Samsung Computer Engineering Challenge 2023
Round 1 Report

Our LLaMA inference on entire HellaSwag takes 480.1 seconds.

1 Team Information

Team name	Team H	
Members	Heehoon Kim	Junyeol Ryu
Affiliation	Seoul National University	Seoul National University
Email	csehydrogen@gmail.com	jyeol.ryu@gmail.com
Phone	010-2569-3426	010-3329-7561

2 Problem

2.1 Background

Model. LLaMA [4] is a Transformer-based foundation language model trained and released by Meta. Its architecture is a sequence of Transformer blocks, preceded by one input embedding layer and followed by one output embedding layer. Each Transformer block receives an input tensor, performs computation flows through a multi-head attention (MHA) module and a feed-forward (FFD) module, and transmits an output tensor of equivalent size to the next block. Analogous to other Transformer-based models (e.g., GPT-3 [5]), LLaMA inference can utilize KV cache [3] in the MHA module to store key and value tensors associated with the attention mechanism.
Task. HellaSwag [6] is a multiple choice task where each data consists of a context and multiple endings. To find the most likely ending following the context, LLaMA selects the ending with the highest conditional probability. Specifically, let C, l_C be the context and its length, and E_i, l_{E_i} be the i-th ending and its length. A concatenated tensor $(C \parallel E_i)$ of context and one of the endings is fed to LLaMA, returning an output tensor O of size $(l_C + l_{E_i}, V)$. From this, the likelihood of E_i with respect to C can be calculated as the following:

$$P(E_i|C) = \prod_{i=0}^{l_{E_i}-1} O[i + l_C - 1, E_i[i]]. \tag{1}$$

For better accuracy, LLaMA follows Gao et al. [2] where they use the likelihood normalized by the number of characters in the ending, i.e., $\sqrt[n]{P(E_i|C)}$, where n is the number of characters in the ending before tokenization.[1]

2.2 Challenges

Varying input sizes. As the lengths of contexts and endings differ largely among the HellaSwag dataset, a naive batching approach can be to generate a rectangular input batch tensor by padding

[1]We use log probability $logP(E_i|C) = \sum_{i=0}^{l_{E_i}-1} logO[i + l_C - 1, E_i[i]]$ or normalized log probability $\frac{logP(E_i|C)}{n}$ for better numerical stability, but neither affects the correctness.

the inputs to the maximum input length in the batch. However, this results in redundant computations for the padding and reduces effective computations per time unit. Another batching approach can be to batch inputs of the same length. However, this approach would produce input tensors with small batch sizes insufficient to fully utilize GPU. Therefore, a batching schedule with high computation efficiency is critical.

Model parallelism. As large language models cannot fit in a single GPU memory, a model parallelism technique is required to partition the model onto multiple GPUs. However, tensor parallelism (TP) introduces communication overhead due to frequent communication between partitions while pipeline parallelism (PP) requires careful balancing of stages and workloads to eliminate bubbles. Thus, a model partition scheme and corresponding refinements are crucial.

3 Our Solution

3.1 Computation Analysis

We use theoretical FLOPs (FLoating point OPerations) as a metric for the amount of workload. The computation requirement of the MHA module consists of four matrix multiplications, where each has FLOPs of $6SBH^2$, $2BSCH$, $2BSCH$, and $2BSH^2$. H is a hidden dimension where the LLaMA 33B model uses $H = 6656$. C is the sum of S and the length of tokens in the KV cache. FFD module consists of three matrix multiplications, where each has FLOPS of $2BSHD$. D is the second hidden dimension where the model uses $D = 17920$. At the end of the model, there is an output embedding layer that has FLOPs of $2BSHV$. Other operations have negligible FLOPs compared to the matrix multiplications, so we omit them in the analysis. Thus, the total FLOPs of the model is given as:

$$F = L(8BSH^2 + 4BSCH + 6BSHD) + 2BSHV \tag{2}$$

where L is the number of Transformer blocks. In HellaSwag, the maximum length is only 170, so S and C are much smaller than H and D. Thus, the $BSCH$ term is negligible and F is nearly proportional to the $B \times S$, i.e., $F \propto B \times S$. **This implies that we can use the number of tokens as a metric to estimate workload, without any complex analytic model.**

3.2 Batch Scheduling

Minimizing Waste of Computation. Based on the analysis, we find an efficient batching strategy for the entire input dataset that minimizes the padded area using Dynamic programming.

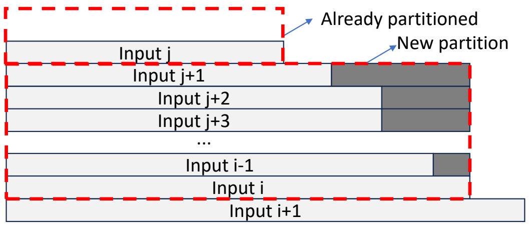


Figure 1: Visualization of a single step in Dynamic programming. The grey area is the padded tokens.

