Optimization of Armv9 architecture general large language model inference performance based on Llama.cpp

Take Qwen and Yitian 710 as examples

Longhao Chen*, Yina Zhao†, Qiangjun Xie‡, Qinghua Sheng‡
*CEATRG, Hangzhou Dianzi University, Hangzhou, CN
Email: Longhao.Chen@outlook.com
†CEATRG, Wuhan University, Wuhan, CN
Email: 2022302141057@whu.edu.cn
‡Hangzhou Dianzi University, Hangzhou, CN
Email: qjunxie@163.com
‡Hangzhou Dianzi University, Hangzhou, CN
Email: sheng7@hdu.edu.cn

Abstract—This article optimizes the inference performance of the Qwen-1.8B [6] model by performing Int8 quantization, vectorizing some operators in llama.cpp [9], and modifying the compilation script to improve the compiler optimization level. On the Yitian 710 experimental platform, the prefill performance is increased by 1.6 times, the decoding performance is increased by 24 times, the memory usage is reduced to 1/5 of the original, and the accuracy loss is almost negligible.

Index Terms—LLM, inference, ARM, llama.cpp, Optimization

I. Introduction

Large Language Models (LLMs) have achieved remarkable performance in most natural language processing downstream tasks, such as text understanding, text generation, machine translation, and interactive Q&A. However, the billions or even trillions of model parameters pose significant challenges for efficient deployment of LLMs at the edge. With the growth rate of model parameters far outpacing the improvement in hardware performance, the academic and industrial communities are exploring software and hardware collaborative methods like model compression, dataflow optimization and operator invocation to deploy and run large models under the limited hardware conditions.

This article uses the default quantizer of llama.cpp to perform Int8 quantization on the Qwen-1.8B model, uses ARM's NEON instructions to vectorize some operators in llama.cpp, and modifies the compilation script to improve the GCC compiler optimization level. In the test, the prefill performance increased from 86 token/s to 145 token/s, the decode performance increased from 2 token/s to 48 token/s, the memory usage was reduced from 10GiB to 2.3GiB, and tested using the piqa [7] data set in lm-evaluation-harne [8], the accuracy is only reduced by 0.0076. All codes in this article are open source to https://github.com/Longhao-Chen/Aicas2024

II. Experiment platform

A. Yitian 710 processor

Yitian 710 is the first Arm server chip released by T-Head. It is independently designed and developed by T-Head. It adopts an advanced architecture and has the characteristics of high energy efficiency, high bandwidth and is compatible with the Armv9 architecture.

In the Armv9 architecture, VDOT, MMLA and other instructions for Int8 calculations are provided, which can greatly accelerate Int8 type model inference.

B. Qwen

Qwen is a LLM officially released by Alibaba Cloud Computing Co. Ltd., with parameter scales ranging from billions to trillions. The comprehensive performance of this model is well-rounded in mainstream benchmark evaluations.

In this article, we use Qwen 1.8B with 24 decode layers as an experimental model.

III. Optimization

A. Use the latest compiler

Old compilers do not support Integer Matrix Multiply intrinsics, specifically __ARM_FEATURE_MATMUL_INT8 is not defined [2]. To be able to use Integer Matrix Multiply intrinsics we need to use a newer compiler. We are using GCC 13.2.0. In this version, Integer Matrix Multiply intrinsics are fully supported.

B. Use more compiler optimizations

Compiler optimization is an effective and convenient optimization method, and the GCC compiler supports multiple optimization levels. By default, llama.cpp uses -O3 optimization. To use higher-level optimization methods, you can use the command LLAMA_FAST=1 make -j8 to compile. This will enable a higher level of -Ofast optimization. Compared with -O3, -Ofast will enable -fallow-store-data-races, -fassociative-math, -fcx-limited-range, -fexcess-precision=fast, -ffinite-math-only, -freciprocal-math, -funsafe-math-optimizations, and disable -fsemantic-interposition, -fsigned-zeros, -fmath-errno, -ftrapping-math

Link Time Optimization (LTO) gives GCC the capability of dumping its internal representation (GIMPLE) to disk, so that all the different compilation units that make up a single executable can be optimized as a single module. This expands the scope of inter-procedural optimizations to encompass the whole program (or, rather, everything that is visible at link time) [4]. Therefore, for programs composed of multiple source files, enabling LTO can achieve better optimization. In GCC, we can enable LTO by passing -flto.

C. Select the architecture of your host system

Modern cloud computing facilities usually use virtual machines or containers to isolate processes of different users. In these virtual machines or containers, programs often cannot obtain information about the underlying hardware. Therefore, at compile time, the host's architecture needs to be explicitly specified to the compiler. For Yitian 710 processor, you can use -mcpu=neoverse-n2 -mtune=neoverse-n2 [1].

