Accelerating LLM Inference Using Multithreading

Report

The solution to the problem is provided in llama2_3035946760.c and the tools used to test/benchmark it are provided in the Testing directory, more details about testing can be found in the README included in the directory.

Methodology

We perform testing with some threads between 0 and 32, where the number represents the child threads created. We then record, parse and process the output. See more in the README found in the Testing directory.

Results

Thread Numbers	Speed (tok/s)	User Time	System Time	User Time / Sys Time
0	50.960	5.004	0.087	57.197
1	42.942	5.873	0.275	21.344
2	72.804	5.923	0.277	21.401
4	115.132	6.088	0.327	18.635
6	129.588	6.703	0.465	14.421
8	144.645	7.157	0.504	14.213
10	162.537	7.361	0.614	11.985
12	168.613	8.266	0.707	11.698
16	174.171	8.554	1.036	8.257
18	174.275	9.358	1.285	7.284
20	166.034	10.506	1.451	7.241
24	166.531	10.884	2.166	5.025
32	149.168	12.478	4.316	2.891

Note that these are averages obtained by running 100+ random seeds (on the CS Server) for each thread. Please check *results.txt* and the Testing/*Scores* directory for the complete test results and data (along with the testing and graphing scripts).

We see clear patterns here, except for the jump from zero child thread to one child thread, we see that having more threads is generally better. We see that we do however get diminishing returns on the performance and—in fact—after a certain number of threads also start to *lose on* performance.

This phenomena is likely explained by the fact that after we have a certain large amount of threads, the work assigned to each thread is small in comparison to the overhead of creating and handling that thread. And so we have a sweet spot between the performance and number of threads used. In our case it lies between 15-20 threads as we can see in Figure 1 (shown on the following page).

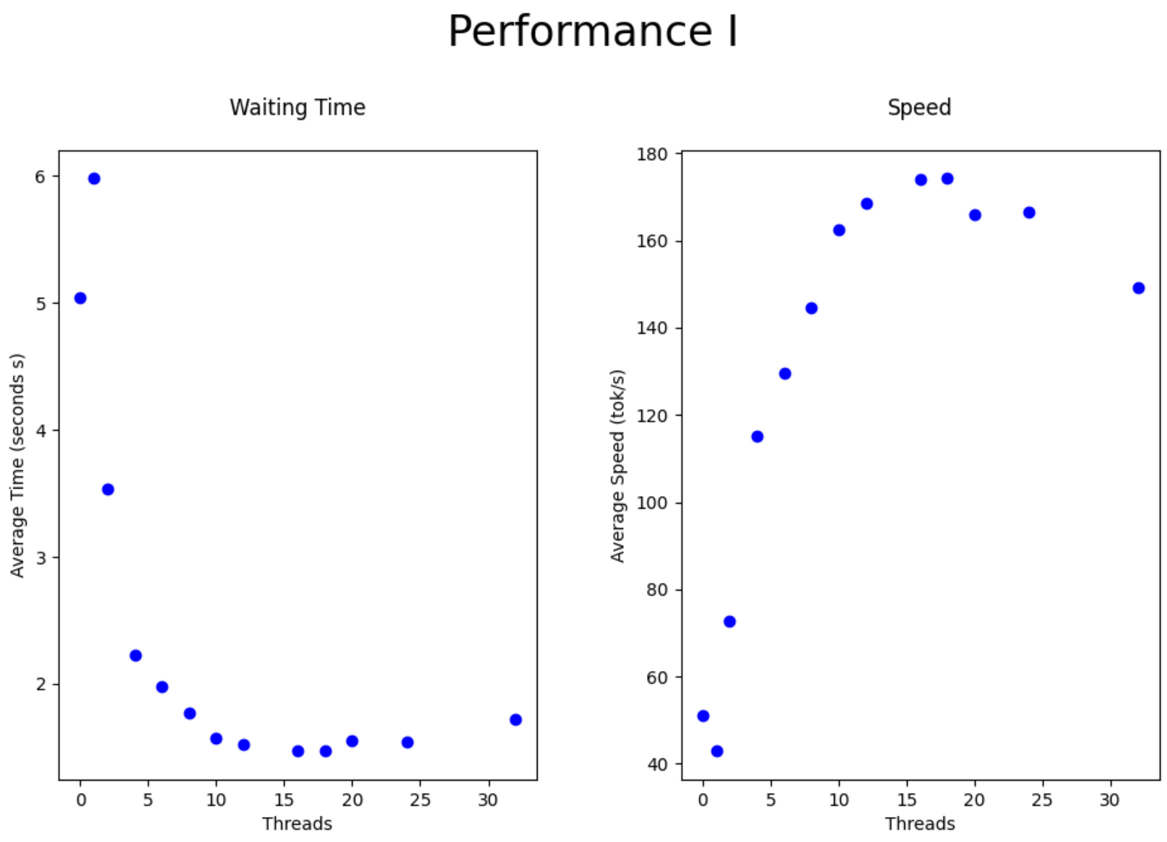

Fig 1: The average time spent waiting for the LLM output to finish (left) and the average token speed (right).

