## Part 1: Break-Even Analysis for LLM Inference

We can model the latency for generating a single token in two parts: memory latency and compute latency.

It takes approximately $2.P$ FLOPS to run inference on our model for a single token, where $P$ is the number of parameters in a model.

**Memory Latency:** The time taken to read the model weights from memory.

$$\text{memory latency} = \frac{2 \cdot P \cdot n_{bytes}}{n_{\text{memory bandwidth}}}$$

Where:

- $P$ is the number of parameters in the model.
- $n_{bytes}$ is the number of bytes per parameter (e.g., 2 for FP16).
- $n_{\text{memory bandwidth}}$ is the memory bandwidth of the hardware (in bytes/sec).

**Compute Latency:** The time taken to perform the calculations (FLOPs).

$$\text{Compute latency} = \frac{2 \cdot P \cdot B}{n_{FLOPs}}$$

Where:

- $B$ is the batch size.
- $n_{FLOPs}$ is the computational throughput of the hardware (in FLOPs/sec).

For a single token, let's simplify and set $n_{bytes} = 1$ for the sake of this analysis. Let's take the specs for an **NVIDIA A100 GPU**:

- $n_{FLOPs} = 3.12 \times 10^{14}$ (312 TFLOPs for FP16)
- $n_{mem} = 1.935 \times 10^{12}$ (1935 GB/s)

Let's do the break-even analysis to find the batch size ($B$) at which the system transitions from being memory-bound to compute-bound. This happens when memory latency equals compute latency.

$$\frac{2 \cdot P \cdot B}{n_{flops}} = \frac{2 \cdot P \cdot n_{bytes}}{n_{mem}}$$

Assuming $n_{bytes} = 1$ and cancelling $2 \cdot P$ from both sides:

$$\frac{B}{n_{flops}} = \frac{1}{n_{mem}}$$

$$\rightarrow B = \frac{n_{flops}}{n_{mem}} = \frac{3.12 \times 10^{14}}{1.935 \times 10^{12}}$$

$$\approx 161.24$$

So, as long as the **batch size is less than ~161, we are memory-bound**. For typical single-user inference (batch size = 1), the process is heavily memory-bound.

**What does this mean?**
This means that the memory bandwidth, not the raw computational power (FLOPs), dictates our generation speed.

$$\text{generation speed} \propto \frac{1}{\text{memory latency}}$$

And for any $batchsize < 161$, the time to get the $batch\_size$ next tokens is always the same, be it $batch\_size = 1\ or\ batch\_size = 160$.

Now, the problem is that we can't naively sample K tokens at a time because every N-th token depends on the token we sampled at position N-1. This autoregressive nature is inherently sequential.

```
|---- Flops ----|------------------ memory ------------------|--> time
t_0
```

*A visual representation showing memory access time dominating compute time for a single token.*

So, memory will dictate the generation speed. This same reason is why applications like `llama.cpp` exist and we can use them at relatively high generation speeds on consumer hardware. We don't need extremely powerful GPUs to run LLMs, but we need extremely fast memory and memory optimization algorithms to fully utilize stronger hardware.

**On the side:** vLLM and SGLang, the most common LLM serving frameworks, are able to reach very high generation speeds, by optimizing memory usage through several techniques, mainly PagedAttention for vLLM and Radix Attention for SGLang. (of course there are many other details, but these are the key features for speedup)

(Let's continue)

To give it more perspective, let's compare an A100 to an Apple M2 chip:

- **A100**: 1935 GB/s, 1248 TFLOPs
- **M2**: 100 GB/s, ~7 TFLOPs
- The compute is **~200x slower** on the M2 ($1248/7 \approx 178$).
- The memory is **~20x slower** on the M2 ($1935/100 \approx 19.4$).

Now, since in both cases the speed for single-token generation is memory-bound, the M2 MacBook will only be **20x slower, rather than 200x slower**. This is why `llama.cpp` exists and it can run fairly well on consumer GPUs or even CPUs, as their memory bandwidth is not as disproportionately lower than their compute power compared to high-end server GPUs. (Again this doesn't mean that the M2 is extremely fast, it means that we are not fully utilizing the GPUs and they have a lot of idle time)

---

## Part 2: Speculative Decoding

Normally, generation is a sequential operation,. We can't predict the $N-th$ token unless we already predicted $N-1$ token, this how inference works, predicting from left to right in an autoregressive manner:

| Input | Output |
|---|---|
| `Prompt` | $O_0$ |
| `Prompt` + $O_0$ | $O_1$ |
| $\vdots$ | $\vdots$ |
| `Prompt` + $O_0 \dots O_{n-1}$ | $O_n$ |

Now comes a clever idea to optimize inference speed: **speculative decoding**.

To make use of parallelization, we will use this trick:

1. We will pass the input through a smaller, faster **"draft" model**.
2. We will generate $k$ output tokens from this draft model. Let's denote them by $d$:

$$d_1, d_2, d_3, d_4, \dots, d_k$$

We will use this output as a draft output for the big, main model.

**Note:** Both the draft model and the main model should use the same tokenizer.

Next, instead, we will use the output of the draft model. But since we have the whole draft output vector already, we can run the main model on all draft tokens **in parallel** in a single forward pass.

| Input | Output (from main model) |
|---|---|
| `Prompt` | $O_0$ |
| `Prompt` + $d_0$ | $O_1$ |
| `Prompt` + $d_0$ + $d_1$ | $O_2$ |
| $\vdots$ | $\vdots$ |
| `Prompt` + $d_0 + \dots + d_{n-1}$ | $O_n$ |

Remember that for **A100** we will need the same delay to get $1$ or $k$ output tokens in parallel for $k < 161$ (from part 1). So getting all the output vector from the main model will now cost us the same time as generating single token using the same model.

Here, we will compare the two output vectors: the draft vector $d$ and the main model's verification vector $O$.

Draft vector:

$$[d_1, d_2, d_3, \dots, d_n]$$

Main model verification:

$$[O_1, O_2, O_3, \dots, O_n]$$

- Is $d_1 == O_1$? True. (Accept $d_1$)
- Is $d_2 == O_2$? True. (Accept $d_2$)
- $\vdots$

- Is $d_x == O_x$? **False**. (Reject $d_x$ and all subsequent tokens. Drop the rest of the draft).
- $\vdots$
- Is $d_n == O_n$? (This check is not reached if a previous one fails).

If the first mismatch occurs at position $x$, we have successfully generated $x - 1$ tokens in parallel. Now we only need to generate the correct token at position $x$ by running a normal forward pass on the main model with the prefix ending in the accepted tokens, plus some extra delay from dumping the unrequired draft outputs.

**Summary of Time/Benefit**

So in total, the time to generate a sequence of tokens is modified. Instead of N sequential steps, we take $N$ small model steps (to generate draft vector) and one large model step (to verify in parallel).

**Why does it work?**

This works because most of the time the draft tokens are accepted, as they often represent "easy" or predictable parts of the sequence. For "harder" tokens, where the big model disagrees, we simply fall back to the original sequential generation speed for that single token, with the small overhead of having run the draft model. This allows us to generate multiple tokens for the price of a single pass through the large model, significantly speeding up inference.