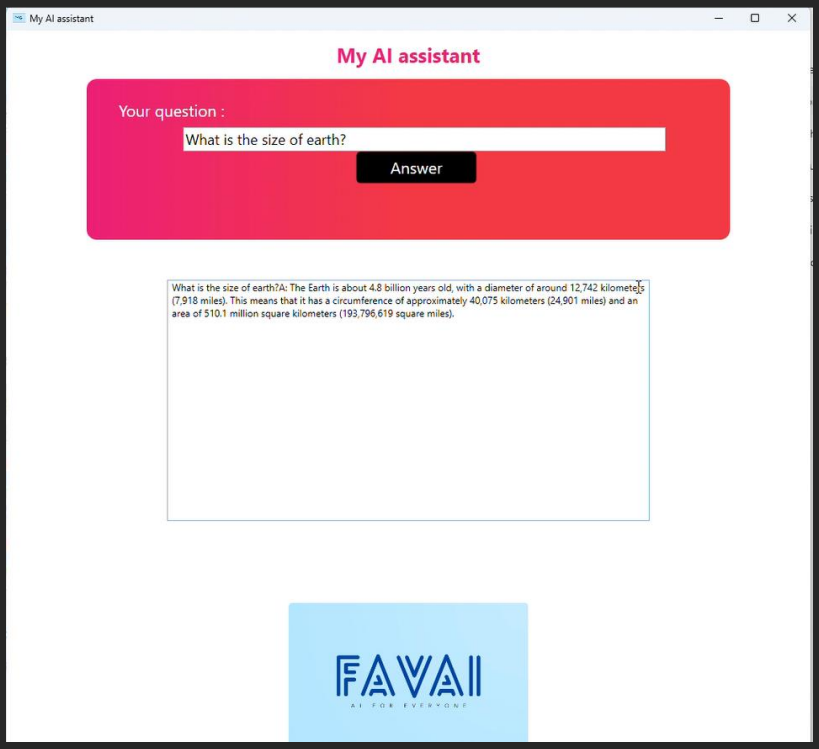
Local LLM on a Laptop



Introduction

OpenAI revolutionized the AI landscape with ChatGPT, reigniting the AI hype with generative AI (GenAI). As organizations began to adopt large language models (LLMs) like ChatGPT, they quickly recognized both the potential efficiency gains for employees and the risks associated with uncontrolled use. Publicly available LLMs can learn and retain information from users, posing a risk of inadvertently exposing confidential company data.

A local LLM for privacy

To address this concern, one of the solutions would be to have the LLM directly running on the laptops. This approach ensures that the LLM is isolated from the internet and the public use, safeguarding your sensitive information. Since no information is sent through the web, it ensures better privacy by reducing the attack surface. Additionally, it provides a reduced latency, which enhance greatly the user experience. But most of all, it doesn't need internet to run.

How can that work if ChatGPT and other LLM must run in the cloud??

First, multiple kinds of models exist. Some are big with dozens of billions of parameters like Llama2 70B (70 billion parameter) or ChatGPT 4o. These kinds of LLMs require a lot of compute capabilities and a huge quantity of memory. That's the reason why they can only run in data centers as of today, with very powerful GPUs and AI Accelerators. But smaller LLMs exist, going down to a few billion parameters, and a weight of only a few Gb. This is the case with Microsoft's Phi 2. The model has 2.7 billion parameters, and depending on the quantization method, it can weight less than 2Gb in GGUF format. Of course, the bigger the model, the better it is usually. But for its size, Phi 2 gives compelling answers at a very low latency with a high token throughput (see performance results below).

Leverage Intel accelerators

Since many years, Intel has worked to accelerate AI workloads. Whether it was through hardware accelerators with its NPU, or new instruction sets like VNNI, or even software with the Intel distribution of TensorFlow and the Intel neural compressor. Here, I tried to leverage the VNNI instruction set. That instruction set fusion 3 operation into 1 for Int8 models while leveraging AVX512. AVX512 is a longer register, made of 512 bits, which allows the CPU to make more operations in one shot. Usually, registers are 256 or 128 bits wide. The speed up is of the same order of magnitude as the number of bits. However, it isn't the case since the frequency usually decreases slightly as the number of bits by register

increases. The increase of bits offloads the decrease of frequency, resulting in some speed up anyway. Usually, the expected speed up is about 2 to a little bit less than 3x. My application relies on llama.cpp, so I built it to leverage AVX512 VNNI, but I also tried to build it without taking these instruction sets into account. Below are my results on an Intel i7-1165G7 @ 2.80GHz with 8Gb of RAM, and Windows 11.

