverse

Open Source | Accelerator Vendor | nVidia | Arm

Models

Large Language Models | Generative AI | Vision | NLP | Recommender | ...

Frameworks / Runtimes

TensorRT | PyTorch | TensorFlow | ONNX | PaddlePaddle | llama.cpp, gemma.cpp

Libraries / Backends

CUDA | Custom

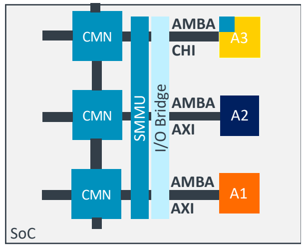
Arm Compute Library (ACL) (via oneDNN in some cases) | XLA/MLIR | KleidiAI

Hardware

Neoverse Host

+

Accelerated Compute



Neoverse On-CPU ML

NEON | SVE

SDOT | MMLA | INT8, BF16

Llama.cpp AArch64 Optimizated GEMV and GEMM kernels for Q4_0 quantization

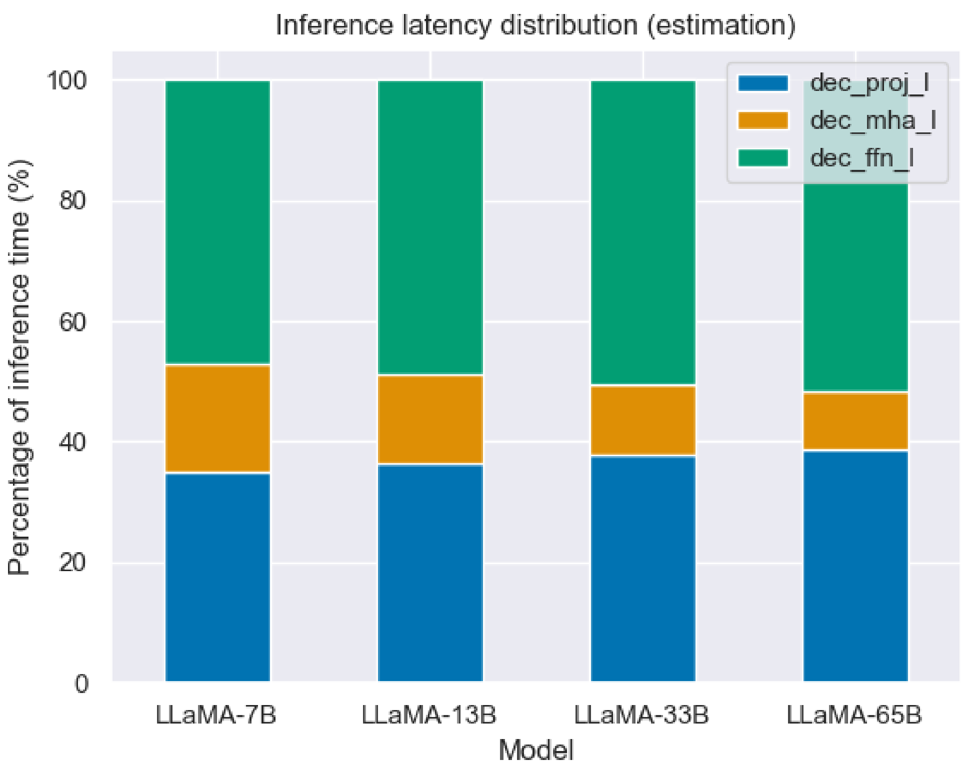
Llama.cpp Introduction

- Inference of Llama2 and other LLMs in C/C++
- Enable LLM inference with minimal setup
- State-of-the-art performance
- Wide support across hardware/OS/models
- 1.5 - 8 bits quantization support for faster inference and reduced memory use

```
$ make -j && ./main -m models/llama-13b-v2/ggml-model-q4_0.gguf -p "Building a website can be
```

```
 Building a website can be done in 10 simple steps:
Step 1: Find the right website platform.
Step 2: Choose your domain name and hosting plan.
Step 3: Design your website layout.
Step 4: Write your website content and add images.