We also notice that both the user time and system time both rise as the number of threads increase. There does appear to be a surprising jump in the usr/sys time after 12 threads and maybe more testing in the future can help answer this anomaly but the general rise in these times can be easily explained by the rise in number of threads. More points of execution mean more time spent on the CPU both in user and kernel (or system) mode especially since we have multiple threads.

Although we should note that the rate of increase in the user and system times is not equivalent as we can see by observing their ratios. System time quadruples by the end while user time only doubles by the end of the table.

A probable explanation for this is that when the number of threads increase total resource utilisation increases and the number of instructions running increases (for example the commands to initialise variables, set things up or wait etc) but increasing the number of threads effects system time significantly more because we are making lots more system calls and likely switching between multiple threads all of which are protected actions that run in kernel mode.

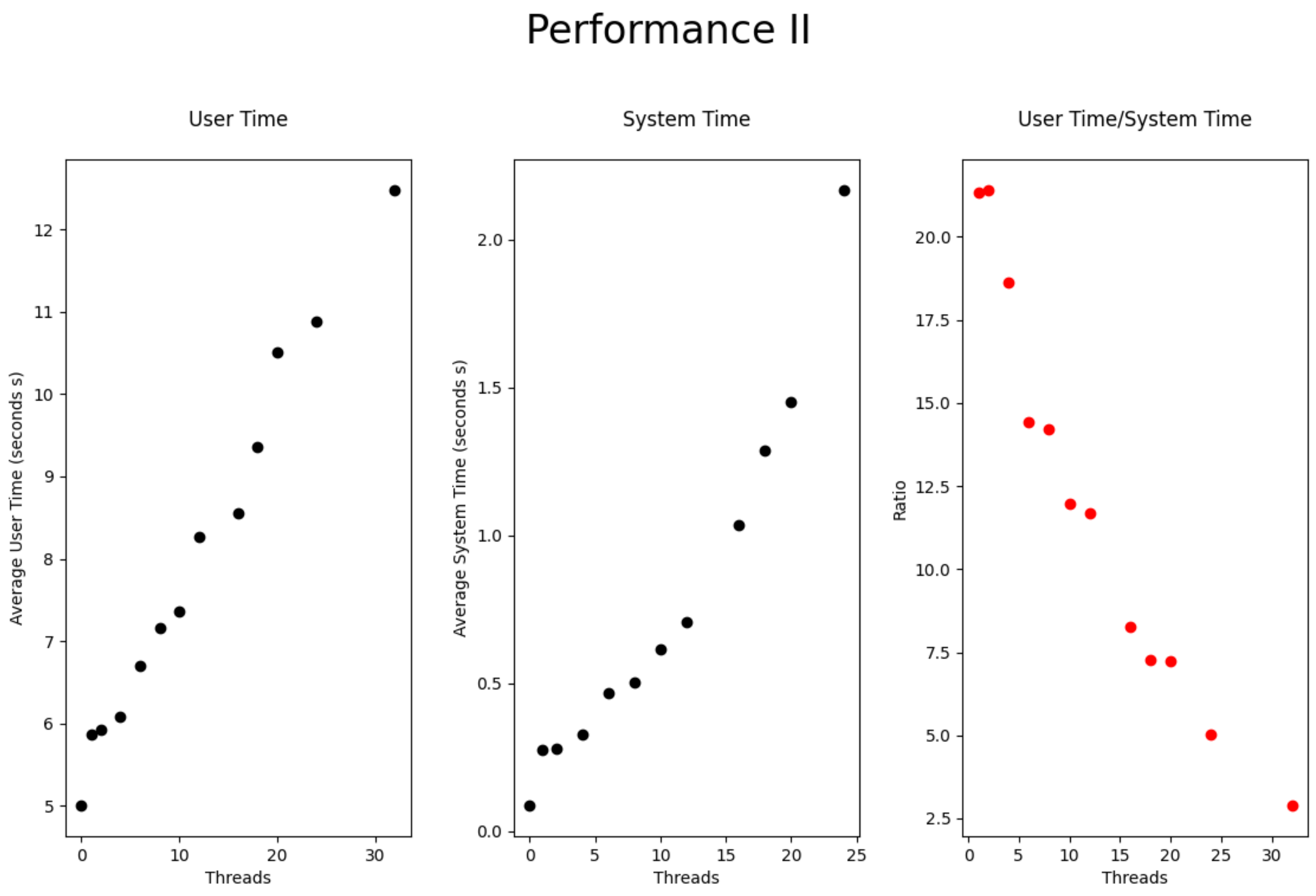

Fig 2: The average user time (left), average system time (center) and their ration (right)