Without AVX512_VNNI
llama_print_timings: load time = 534.72 ms
llama_print_timings: sample time = 88.18 ms / 400 runs (0.22 ms per token, 4536.33 tokens per second)
llama_print_timings: prompt eval time = 680.21 ms / 15 tokens (45.35 ms per token, 22.05 tokens per second)
llama_print_timings: eval time = 47219.19 ms / 399 runs (118.34 ms per token, 8.45 tokens per second)
llama_print_timings: total time = 48160.59 ms / 414 tokens

With AVX512_VNNI

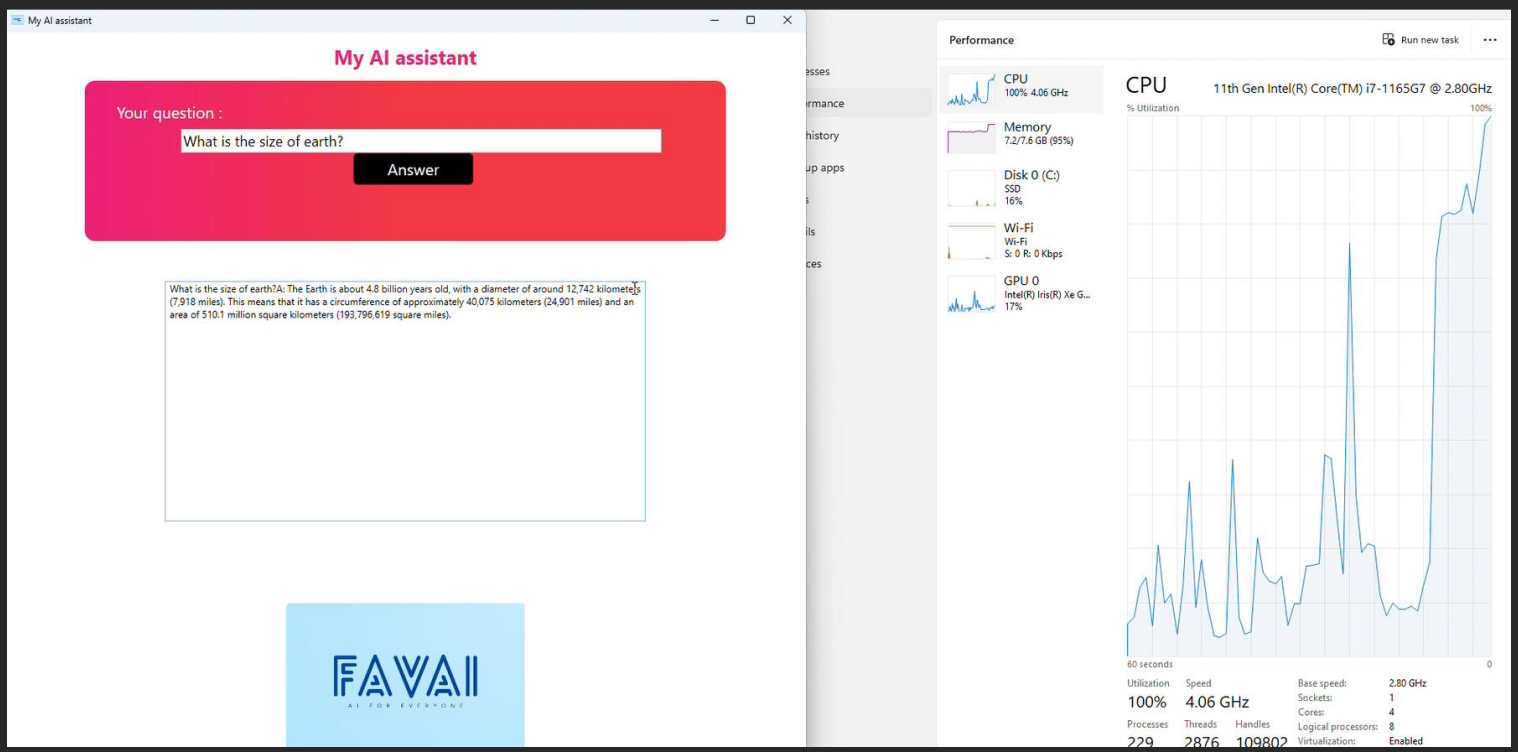
llama_print_timings: load time = 582.99 ms
llama_print_timings: sample time = 113.54 ms / 400 runs (0.28 ms per token, 3522.93 tokens per second)
llama_print_timings: prompt eval time = 629.88 ms / 15 tokens (41.99 ms per token, 23.81 tokens per second)
llama_print_timings: eval time = 50896.78 ms / 399 runs (127.56 ms per token, 7.84 tokens per second)
llama_print_timings: total time = 52056.90 ms / 414 tokens

The results are surprising. AVX512_VNNI doesn't enhance the performance as expected. However, this isn't an issue on Intel side. I couldn't dig deeper but here are my 2 cents:

- Maybe the model doesn't use a lot of the kind of operations benefitting from the AVX512_VNNI. In that case, it doesn't benefit from the 3 in 1 fusion of operation and just face the lower frequency.
- Llama.cpp is quite recent. We may expect it to not fully leverage any feature available in every CPU on the market. When I was working on that project, the enablement of AVX_512 was a hot topic on the llama.cpp github.

How does it look?

Here is the app I built to display my LLM results. You can see the CPU being at 100% utilization to provide the answer as fast as possible.



The result

As you can see above, the LLM is able to provide answers to basic questions at a user-friendly speed of almost 9 tokens per second. The model doesn't weight too much so it can run in the meantime of other applications like Teams or some Office 365 applications without any issue. However, it was keen to hallucinations, trying to always match the output token limit I set.

Next steps

- As you can see, the UI isn't as good as it could be.
- My AI Assistant would need some prompt work to prevent hallucination. If the answer is short, then it should give a short answer and not try to match the limit of token allowed.
- I figured out a way to make the input prompt bigger thanks to the domain extension. I should try to implement it in the future as it could be very useful for summaries.