Emulação de Large Language Models para RISC-V usando QEMU

Giovani L. B. Santos[1] **- email giovanilbsantos@gmail.com**
Lucas Wanner[1] **- email wanner@unicamp.br**

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brasil

Abstract. *This work aims to execute DeepSeek on a RISC-V emulator, with the objective of showing the viability of executing LLMs in RISC-V devices, as well as in embedded systems. This is important for the purposes of embedding AI, infra independency, energetic efficiency and open source. We used the QEMU emulator to execute RISC-V and obtained the tokens per second count in various tests. Our best result was 1.2415 tks/sec, a value considered slow, albeit viable.*

Resumo. *Este trabalho tem como objetivo executar o DeepSeek em um emulador RISC-V, com o intuito de mostrar a viabilidade de executar LLMs em aparelhos RISC-V, e também em sistemas embarcados. Isso é importante para os propósitos de integração de IA, independência de infra, eficiência energética e código aberto. Utilizamos o emulador QEMU para rodar RISC-V e obtivemos a contagem de tokens por segundo em vários testes. O melhor resultado obtido foi 1,2415 tks/s, valor considerado lento, porém viável.*

1. Introdução

Desde o lançamento do GPT-3 em 2020, os LLMs vêm causando um impacto tremendo comparado com o breve período que eles existem por, e a tendência é esse impacto aumentar conforme o tempo passa. Por isso, é importante que os LLMs estejam disponíveis em diferentes escalas, para que seja possível utilizá-los tanto em computadores de ponta, onde memória e processamento não estão em falta, quanto em dispositivos de borda, onde o contrário se aplica. Este trabalho concentra-se nesses dispositivos, utilizando o emulador QEMU com a arquitetura RISC-V para executar o LLM DeepSeek-R1-Distill-Qwen-1.5B, e obteve valores de tokens por segundo entre 0.0358 tks/s e 1,2415 tks/s em vários testes realizados.

O maior desafio em rodar LLMs em um emulador RISC-V é desempenho. O principal gasto de tempo durante o funcionamento de um LLM é em multiplicação de matrizes. Algoritmos de GEMM e instruções de MAC são, portanto, cruciais para um desempenho bom, assim como um hardware e um kernel especializados para esse tipo de operação e as ferramentas utilizadas para acelerá-los, como AVX e AMX, que criam novas instruções vetoriais e matriciais, respectivamente, para a ISA x86, tanto como CUDA, que habilita o uso das GPUs da NVIDIA para processamento acelerado. Contudo, nenhuma dessas tecnologias estão disponíveis para RISC-V, que possui apenas uma extensão vetorial, a RVV. Existem também alternativas para o CUDA que funcionam com RISC-V, como OpenCL ou ROCm, mas mesmo utilizando essas ferramentas, o desempenho

ainda está perceptivelmente abaixo daquele obtido com as tecnologias licenciadas. Felizmente, uma extensão de instruções matriciais para RISC-V está sendo desenvolvida junto de hardware especializado para processá-la, o que pode ajudar bastante a diminuir a diferença de desempenho, ou até mesmo extingui-la, quando estiver pronta.

Outro problema é suporte: mesmo que RISC-V seja a ISA que mais cresce hoje em dia, a maior parte da tecnologia existente ainda é feita para funcionar com x86 e ARM, que veem uso no mercado, ao contrário de RISC-V, cujo maior uso ocorre, por enquanto, no meio acadêmico. A consequência disso é que grande parte das ferramentas existentes para RISC-V são novas e mantidas por times pequenos, o que complica a utilização da ISA.

