CSCI 596 Final Proposal: FlashAttention and LLaMA for In-context Learning with Document Retrieval

Shicheng Wen
Thomas Lord Department of Computer Science
University of Southern California
`wenshich@usc.edu`

Abstract

This is a final project for the course CSCI 596 Scientific Computing and Visualization at USC. The project includes an exploration of the FlashAttention mechanism, and a proposal on equipping the LLaMA model with FlashAttention for In-context learning with document retrieval. The research aims to make large language models capable of local machines such that general users can deploy the model locally for conversations and text generation with document retrieval.

1 Introduction

The rise of chatbots powered by large language models, such as ChatGPT, has shown a remarkable ability to generate human-like conversations with users, especially in answering questions. However, these large closed-sourced models are lack flexibility and customization options, resulting in users not able to generate controllable text without costing a huge amount of time for prompt engineering. Moreover, even some open-source LLMs, such as LLaMA (Touvron et al., 2023), are available for people to deploy on their own machine, but training or fine-tuning such models is still impossible for general users with low GPU resources. In this proposal, we aimed to apply FlashAttention (Dao et al., 2022) to the LLaMA model to train an in-context learning model, such that it may cost much lower computational resources for training, and may be able to deploy locally on a consumer GPU. Moreover, we aim to adapt FlashAttention to GPU in Turing architecture so that old GPUs, including NVIDIA RTX 2080 and NVIDIA Tesla T4, may also be able to deploy locally.

2 Background and Problem Description

2.1 GPU Architecture

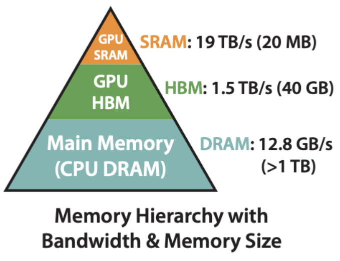

Figure 1

As shown in Fig. 1, the hierarchical structure of GPU memory is characterized by a diversity of memory types, varying in size and speed. Notably, smaller memory units typically exhibit higher speeds. For instance, the A100 GPU is equipped with 40-80 GB of high-bandwidth memory (HBM), offering a bandwidth of 1.5-2.0 TB/s. Additionally, it includes 192 KB of on-chip SRAM for each of its 108 streaming multiprocessors, which boasts an estimated bandwidth of approximately 19 TB/s (Abdelkhalik et al. 2022, Jia et al. 2018). This on-chip SRAM is significantly faster than HBM, yet it is substantially smaller in capacity. The efficiency of operations is frequently limited by the rate of memory (particularly HBM) access. Consequently, strategically using the faster SRAM is becoming increasingly crucial to optimize performance.

2.2 Self Attention

The core technology of large language models based on the Transformer(Vaswani et al., 2017) architecture uses self-attention. Fig. 2 is a recap of self-attention architecture. Given input sequences $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times N}$, where N is the sequence length, and d is the head dimension, the self-attention output $\mathbf{O}$ is calculated as:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N\times N},$$
$$\mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N},$$
$$\mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d}$$

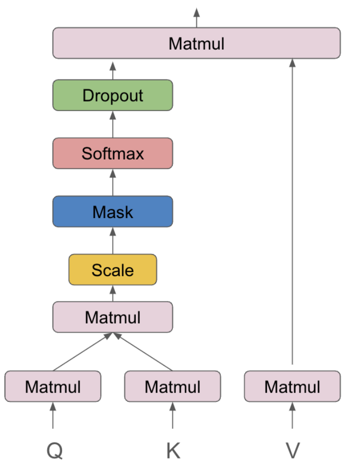

Figure 2: Self-Attention Mechanism

It is noticeable that each step in the self-attention calculation requires GPU to read matrices from HBM to SRAM, do the calculation, and write the output back to HBM. As the intermediate matrices are very large, these actions require a huge amount of transactions between HBM and SRAM, resulting in large computational costs and a long training time.