Step 5: Install security features to protect your site from hackers or spammers
Step 6: Test your website on multiple browsers, mobile devices, operating systems etc…
Step 7: Test it again with people who are not related to you personally – friends or family me
Step 8: Start marketing and promoting the website via social media channels or paid ads
Step 9: Analyze how many visitors have come to your site so far, what type of people visit mor
Step 10: Continue to improve upon all aspects mentioned above by following trends in web desig
How does a Website Work?
A website works by having pages, which are made of HTML code. This code tells your computer ho
The most common type is called static HTML pages because they remain unchanged over time unles
How to
llama_print_timings:        load time =   576.45 ms
llama_print_timings:      sample time =   283.10 ms /   400 runs   (    0.71 ms per token,  14
llama_print_timings: prompt eval time =   599.83 ms /    19 tokens (   31.57 ms per token,
llama_print_timings:        eval time = 24513.59 ms /   399 runs   (   61.44 ms per token,
llama_print_timings:       total time = 25431.49 ms
```

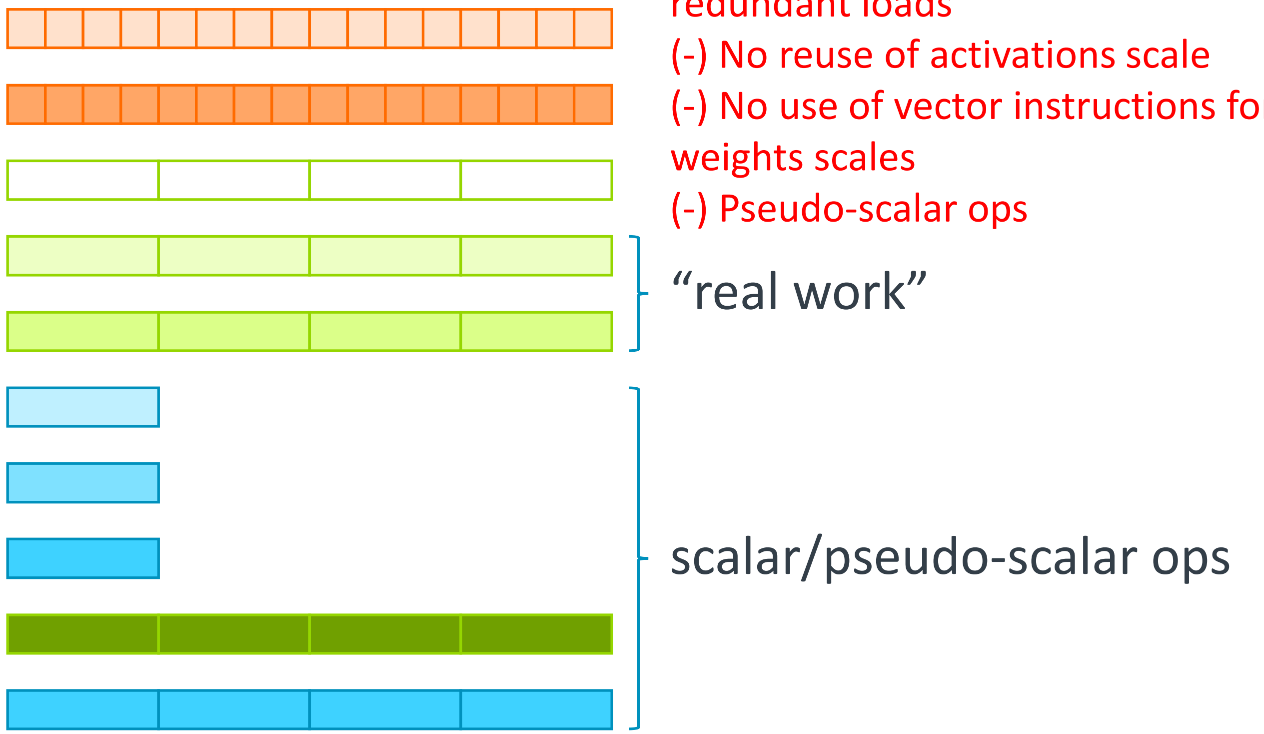
Inference of LLMs – most cost operation GEMM/GEMV