2. Metodologia

O único hardware existente que roda RISC-V nativamente são placas de desenvolvimento, não há equivalentes a desktops ou notebooks, embora estes já estejam em desenvolvimento, podendo lançar até mesmo esse ano. Ao invés de utilizar uma dessas placas, decidimos optar por um emulador RISC-V para um ambiente x86. O escolhido foi o QEMU, que possui o melhor suporte para a arquitetura dentre todos os emuladores disponíveis e um desempenho razoável. Outra grande vantagem é que ele é um emulador de sistema completo, com a ISA, memória, disco, I/O, booting, SMP, entre outros processos todos emulados até os detalhes. Porém, para o QEMU emular uma arquitetura em um ambiente com outra arquitetura, é necessário fazer cross-compiling de todas as ferramentas utilizadas, ou seja, compilar elas dentro da emulação RISC-V com o intuito de rodar em hardware x86, o que dificulta a portabilidade, já que a maior parte da atenção da comunidade que cria essas ferramentas é dedicada a usos mais comuns. O QEMU também oferece emulação de user-mode, que apenas traduz arquivos binários de uma arquitetura para outra, emulando o mínimo necessário, como CPU e Syscalls. A vantagem do user-mode é que ele é bem mais rápido do que uma emulação de sistema completa, mas é menos preciso, especialmente quando seu objetivo envolve desempenho de hardware. Por fim, o QEMU possui uma simulação baremetal, não coberta neste artigo, já que essa emulação não possui nem as bibliotecas essenciais, como OpenBLAS, nem sistema operacional, inviabilizando seu uso neste trabalho.

Além do QEMU, utilizamos Ubuntu como sistema host, Debian dentro da emulação de sistema, llama.cpp[x] para inferência e OpenBLAS[x] para os algoritmos de GEMM, junto de todas as dependências necessárias para utilizar essas ferramentas, como a RISC-V GNU Toolchain.

3. Experimentos

Realizamos dois experimentos, cada um com múltiplos testes, para obter o número de tokens em dois modelos levemente diferentes, o DeepSeek-R1-Distill-Qwen-1.5B-Q2_K e o DeepSeek-R1-Distill-Qwen-1.5B-Q8_0, referidos como Q2 e Q8, respectivamente, daqui em diante. Ambos os experimentos foram realizados em um computador com um processador Intel i5-10400, 8 GB de memória RAM e sem placa de vídeo. As entradas para todos os testes seguiram o seguinte formato:

```
qemu-riscv64 --cpu rv64,g=true,c=true,v=true, vext_spec=v1.0,
vlen=256,elen=64 llama.cpp/build/bin/llama-cli
```

```
-m llama.cpp/models/DeepSeek-R1-Distill-Qwen-1.5B-Q2_K.gguf
--prompt "Who was Julius Caesar?" -n 100 --repeat-penalty 1.1
-no-cnv --no-warmup
```

Quanto às flags, –cpu habilita suporte a RVV, e foi mantida nesse formato até para os testes sem RVV, por consistência (desabilitar RVV teve impacto desprezível no número de tokens por segundo). –prompt permaneceu o mesmo também por consistência, -n 100 limita o número máximo de tokens gerados, encerrando a execução quando chega em 100, o que previne, por exemplo, o tempo de geração de 100,9 tokens de ser erroneamente classificado como o de 100, além de padronizar um objetivo para cada teste alcançar (nenhum deles terminou em menos de 100 tokens). Por fim, –repeat-penalty 1.1 apenas ajuda o modelo a não ficar travado gerando as mesmas frases tão facilmente, -no-cnv faz com que o modelo não fique esperando outro prompt depois de terminar sua resposta e –no-warmup pula a fase de aquecimento do modelo, o que agiliza bastante os testes mais lentos, sendo mantido em todos por consistência.

Após terminar a execução, o llama.cpp disponibiliza os resultados no seguinte formato:

```
sampling time = 894.59 ms / 106 runs (8.44 ms per token, 118.49 tokens per second)
load time = 25228.29 ms
prompt eval time = 22135.37 ms / 6 tokens (3689.23 ms per token, 0.27 tokens per second)
eval time = 412247.85 ms / 99 runs (4164.12 ms per token, 0.24 tokens per second)
total time =  438433.23 ms / 105 tokens
```

Dessa saída, os dados importantes fornecidos são *sampling time*, que é o tempo que o modelo passa decidindo aleatoriamente qual é o próximo token a ser passado como saída, *prompt evaluation time*, o tempo que o modelo passa interpretando a entrada, e *evaluation time*, o tempo que o modelo passa gerando a saída em si a partir da entrada. Esses três tempos foram medidos para todos os testes.

