Exploring Open End-to-End Systems-centric Profiling and Benchmarking of Inference-oriented Services

1. Project Description

Machine Learning (ML) is rapidly being integrated and used in computational services. A prominent example is the use of Large Language Models (LLMs) in various applications. Every day, a new use and architecture for their integration is proposed [1]. LLMs and their use indicate why, more generally, we can expect ML model-based computation to be a dominant component of all applications. ML model-based computation can be divided into two phases – training and inference. In the training phase, significant computational resources are used to mine massive amounts of data to construct an artifact we call a model. In the inference phase, the model is "evaluated" by making "queries" with application-specific input data to produce responses utilized within the broader application. Despite rapid industrialization, the core architecture of how models are constructed and represented through training is very much an open area of research. As an example, while "Transformers" [2] are a dominant architecture now, there is growing evidence that other architectures may be equal, if not better, both in functionality and efficiency [3, 4]. One should expect that the details of how models are trained, represented, and evaluated will change. Thus, we should be ready to assess and optimize the ML computational infrastructure in this changing landscape.

As systems researchers and scientists, we must establish methodologies and data that can help ensure that the end-to-end systems constructed using ML model-based computation can be evaluated for traditional system concerns such as scalability, hardware efficiency, reliability, and performance. Despite the ever-evolving nature of ML model-based computation, there is a modular path that can allow systems researchers to help guide the development of future LLM systems meaningfully.

So what can we do?

While there is significant churn in the specifics of the ML architectures, one can observe that all ML model-based approaches must roughly conform to the same external interface and observable functionality for inference. The inference component must provide a request-reply-like interface to the larger application being constructed. From this perspective, we can exploit this stability to apply standard system approaches to creating a framework and methodology for profiling and evaluating different ML inference configurations of software and hardware.

1. Carefully construct a reproducible and auditable environment where the entire service for the proposed ML model-based computation can be deployed. This includes being

precise and transparent concerning all software and hardware being used. Thus, claims that one hardware or software component is superior for inference can be evaluated comparably.
2. Define workloads and performance metrics concerning the end-to-end application that the ML model is used within, enabling fair and meaningful evaluation of the impacts of one ML setup compared to another. For example, in the case of an LLM-based application such as a chatbot, one should not simply evaluate model inference throughput. Instead, one should carefully define end-to-end latency measures that reflect the global impact of the ML software and hardware configuration on the complete chatbot service.
3. Gather detailed data traces that capture the behavior of all hardware and software components.

Proposal

The goal is to construct a testbed and gather data that help inform optimization in LLM systems.

Outcomes

High-level outcomes of the proposed work

1. A data set that documents the behavior of following LLM application on hardware and software combinations:
    a. Different NVIDIA GPUs, running LLMs on single GPU, multiple GPUs, along with sharding a single GPU (using Multi-Instance GPU)
    b. Different CPU processors
    c. Different OSes with different kernel versions.
2. The construction of an end-to-end benchmark in which a stream of application queries can be generated conforming to a particular distribution of query complexity (Eg, context length) and query arrival times. Accordingly, benchmark performance will be expressed as an achieved response latency distribution that accounts for query complexity. Current approaches to LLM evaluation report latency to the first response token and inter-response token latency. These measures are flawed as they do not account for input context length and output length dependency, nor do they reflect the total service time (e.g., I/O, pre-and post-processing).
3. Exploit detailed nature of trace data to identify possible avenues for optimization in the OS and LLM system
4. A systems model for evaluating ML model-based applications beyond the LLM ones concretely evaluated.