First, we sort the inputs by the number of tokens. We define $D[i]$ as the minimum area of padding after we partition until i-th input. Then, $D[i]$ can be computed as follows:

$$D[i] = \min\{D[j] + l_i \times (i - j) - \sum_{k=j+1}^{i} l_k \mid j \in [0, i), l_i \times (i - j) \geq \theta\}, \tag{3}$$

where l_i is the number of tokens of i-th input, and θ is the threshold to prevent GPU underutilization. The term $l_i \times (i - j)$ describes $B \times S$ of the current partition and the term $\sum_{k=j+1}^{i} l_k$ is the area filled with actual tokens. Thus, the subtraction of two terms equals the padded area. Figure 1 visualizes the single step in Dynamic programming.

Removing Redundant Computation. As HellaSwag is a multiple-choice task, four inputs share the same context. By storing the key and value vectors in the cache, we can reduce the computation for the contexts by a factor of four.

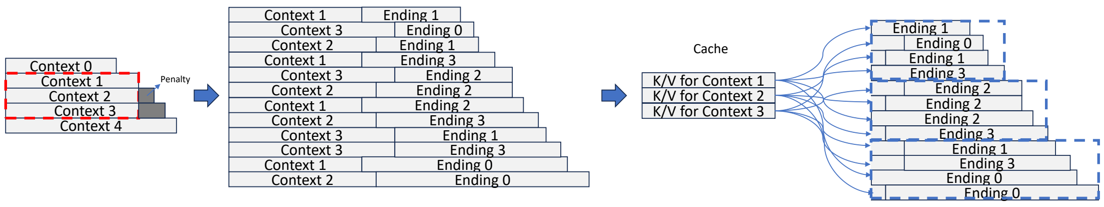


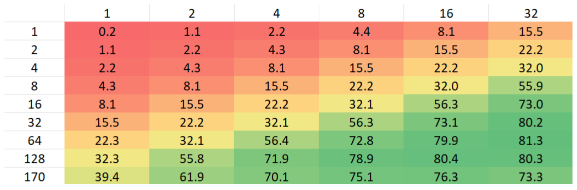
Figure 2: Example of removing redundant computation.

Figure 2 depicts the overall process. First, we sort the dataset by the length of context tokens. Because the layer implementations do not support a jagged array, we have to use the same length for all key/value vectors in a single batch. This means we should construct a batch with the minimum length of the contexts. Thus, we use a slightly different recursive formula from Equation 3 to partition the contexts:

$$D[i] = \min\{D[j] + \sum_{k=j+1}^{i} l_k - l_{j+1} \times (i-j) \mid j \in [0,i), l_{j+1} \times (i-j) \geq \theta_{ctx}\}, \tag{4}$$

which penalizes the area beyond the minimum length as the area translates to redundant computation. After partitioning the context, we sort the inputs inside the partition by the length of full inputs; contexts followed by the endings. Then, the partitions are partitioned once more using Equation 3.

Deciding the threshold. It is important to give proper values to θ and θ_{ctx}. We execute the single Transformer block with various B and S and measure the achieved FLOPS. As you see in Figure 3, the Transformer block converges to around 80 TFLOPS beyond the line $B \times S \sim 1024$. (Note that the theoretical peak FLOPS of V100 is 112 TFLOPS.) Thus, we gave $\theta = 2048$ to keep GPU utilization high. For context partitioning, it is important to keep the batch size high enough because the small batch size may result in too few inputs inside the partition, forcing the large padding in the second partitioning step. Thus, we gave $\theta_{ctx} = 10000$, which is approximately the largest value without Out-Of-Memory error due to the size of the cache.



	1	2	4	8	16	32
1	0.2	1.1	2.2	4.4	8.1	15.5
2	1.1	2.2	4.3	8.1	15.5	22.2
4	2.2	4.3	8.1	15.5	22.2	32.0
8	4.3	8.1	15.5	22.2	32.0	55.9
16	8.1	15.5	22.2	32.1	56.3	73.0
32	15.5	22.2	32.1	56.3	73.1	80.2
64	22.3	32.1	56.4	72.8	79.9	81.3
128	32.3	55.8	71.9	78.9	80.4	80.3
170	39.4	61.9	70.1	75.1	76.3	73.3

Figure 3: Achieved FLOPS for different B and S. The values in the first column is S and the values in the first row is B.

3.3 Pipeline Parallelism and PREFILL Mini-Batching

We use pipeline parallelism since tensor parallelism may yield low inference latency but its frequent communication tends to yield lower throughput. We equally assign 15 Transformer blocks to each GPU. Layers before the first Transformer block are assigned to the first GPU, and the layers after the last Transformer block are assigned to the last GPU. This causes the last GPU slightly lags behind the other GPUs due to the output embedding layer but the imbalance is negligible. The key to the pipeline mechanism is to balance its stages. As the batch size of `PREFILL` stage is larger than that of `DECODE` stage, the `PREFILL` computation delays the entire pipeline as well as takes a longer period to fill the pipeline. Hence, we partition the `PREFILL` batch into fine-grained mini-batches such that the input tensor size of the stage is similar to that of `DECODE` stage, balancing the stage workloads and eliminating the stall.