- 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

Matmul 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

- 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)

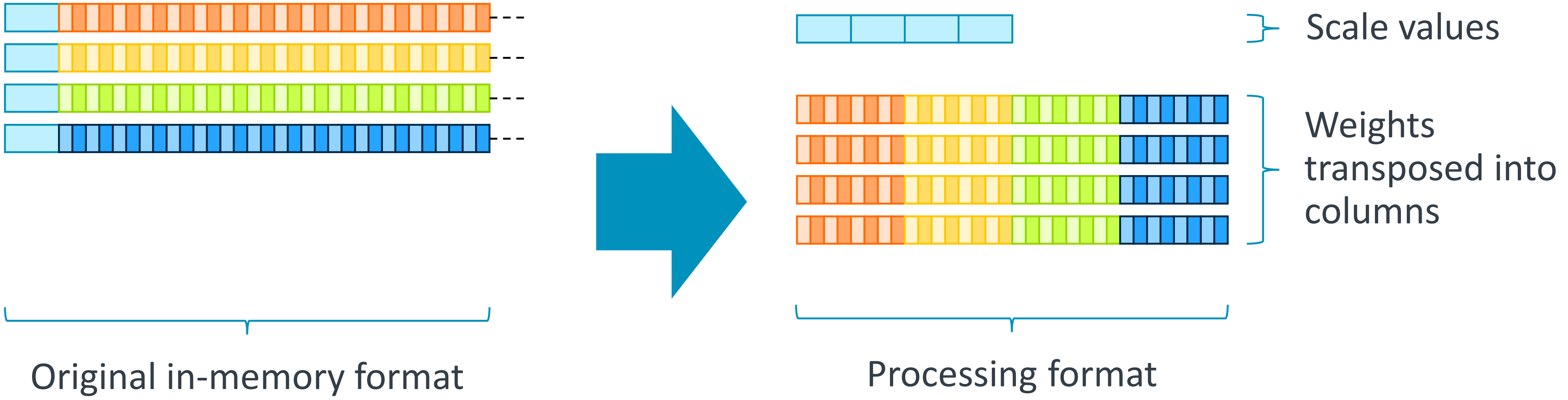
Avoiding pseudo-scalar operations

- 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 improve performance by around 60%.

- => 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.

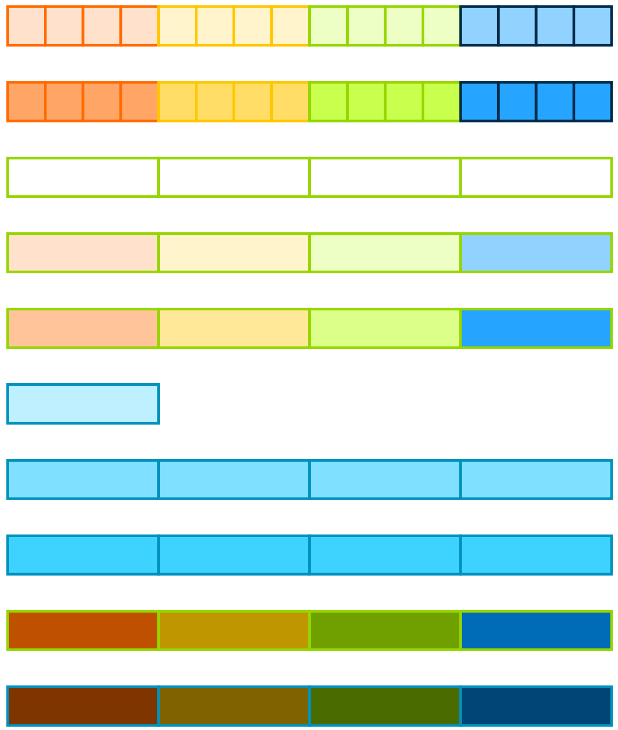
Optimizing in-memory format

- 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.