Benchmark

Our initial benchmark set consist of running the following Cartesian product:
{models} x {batch size} x {workloads} x {LLM-runtimes}

with:
{models} = {OPT-13B/66B/175B, LLaMA 3.1 8B/70B/405B}

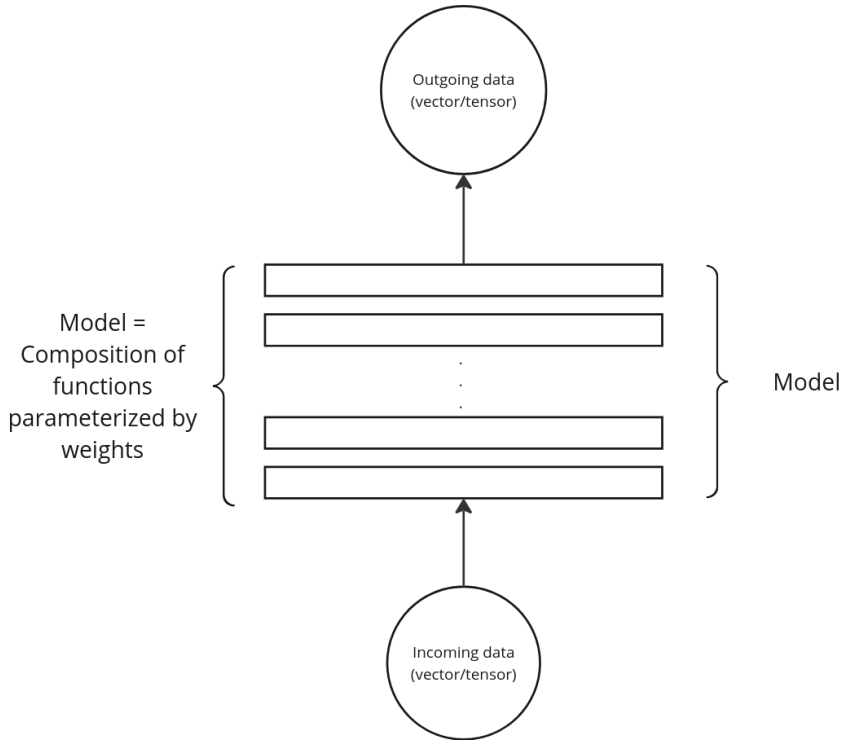
{batch size} = {1, 2, 4, 8, 16, 32, 64}
{workloads} = {ShareGPT, Alpaca}
{LLM-runtimes} = {vLLM, llama.cpp, vanilla pytorch, NVIDIA Triton}

This can be expanded to explore this in the context of other GPUs and architectures from previous generations and other specialized models for use cases such as SegmentAnything 2 (image segmentation), OpenSora (video generation), Sapiens (human oriented vision tasks e.g. depth estimation, pose estimation).

Metrics and Logging
In contrast to other works [5,7,8,9] that typically focus on documenting the end-to-end performance of the LLM systems, we intend to conduct a rigorous analysis and understanding of both OS software and hardware. To achieve this, we intend to utilize tools such as NVIDIA Nsight Systems and NVIDIA Nsight Compute to accurately profile both the LLM systems from both the OS perspective and the GPU processing perspective in order to gather a set of detailed traces of systems behavior. We also intend to gather classical OS metrics such as caches, memory, paging, and other hardware components such as the NICs, SSDs, HDDs in order to provide a holistic view of how GPU runs with the different models, workloads, LLM-runtimes and its interactions with the host OS and its components.

Automation and Data Curation
We intend to document and open-source all of the tooling, scripts, and data for this endeavor so that other users can build upon our work to better evaluate and understand new LLM training and inferencing technologies as well as any future hardware and software advancements.

2. Interesting Result on A100



The forward pass stage of inferencing typically consists of very SIMD-friendly operations across multiple large vectors that are copied back and forth to the GPU. We began with a small microbenchmark that used a simple vector addition to understand and explore the use of 4 KB and 2 MB page sizes and how it impacts the hostToDevice and deviceToHost CUDA memcpy time. As the vector memory is typically backed by pages in the host OS, we were curious what the benefits were to reducing extraneous page faults through explicit use of larger page sizes.