4 Implementation

4.1 Local Machine Environment

Table 1: System configuration.

OS	Ubuntu 20.04 64-bit
CPU	AMD EPYC 7452 32-Core Processor
Storage	Seagate FireCuda 520 SSD 2TB
RAM	512GB
GPU	4 x NVIDIA Tesla V100 32GB (PCIe)

Table 1 describes the system configuration we used. We use the same type and number of GPUs with the target system in Round 2, except for the lack of `NVLink`. We expect that the results of our system will be reproduced in the target system, as the communication through PCIe links was not a bottleneck in our implementation.

4.2 Open Sources and Libraries Used

We used `PyTorch` 2.0.1 as a deep learning framework and `datasets` 2.14.5 to load HellaSwag. We used the optimized implementation of RMS normalization, `FusedRMSNorm`, from `apex` 23.08.

4.3 Core Implementation Content

Our implementation consists of approximately 3K LOCs of Python, CUDA, and C++. Code is uploaded at `https://github.com/csehydrogen/SCEC2023-TeamH` (branch: SCEC-submit).

We first repartition pretrained LLaMA weights along the vertical direction (i.e., along the layers, not inside a layer) to support pipeline parallelism. We provide a script `repartition_ckpt.py` that preprocesses LLaMA weights (`consolidated.*.pth`) into our format(`30B_cpu_*.pth`). Please see `Setup` section of our `README` for detailed instructions.

To overlap communication and computation, all communication APIs are called with a non-blocking option. Specifically, we used `isend` and `irecv` instead of `send` and `recv` for GPU-to-GPU communication. For communication between the CPU and GPUs, we used a pinned buffer and issued a copy kernel on a separate CUDA stream.

We made minor modifications to the model implementation. As we do not use tensor parallelism, we replaced all parallelized layers from `fairscale` with vanilla `PyTorch` layers. We used `scaled_dot_product_attention` layer from `PyTorch` which is more memory efficient than the MHA module with multiple layers. We also removed the fixed-size cache and made the model return the key-value vectors along the output tensor. This enables more dynamic allocation and deallocation of key-value vectors, which leads to less memory consumption.

5 Local Evaluation Results

To correctly measure the performance, we put a synchronization barrier after the initialization phase, record the start time, and record the end time after all GPUs finish their jobs. The initialization phase takes about 20 seconds and includes loading the dataset, loading the checkpoint on GPUs, and loading the tokenizer. As scheduling the whole dataset with Dynamic programming just takes a few seconds, we put the scheduling procedure in the initialization phase.

Table 2: Performance in various implementations.

ID	Parallelism	Batch scheduling		KV cache	PREFILL Mini-Batching	Activity label	Time(s)	Accuracy
		PREFILL	DECODE					
A	TP	Fixed(32)		X	X	O	2840.4	82.65
B	PP	Fixed(32)		X	X	O	1121.6	82.65
C	PP	Dynamic		X	X	O	849.9	82.65
D	PP	Fixed(128)	Fixed(32)	O	X	O	518.9	82.65
E	PP	Dynamic	Fixed(32)	O	X	O	511.1	82.65
F	PP	Dynamic	Dynamic	O	X	O	507.9	82.65
G	PP	Dynamic	Dynamic	O	X	X	492.5	81.68
H	**PP**	**Dynamic**	**Dynamic**	**O**	**O**	**O**	**480.1**	**82.65**

Table 2 shows the impact of each optimization. Experiment A is a minimal TP example provided by Meta [1] and is the performance baseline. Experiment B is a minimal PP example that processes the whole dataset sequentially without any batching strategy. Experiment C leverages batch scheduling obtained by Dynamic programming so it uses dynamic batch sizes. Note that Experiments A, B, and C do not utilize KV cache so they have a single batch scheduling.

Experiments D, E, F, G, and H add KV cache to remove redundant computation on the contexts. Compared to Experiment F, Experiments D and E show the inference time increases when the fixed batch sizes are used on either contexts or endings. While other experiments prepend the activity label to the context following Gao et al. [2], Experiment G uses the context without any preprocessing. As a result, it shows a 3% speedup in exchange for a 1% accuracy drop compared to Experiment F. Our best implementation, Experiment H, further adds PREFILL mini-batching. It takes **480.1 seconds** and improves the baseline inference time by **5.92**× while preserving the reported accuracy close to 82.8% reported in LLaMA paper [4].

References

[1] LLaMA. https://github.com/facebookresearch/llama/tree/llama_v1.

[2] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation. *Version v0.0.1. Sept*, 2021.

[3] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[4] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[5] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[6] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.