D. Rewrite some operators using NEON

Arm Neon technology is an advanced Single Instruction Multiple Data (SIMD) architecture extension for the A-profile and R-profile processors. Neon technology is a packed SIMD architecture. Neon registers are considered as vectors of elements of the same data type, with Neon instructions operating on multiple elements simultaneously. Multiple data types are supported by the technology, including floating-point and integer operations [5]. For simple functions, the compiler's Auto-vectorization can already generate high-performance NEON instructions, but complex functions require manual writing of NEON instructions. We rewrote functions ggml_fp16_to_fp32_row, ggml_fp32_to_fp16_row, ggml_compute_forward_norm_f32, ggml_compute_forward_rms_norm_f32, ggml_compute_fp16_to_fp32.

E. Reduce unnecessary type conversions

The 8-bit quantization of llama.cpp treats 32 data as a group by default, and each group uses the same scaling factor. This scaling factor is saved as a float16 type [3]. Limited by processor instruction set, in each calculation, the corresponding scaling coefficient needs to be converted into data types, which will take up part of the time. If you directly use the float32 type to save the scaling factor, no conversion is required, which will save more time. But the memory usage will increase 5.88% .

IV. Evaluate

We use the speed of the inference interface in the Python package transformers 4.38.2 as a baseline.

The llama.cpp item in the table is the unmodified original program.

TABLE I
Precision

Test items	Accuracy(piqa)
Baseline	0.7312
llama.cpp + fp16[a]	0.7252
llama.cpp + Int8[a]	0.7236
llama.cpp + fp16[b]	0.7252
llama.cpp + Int8[b]	0.7236
Ours	0.7236

[a]Compiler: gcc 9.4.0
[b]Compiler: gcc 13.2.0

TABLE II
Memory usage

Test items	Physical memory (MiB)	Virtual memory (MiB)
Baseline	10627	12228
llama.cpp + fp16[a]	3807	4204
llama.cpp + Int8[a]	2165	2562
llama.cpp + fp16[b]	3807	4205
llama.cpp + Int8[b]	2166	2563
Ours	2250	2649

[a]Compiler: gcc 9.4.0
[b]Compiler: gcc 13.2.0

TABLE III
Inference rate

Test items	Prefill rate (tokens/s)	Decode rate (tokens/s)
Baseline	86.79	2.07
llama.cpp + fp16[a]	113.63	24.05
llama.cpp + Int8[a]	38.53	24.36
llama.cpp + fp16[b]	116.85	23.51
llama.cpp + Int8[b]	98.55	37.88
Ours	145.86	48.36

[a]Compiler: gcc 9.4.0
[b]Compiler: gcc 13.2.0

V. Discussion

Through experimental data, it can be seen that our solution greatly improves the inference performance of large language models with a low accuracy drop. Next, we can consider using the float type to save the scaling factor during quantization, which may result in smaller accuracy loss.

Acknowledgment

I would like to thank my parents and school for their support, and I would also like to thank the conference organizers for their support.

References

[1] AArch64 Options (Using the GNU Compiler Collection (GCC)) — gcc.gnu.org. https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/AArch64-Options.html#index-mtune. [Accessed 14-04-2024].

[2] Arm C Language Extensions — arm-software.github.io. https://arm-software.github.io/acle/main/acle.html. [Accessed 14-04-2024].

[3] llama.cpp/ggml-common.h at 132f55 · ggerganov/llama.cpp — github.com. https://github.com/ggerganov/llama.cpp/blob/132f55795e51094954f1b1f647f97648be724a3a/ggml-common.h#L6. [Accessed 15-04-2024].

[4] LTO (GNU Compiler Collection (GCC) Internals) — gcc.gnu.org. https://gcc.gnu.org/onlinedocs/gccint/LTO.html. [Accessed 14-04-2024].

[5] Neon — developer.arm.com. https://developer.arm.com/Architectures/Neon. [Accessed 15-04-2024].

[6] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu. Qwen technical report. arXiv preprint arXiv:2309.16609, 2023.

[7] Y. Bisk, R. Zellers, R. Le bras, J. Gao, and Y. Choi. Piqa: Reasoning about physical commonsense in natural language. Proceedings of the AAAI Conference on Artificial Intelligence, 34(05):7432–7439, Apr. 2020.

[8] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac'h, H. Li, K. McDonell, N. Muennighoff, C. Ociepa, J. Phang, L. Reynolds, H. Schoelkopf, A. Skowron, L. Sutawika, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou. A framework for few-shot language model evaluation, 12 2023.

[9] G. Gerganov. GitHub-ggerganov/llama.cpp: LLM inference in C/C++. https://github.com/ggerganov/llama.cpp, 2023. [Accessed 12-04-2024].