Matmul processing steps – optimized memory format

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)



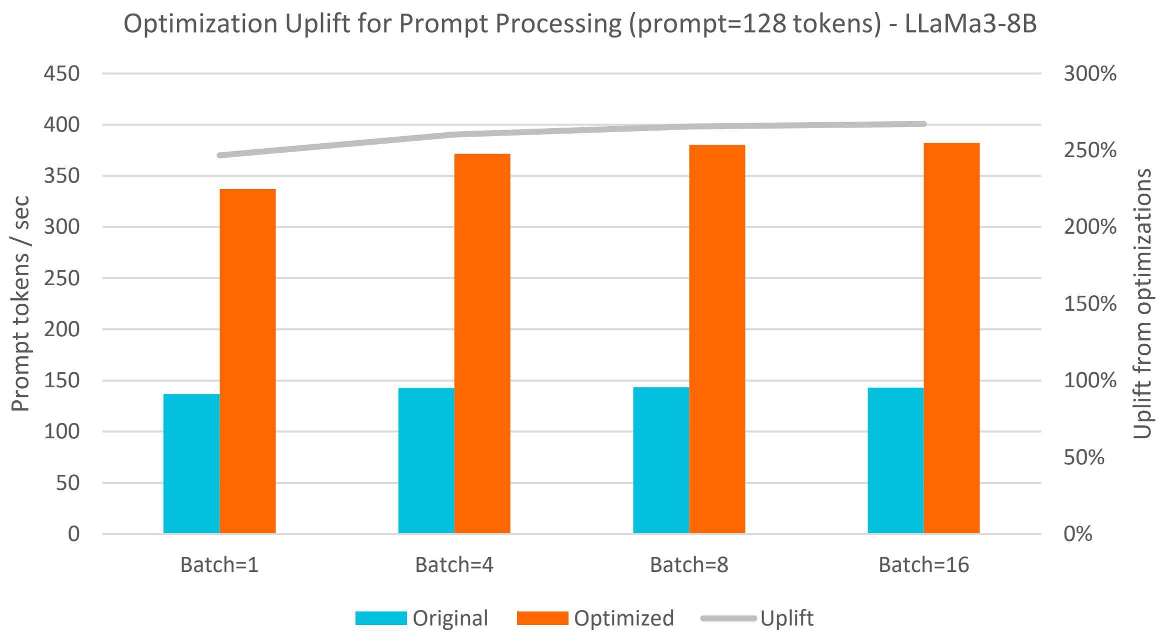
(+) Reuse of activations – No redundant loads
(+) Reuse of activations scale
(+) Use of vector instructions for weights scales
(+) No pseudo-scalar ops