Example vector addition:

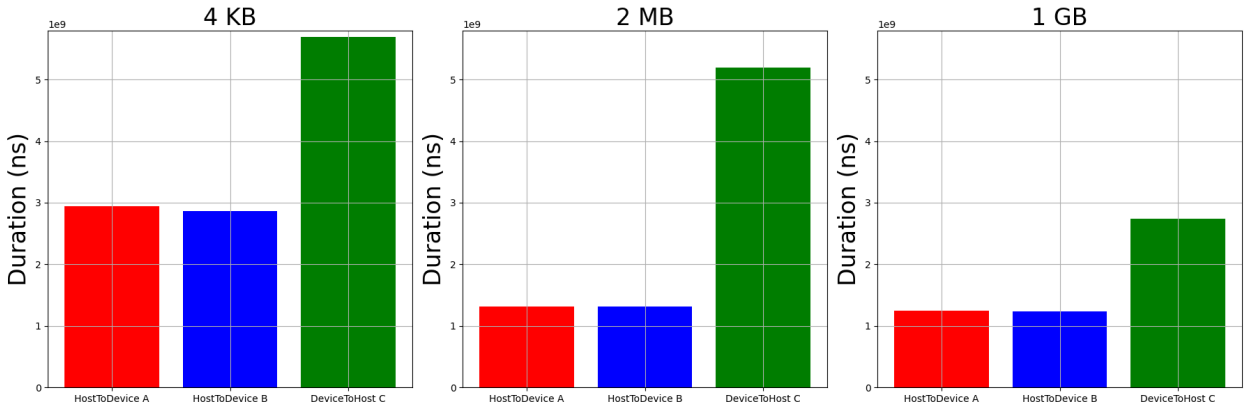
```
__global__ void vectorAdd(const double *A, const double *B, double *C, unsigned long long int numElements) {
  unsigned long long int i = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < numElements) {
    C[i] = A[i] + B[i] + 0.0f;
  }
}
```

Example memory mapped vectors with different page sizes:

```
#ifdef FOURKB
  double *h_A = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  double *h_B = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  double *h_C = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
#elif TWOMB
  double *h_A = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB | MAP_HUGE_2MB, -1, 0);
```

```
  double *h_B = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS |
MAP_HUGETLB | MAP_HUGE_2MB, -1, 0);
  double *h_C = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS |
MAP_HUGETLB | MAP_HUGE_2MB, -1, 0);
#elif ONEGB
  double *h_A = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS |
MAP_HUGETLB | MAP_HUGE_1GB, -1, 0);
  double *h_B = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS |
MAP_HUGETLB | MAP_HUGE_1GB, -1, 0);
  double *h_C = (double *) mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS |
MAP_HUGETLB | MAP_HUGE_1GB, -1, 0);
#endif
```

Results:



We found a 1.7X - 2X CUDA memcpy speed difference between 4 KB and 2 MB pages when coping from the Host to the GPU device. Interestingly, there is also a dramatic difference when copying from GPU device to Host when moving to a 1 GB page size. However, a caveat is that Linux by default uses 2 MB pages with Transparent Hugepages support, so the 4 KB result is mostly unrealistic. However, there is still something interesting and requires more investigation with the 2 MB to 1 GB setup given the difference in device to host copying time.

Bibliography

[1] https://www.assemblyai.com/blog/llm-use-cases/
[2] https://jalammar.github.io/illustrated-transformer/
[3] https://arxiv.org/abs/2010.11929
[4] https://lilianweng.github.io/posts/2023-01-27-the-transformer-family-v2/
[5] Efficient Memory Management for Large Language Model Serving with PagedAttention
[6] https://arxiv.org/pdf/2407.07000
[7] https://www.redhat.com/en/blog/evaluating-llm-inference-performance-red-hat-openshift-ai
[8] https://www.confident-ai.com/blog/evaluating-llm-systems-metrics-benchmarks-and-best-practices