3.1. Tokens por Segundo (Q2)

O primeiro experimento realizado executou o Q2 e mediu a quantidade de tokens por segundo que podem ser obtidos dependendo de duas variáveis: se estamos utilizando System-Mode ou User-Mode, e se a extensão vetorial RVV está ou não habilitada, resultando em quatro testes. Os resultados obtidos foram os seguintes:

	User-Mode com RVV	System-Mode com RVV	User-Mode sem RVV	System-Mode sem RVV
Sampling Time	118,4900 tks/s	251,9490 tks/s	334,9237 tks/s	248,8263 tks/s
Prompt Evaluation Time	0,2711 tks/s	0,0428 tks/s	1,2553 tks/s	0,1315 tks/s
Evaluation Time	0,2401 tks/s	0,0358 tks/s	1,0000 tks/s	0,1095 tks/s

Tabela 1. Resultados obtidos no experimento de Tokens por Segundo com o modelo Q2 para cada uma das quatro configurações medidas.

Como esperado, User-Mode é muito mais rápido do que System-Mode. Entretanto, observou-se que o DeepSeek apresentou maior velocidade no modo de usuário sem RVV em comparação com sua utilização, e não por uma pequena margem: sem a extensão, o sampling foi feito três vezes mais rápido, os tokens foram gerados 4 vezes mais rápido, e a verificação do prompt quase 5 vezes mais rápida. Esse resultado faz sentido, porém, quando se leva em consideração o fato que esses são os resultados de uma

emulação das instruções, e não do RISC-V em si. RVV faz uso de várias funcionalidades do hardware, como paralelismo e acesso eficiente à memória cache, para conseguir obter um desempenho maior. O QEMU, por sua vez, emula as instruções sem possuir essas mesmas funcionalidades, o que ocasiona em um aumento do número de instruções sem o ganho de desempenho proporcionado pelo uso inteligente do hardware, culminando em um ganho negativo de desempenho, um *overhead* de instruções.

3.2. Tokens por Segundo (Q8)

O segundo experimento realiza os mesmos testes que o primeiro utilizando Q8. Os resultados obtidos foram os seguintes:

	User-Mode com RVV	System-Mode com RVV	User-Mode sem RVV	System-Mode sem RVV
Sampling Time	116,7401 tks/s	248,2784 tks/s tks/s	335,0931 tks/s	242,6018 tks/s
Prompt Evaluation Time	0,5805 tks/s	0,0950 tks/s	1,5741 tks/s	0,2109 tks/s
Evaluation Time	0,4658 tks/s	0,0788 tks/s	1,2415 tks/s	0,1806 tks/s

Tabela 2. Resultados obtidos no experimento de Tokens por Segundo com o modelo Q8 para cada uma das quatro configurações medidas.

O Q8, por ter uma quantização de 8 bits, deveria ser mais pesado, preciso e lerdo do que o Q2, mas os resultados mostram que os valores de sampling e evaluation foram maiores para todos os testes, pelo mesmo motivo que o RVV deixa as coisas mais lentas, o *overhead* de instruções. O Q2 utiliza *Block-wise K-Quantization*, uma técnica que, assim como o RVV, é mais complicada, o que dentro de um emulador deixa as coisas mais lentas em comparação com um abordagem mais direta tal qual a feita pelo Q8, *Linear Quantization*.

4. Conclusão

Os resultados demonstram a viabilidade de execução do LLM DeepSeek-R1-Distill-Qwen-1.5B no QEMU. O melhor desempenho, cerca de 1,24 tokens por segundo de evaluation, foi obtido na configuração user mode sem RVV com quantização Q8, mais rápida que as outras por ser a mais simples, ou seja, a que utiliza o menor número de técnicas que, em um simulador, acabam causando lentidão devido ao *overhead* de emulação de *syscalls* e instruções vetoriais maior do que a aceleração obtida. Os próximos passos deste trabalho serão testar modelos com mais parâmetros e em outros emuladores além do QEMU, e possivelmente obter novas medidas, como número de ciclos, e comparar o desempenho obtido em um ambiente RISC-V com um ambiente x86, com e sem o uso de emulador.

Referências

Fang, J., Varbanescu, A. L., and Sips, H. (2011). A comprehensive performance comparison of cuda and opencl. *International Conference on Parallel Processing*, pages 216–225.

Gerganov, G. (2023). llama.cpp. `https://github.com/ggml-org/llama.cpp`.

Moore, S. K. (2023). Risc-v laptops now available. `https://spectrum.ieee.org/risc-v-laptops`.

OpenMathLib (2025). Openblas. `https://github.com/OpenMathLib/OpenBLAS`.

Team, Q. (2025). Qemu documentation. https://www.qemu.org/docs/master/.