Language Models are the new Doom

Caio Madeira, Maurício Cecílio Magnaguagno, Dalvan Griebler

[1] Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

{caio.madeira, mauricio.magnaguagno}@edu.pucrs.br, dalvan.griebler@pucrs.br

Abstract. *As more devices include a display, more probable they may run Doom. Various techniques are used to limit the processing required by the game. A parallel can be drawn with small language models, which are being ported to a wide variety of devices, including legacy ones. While less capable than their large-scale counterparts, these models retain the essence of natural language processing. In this work, we map parallels between Doom and language models, focusing on a TinyLlama model trained on the TinyStories dataset ported to the PlayStation Portable, with hyper-parameter value experimentation.*

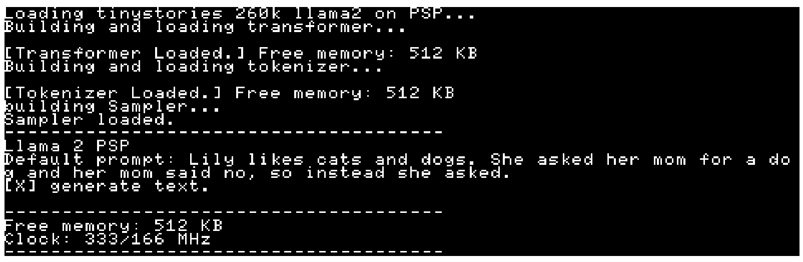
1. Introduction

Since its release, the game *Doom* has been widely ported, especially by the community after its source code was released. This history exemplifies a core challenge in software portability: while porting to more powerful hardware is relatively simple, adapting it for devices with limited resources or different architecture requires complex optimizations and significant design changes to overcome memory and display constraints. This challenge is directly analogous to the current landscape of Large Language Models (LLMs). Similar to games, LLMs are complex pieces of software demanding hardware requirements, such as significant VRAM for execution and extensive floating-point calculations. These models, typically based on the Transformer architecture, are trained on vast text corpora to perform tasks like question answering, text summarization, and text translation [Touvron et al. 2023]. The quest for efficient models that require a fraction of the resources of their large counterparts is intensifying with the proliferation of mobile and smart devices and it will be our doom to be limited to only large language models controlled by few companies, compromising user privacy and ensuring dependence on their infrastructures, thus motivating the exploration of running AI on limited hardware.

Therefore, the main goal of this work is to provide a technical analysis of the challenges and effort required to port a language model inference engine to a limited and unconventional hardware platform, the PlayStation Portable (PSP). We detail this process by attempting to run a small, non-quantized language model, focusing specifically on the necessary toolchain, algorithmic strategies, architectural adaptations, and memory management techniques developed to operate within the device's limitations. Experiments are conducted by varying hyper-parameters (top-p and temperature), to analyze output quality.

2. Methodology

Our project is a port of the llama2.c implementation for the PlayStation Portable (PSP), initially inspired by Maciej Witkowiak's llama2.c64 port for the Commodore 64 (C64).



```
Loading tinystories 260k llama2 on PSP...
Building and loading transformer...

[Transformer Loaded.] Free memory: 512 KB
Building and loading tokenizer...

[Tokenizer Loaded.] Free memory: 512 KB
building Sampler...
Sampler loaded.
--------------------------------------
Llama 2 PSP
Default prompt: Lily likes cats and dogs. She asked her mom for a do
g and her mom said no, so instead she asked.
[X] generate text.

--------------------------------------
Free memory: 512 KB
Clock: 333/166 MHz
--------------------------------------
```

Figure 1. During initialization, Transformer, Tokenizer and Sampler are built and loaded, leaving only 512 KB of RAM available. After the setup, the hard-coded prompt, free memory and clock frequency are displayed on screen while the option to generate a prompt is given to the user by a button press.

We use the same 260k-parameter model from the TinyLlama series [Zhang et al. 2024], which was trained on the TinyStories [Eldan and Li 2023] dataset, composed of simple, GPT-generated children's stories. The PSP capabilities changed over the years. The first series, called PSP-1000, has 32 MB of RAM and a CPU based on a MIPS R4000 architecture with a variable frequency between 1 and 333 MHz. Later versions, like PSP-2000, PSP-3000, PSP Go (N1000) and PSP Street (PSP-E1000) came with 64 MB of RAM. Moreover, in this paper we port a Llama 2 model to a PSP-2000 hardware with Custom Firmware (CFW), which is necessary to provide kernel access to the user.

2.1. Development of port to PlayStation Portable

Our first step was to adapt the original C implementation of Llama 2 for the PSP. The llama2.c project allows for the inference of Llama 2 family language models using just a single C file, without requiring complex extra libraries. As a result, llama2.c can be compiled and executed on a wide variety of systems and hardware. The primary technical adaptation was to split the single C file, similar to C64 version, to improve readability and understanding. While the original llama2.c implementation uses the *mmap()* function to efficiently map the file into virtual memory (a POSIX system call), our PSP version (*llama2-psp*) uses the more portable approach of *malloc()* and *read()*. This loads the entire file into RAM, which is more costly in advance but avoids platform-specific dependencies. For development, we used the PSPSDK toolchain (*psp-gcc*, *Newlib*) to compile the project. We also implemented kernel callback functions, which are essential for managing the application's interaction with the console's operating system, particularly for handling user exit requests.

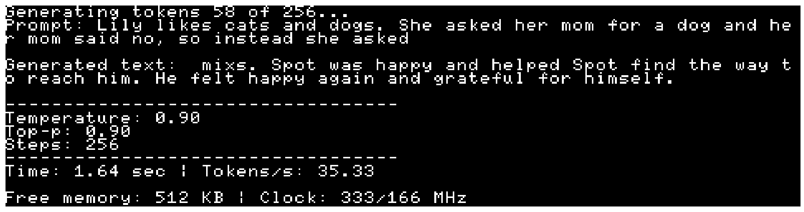
Finally, for the user interface, we opted for a console, which can be seen in Figure 1. This decision was made because the official Graphics Utility is too resource-intensive; for instance, allocating double buffered framebuffers for the 480x272, 32 bit per pixel (4 bytes) screen consumes 1 MB of RAM (i.e., 480 * 272 * 4 * 2 = 1.048.576 bytes). This highlights the importance of economizing resources on such limited hardware, which is crucial for the project's viability.

We developed three distinct tests by varying two parameter values: *temperature* and *top-p*. The *temperature* parameter influences the model capacity to generate creative texts based on a given prompt. In addition, the *top-p* parameter filters the most probable words. Both parameters can assume values $0.0 \leq \text{parameter} \leq 1.0$ and are set to the same value in our tests. The *step* parameter is used to limit the amount of tokens generated and

Table 1. Results from three different configurations with the same input prompt.

Temperature & Top-p	Tokens_Generated	Tokens/s	Total_Generation_Time (s)
0.2	30	36.06	0.83
0.6	52	35.36	1.46
0.9	64	35.11	1.82

is set to 256 in our tests. This approach was designed to isolate the impact of these settings on the quality of the output rather than performance. Additionally, the same input prompt *"Lily likes cats and dogs. She asked her mom for a dog and her mom said no, so instead she asked "* was used for all tests to ensure a fair comparison and the results are summarized in Table 1. Some basic metrics are used to evaluate the time to process an output, Total_Generation_Time = (End_Tick - Start_tick) / Tick_resolution, and the amount of tokens generated, Tokens_per_second = Tokens_Generated / Total_Generation_Time. A result example is displayed at Figure 2. The results suggest that there is not a direct correlation between the settings and performance. For instance, with both parameters at 0.2, the model is expected to be more conservative and deterministic in its word choices, which happened and resulted in a lower Total_Generation_Time compared to the test where the parameters were set to 0.9. The 0.9 setting gives the model more freedom to select from a varied set of words, potentially straying from the initial context, resulting in longer output and greater Total_Generation_Time.



```
Generating tokens 58 of 256...
Prompt: Lily likes cats and dogs. She asked her mom for a dog and he
r mom said no, so instead she asked

Generated text:  mixs. Spot was happy and helped Spot find the way t
o reach him. He felt happy again and grateful for himself.

-----------------------------------
Temperature: 0.90
Top-p: 0.90
Steps: 256
-----------------------------------
Time: 1.64 sec ¦ Tokens/s: 35.33

Free memory: 512 KB ¦ Clock: 333/166 MHz
```

Figure 2. Example text generation result where the *temperature* and *top-p* hyperparameters were set to their to 0.9. The user interface displays both the initial prompt and the model's output for a qualitative assessment of contextual relevance. This run completed in 1.64 seconds, achieving a rate of 35.33 tokens per second and reaching 58 tokens of 256.

2.2. Comparison between Doom ports and language models

It is possible to map several elements from Doom to Language Models, as seen in Table 2. The speed in which they are executed can be measured in either frames or tokens per second, which is affected by resolution and textures sizes, in a similar way to quantization of the model. Also, both change the amount of memory required to represent more or less detail, which is one of the first limitations when comparing current and old devices. Although specialized hardware is not required, both Doom and Language Models are better executed with a sound board or GPU, respectively. Enhancements in light rendering and spatial sound increase realism through more complex computation, similar to an enhanced text *corpus*, which will take more time to gather from specialized sources or in volume to be processed during the training phase. Both ports only became possible as their sources became available and a community was formed around them. If we consider Llama as one of the first large language model able to be executed locally, llama2.c is the counterpart, targeting less powerful devices with a small language model. The llama2.c project is the root of many ports, including machines with a Pentium II and 128 MB of RAM under Windows 98[1] generating 39.31 tokens/s with 260K parameters, DOS with a 486 processor[2] generating 2.08 tokens/s with 260K parameters, and a Commodore 64 equipped with 2 MB of additional RAM to support the 260K parameters model[3].

Table 2. Parallel between Doom and language models.

Doom	**Language Model**
Ported from released source code	Ported from released Llama
Frames/s	Tokens/s
Resolution and textures	Quantization
Sound board for effects	GPU for speed
Enhanced light and sound	Enhanced text corpus

3. Conclusions and Future Work

In this work, we made an analogy between the highly ported game Doom and Language Models, while showing how a small language model can be ported to a PSP, which is more limited than the hardware usually employed to execute Language Models. The port demanded a combination of the Llama 2 implementation and a deep knowledge of the PSP's development ecosystem. Our implementation is available at GitHub[4]. Future work includes testing others sorting algorithms on vocabulary sort in the original implementation, more parameter tests and token count variability, more evaluation metrics for text quality covering coherence, fluency and contextual relevance, tests with the int8 quantization version provided by Karpathy[5] aiming to reduce main memory usage and an a low-level analysis on MIPS PSP for looking for optimizations on assembly level. Ports of small language models today are very similar to the essence of Doom open-source ports, that started being ported to computers and video game consoles, and eventually reached much smaller and strange devices. With TinyLlama running on PSP, it's possible to integrate into a simple game running on plataform, creating an offline, low-latency potential example of an Edge AI application, within the still available resources of the hardware while executing the model. One can extrapolate that, in the future, more limited devices (e.g., embedded systems) will be able to interact through natural language, as models become more optimized and hardware more flexible.

References

Eldan, R. and Li, Y. (2023). Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759.*

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288.*

Zhang, P., Zeng, G., Wang, T., and Lu, W. (2024). Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385.*

[1]https://blog.exolabs.net/day-4/
[2]https://yeokhengmeng.com/2025/04/llama2-llm-on-dos/
[3]https://github.com/ytmytm/llama2.c64
[4]https://github.com/caiomadeira/llama2-psp
[5]https://github.com/karpathy/llama2.c/blob/master/runq.c