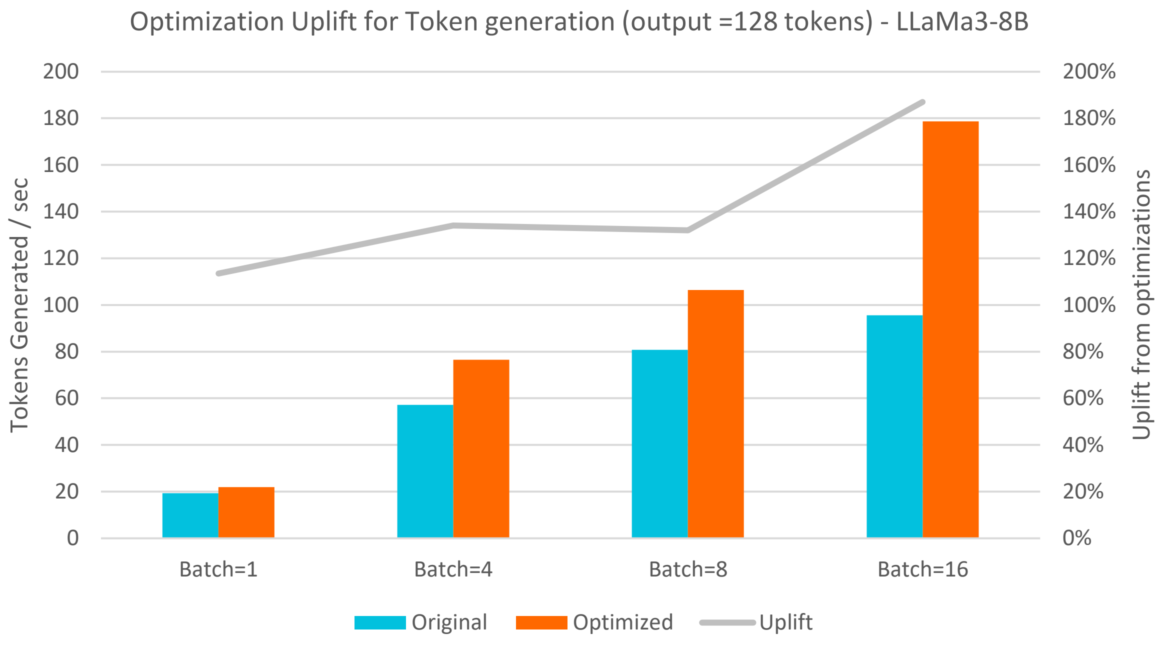
Still scalar, but amortised

Now doing useful vector work

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

Benefits from Software optimizations





Optimized MMLA implementation provides up to 3x improvement for LLMs

- > 2.7x faster time-to-first-token
- ~ 1.9x higher token-gen throughput with < 100 msec time-to-next-token latency

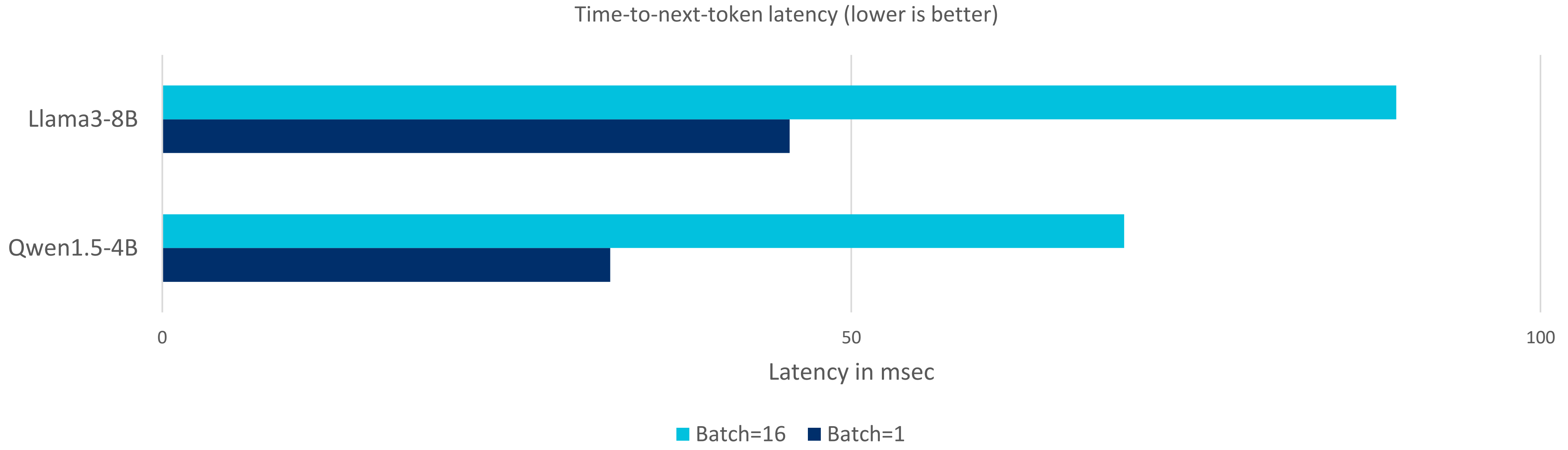
Github PR

Significant uplifts on Neoverse N1 as well.

Using LLaMa-3 8B model with int4 quantization on AliCloud Yitian710 16xlarge instances

Leverages MMLA optimizations for GEMM and GEMV in llama.cpp framework

Meets key performance criteria for LLMs



- 100ms time-to-next-token key requirement for LLM deployments
- Neoverse N2 easily meet this for Small Language models.
- Balance of both latency and throughput (for higher batch-size).

Using LLaMa-3 and Qwen1.5 models with int4 quantization on AliCloud 16xlarge instances

Leverages MMLA optimizations for GEMM and GEMV in llama.cpp framework

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు