# SlowLLM: large language models on consumer hardware

Christophe Cerisara

# SlowLLM: large language models on consumer hardware

Christophe Cerisara

March 4, 2023

## 1 Introduction

*Important note: this is a work-in-progress report and not a publication-quality research paper.*

Democratizing large language models (LLM) involves both distributing pre-trained models freely with open-source like licenses, and enabling these models to run at the lowest possible cost for the end-user. We focus in this work on the second challenge: reducing the cost for running LLMs. More precisely, we distribute an easy-to-use pytorch code that enables to run a 176-billion parameters Bloom or Bloomz model on a consumer computer without any GPU and with as few as 16GB of RAM. Of course, this comes at the cost of a much slower inference time than what may be achieved with high-end resources. However, there are use cases where such slow but nearly free usage of LLMs is useful, if only for the sake of giving a wider public the possibility to try such models at no cost.

The algorithmic principle of the proposed code is very simple and well-known, but there are many ways to actually implement it, and we propose here a brief review of some of them along with identified advantages and limitations, as well as a functional code to start with. The code is open-sourced in `https://github.com/cerisara/slowLLM`.

## 2 Related works

*Accelerate* is a general-purpose and flexible library developped by Huggingface that exploits the available hardware to run inference with large language models. It thus first attempts to load the model's weights onto the available GPUs, and offload the rest of the parameteres in CPU RAM or on an NVMe hard drive. The power of this library lies in its flexibility, making it compatible with a large range of various hardware setups. The proposed approach in this work is rather dedicated to a specific model family (Bloom*) and a typical low-end consumer-grade hardware, namely a computer without GPU and with 16GB of

RAM. These additional restrictions enable specific optimizations, such as not requiring disk offloading for inference, working with the vanilla pickled model weights, simple and easy to adapt code base and enabling finetuning.

The principle of loading the layers of a transformer one by one in memory is extremely simple and obvious, but caveats exist about how to do it efficiently. For instance, a very good blog post [1] details in a nice and pedagocical way how to achieve this in a few lines of codes. However, this code is not up to date with the latest transformers library versions, and it reimplements the forward pass through the transformers, which is nice for an educational purpose, but also prevents the model from exploiting some optimizations in the transformers library and makes it more difficult to extend the code for finetuning or parameter-efficient training.

Other approaches exploit multiple computers working collaboratively, such as developped in [1, 3, 2], or by TogetherComputer (`https://www.together.xyz`) or yet Hivemind and Petalsi (`https://github.com/learning-at-home/hivemind`). However, these approaches require the involvment of several peers, which give rise to new major challenges, such as costs, efforts and time sharing, malevolent peers, bandwidth and peer failure, data privacy... Yet, this track of research has many potential of development, and we have also implemented our own working proof of concept in `https://github.com/cerisara/slowLLM`, which relies on a central server connected with cpu-only clients through web sockets. But this collaborative proof-of-concept is alpha-quality and not as mature as the cited alternatives. Conversely, our main proposed approach only rely on a single personal computer that does not even need to be connected to the internet.

# 3 Method

The proposed approach implements a simple layer-by-layer forward pass with pipeline parallelism. Let $\theta(L_0), \theta(L_1), \ldots, \theta(L_{70})$ be the parameters for respectively Bloomz's word embeddings (called layer 0), layer 1, and so on until layer 70. The parameters of the final language modeling head are tied to those of the word embeddings.

Inference processes as follows:

- Load in RAM a batch of maximum 50 input sentences $(x_1, \ldots, x_{n<50})$ where each $x_i$ is a sequence of tokens. The maximum number of sentences is set to 50 to be sure that the total required RAM stays below 16GB.
- let $z_1, \ldots, z_n = x_1, \ldots, x_n$
- Iterate for $i$ in $0 \ldots 70$:
  - Load in RAM $\theta(L_i)$

[1] `https://nbviewer.org/urls/arteagac.github.io/blog/bloom_local.ipynb`

  - Make a forward pass with all $z_1, \ldots, z_n$ through $L_i$ to get the output and replace $(z_1, \ldots, z_n)$ with them
  - free the memory allocated to $\theta(L_i)$

- Load in RAM $\theta(L_0)$ into the final language modeling head
- Make a forward pass with all $z_1, \ldots, z_n$ through the final LM head to get the logits of the predicted next token and the sentences perplexity.

### 3.1 Specific heuristics to reduce memory requirements

Because pytorch does not support 16-bits operations on the CPU, a wrapper around the dense linear layer is implemented that converts the input 16-bits tensors into 32-bits, computes the matrix multiplication and then converts back the result into 16-bits. An alternative would be to implement everything in 32-bits, but this would require too much memory.

For inference, a single activation tensor is kept in memory and is overwritten after every layer. However, to fine-tune the model, all activation tensors shall be kept, which does not fit within 16GB of RAM. So for training, these activations have to be saved in disk during the forward pass, which greatly slows down the code.

## 4 Benchmarks

The following benchmark is realized on the following hardware: personal computer with 24GB of RAM, no gpu, with a NVMe sdd (Micron/Crucial Technology P2 NVMe PCIe SSD (rev 01)) and an AMD Ryzen 5 3600 6-Core Processor.

We made lots of efforts to also also run the same test with the accelerate library, but repeatly failed to make it work on our low-end hardware: we strictly followed the Huggingface recipe (version of Feb. 2023, available at `https://huggingface.co/blog/bloom-inference-pytorch-scripts`) to make inference with the accelerate library and the Bloom model. The only difference is that we are using Bloomz instead of Bloom. First, the accelerate library needed to convert on disk the weights of the model into the "safetensor" format, which takes a lot of time, about two hours on our NVMe SSD drive. Then, the program systematically crashed after some time:

- We first tried by letting accelerate exploit a Titan X 12GB GPU in our desktop, but the program crashed with cuda out-of-memory error.
- We also tried by forcing accelerate to not use the GPU, but then the program got killed after some time by the kernel, most likely because of memory.

We concluded that despite being an extremely powerful library for middle or high-range hardware, the accelerate library (v0.16.0 with transformers v4.26.1

and pytorch v1.13.1+cu117) is still challenging to use on low end hardware as in our setup.

We also adapted the approach proposed in `https://nbviewer.org/urls/arteagac.github.io/blog/bloom_local.ipynb` to the latest transformers library and tried it on our setup.

| #utts | #toks | Accel. | slowLLM | arteagac |
|---|---|---|---|---|
| 1 | 13 | killed | 791 | 4200 |
| 5 | 65 | killed | 1819 | - |

Table 1: Processing time (in seconds) for inference (forward pass only), including initialization and weights loading time.

# 5 Discussion and limitations

The main limitation of slowLLM is the much slower speed required to perform inference as compared to GPU-based solutions. However, the recommended hardware to run Bloom-176b is eight A100 GPUs with 80GB of VRAM each, which very few non-professional people can have, so when you have access to several high-end GPUs, it is obviously much better to use them than to rely on slowLLM. This reduced speed makes text generation a very unsuitable scenario for slowLLM, because text generation nowadays mostly relies on autoregression, which involves sequentially passing through the whole model for every token generated. However, there are many other scenarios that can be achieved with slowLLM, such as text classification, simple yes/no question answering, comparing the perplexity of models or sentences, computing sentence embeddings, and text generation limited to few words only.

Finally, we believe that the type of approach promoted by the proposed slowLLM software, i.e., running everything slowly, on "heritage" low-end resources and purely locally, is one of the most promising direction to greatly reduce the maintenance, replacement and exploitation costs of AI deployment.

# References


[1] Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Quentin Lhoest, Anton Sinitsin, Dmitry Popov, Dmitry Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, Denis Mazur, Ilia Kobelev, Yacine Jernite, Thomas Wolf, and Gennady Pekhimenko. Distributed deep learning in open collaborations. *ArXiv*, abs/2106.10207, 2021.

[2] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. SWARM parallelism: Training large models can be surprisingly communication-efficient, 2022.

[3] Timo Schick, Jane A. Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. PEER: A collaborative language model. In *International Conference on Learning Representations*, 2023.