2.3 In-Context Learning

In-context learning has emerged as a novel attribute of large language models, enabling them to execute a variety of tasks based on a set of input-output examples, without the need for any parameter updates or fine-tuning. This feature has been demonstrated in major language models such as ChatGPT and LLaMA, garnering widespread attention in the research community. One area of study focuses on understanding the fundamental mechanisms and principles underlying contextual learning. For example, Xie et al. (2022) view contextual learning as implicit Bayesian inference, while Dai et al. (2023) interpret it as meta-optimization. Another area of research explores different strategies for selecting and designing contextual examples for large language models. Wang et al. (2023) proposed a novel training approach to model the interactions between contextual examples, determinantal processes and sequential decision-making are introduced as a preliminary exploration. In contrast, structured prompts break the limitation of input context length and expand the number of contextual examples to several thousands. Retrieval Augmented large language models combine the generative capabilities of large language models with the ability to retrieve relevant information from external sources. This paradigm has the potential to enhance the factual consistency of generated texts, enable large language models to understand up-to-date knowledge and provide a natural way for source attribution (Xie et al., 2022). For contextual learning, the goal of retrieval augmentation is to improve the performance of large language models on downstream tasks by enriching them with information-rich examples retrieved.

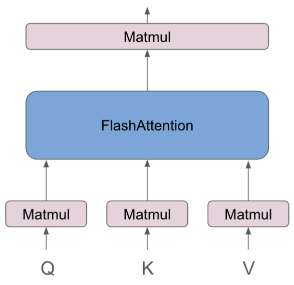

Figure 3: Enter Caption

2.4 LLaMA

LLaMA (Touvron et al., 2023) is a transformer-based, open-sourced large language model. It has four versions, including 7B, 13B, 33B, and 65B. LLaMA-13B outperforms GPT-3 (175B) (Brown et al., 2020) on most benchmarks, and LLaMA-65B is competitive with the state-of-the-art models Chinchilla-70B (Hoffmann et al., 2022) and PaLM-540B (Chowdhery et al., 2022). With Pre-normalization (Zhang and Sennrich, 2019), SwiGLU activation function (Shazeer, 2020), and Rotary Embeddings (Su et al., 2023), LLaMA is trained on various datasets and performs as state-of-the-art in different tasks. More importantly, LLama supports float point 16 precision that reduces the weight and computational cost without losing performance. It is one of the best choices to deploy LLMs locally.

3 Methodology

FlashAttention (Dao et al., 2022) is a novel way of calculating self-attention, that requires fewer HBM read/write and without strong large intermediate matrices for backward pass. As shown in Fig. 3, FlashAttention will calculate the attention output **O** at once in SRAM without repeated reading and writing HBM. FlashAttention involves two major techniques: tilling and recomputation.

3.1 Tilling

FlashAttention calculates the softmax by blocks one at a time and combines the results after computation. The decomposed softmax is calculated as follows:

$$m(x) = m\left(\left[x^{(1)}x^{(2)}\right]\right) = \max\left(m\left(x^{(1)}\right), m\left(x^{(2)}\right)\right)$$
$$f(x) = \left[\ e^{m\left(x^{(1)}\right)-m(x)}f\left(x^{(1)}\right), e^{m\left(x^{(2)}\right)-m(x)}f\left(x^{(2)}\right)\ \right]$$
$$\begin{aligned}\ell(x) &= \ell\left(\left[x^{(1)}x^{(2)}\right]\right)\\ &= e^{m\left(x^{(1)}\right)-m(x)}\ell\left(x^{(1)}\right) + e^{m\left(x^{(2)}\right)-m(x)}\ell\left(x^{(2)}\right)\end{aligned}$$
$$\text{softmax}(x) = \frac{f(x)}{\ell(x)}$$

See Algorithm 1 for a detailed forward pass algorithm.

3.2 Recomputation

As the goal of FlashAttention is not to store the large intermediate matrix $\mathbf{S}, \mathbf{P} \in \mathbb{R}^{N \times N}$ that cost $O(N^2)$ memory, it is a challenge to compute the gradients of **S** and **P** with respect to **Q**, **K**, and **V**. However, FlashAttention will store the output **O** and the softmax normalization statistics (m, l) that could recompute **S** and **P** easily in SRAM from blocks of **Q**, **K**, and **V**. This can be seen as a form of selective gradient checkpointing. While gradient checkpointing has been suggested to reduce the maximum amount of memory required, all implementations have to trade speed for memory. In contrast, even with more FLOPs, the recomputation method speeds up the backward pass due to reduced HBM accesses(Dao et al., 2022). See Algorithm 2 for a detailed backward pass algorithm.

3.3 Document Retrieval

LLM-R (**LLM R**etriever), proposed by Wang et al. (2023), is a novel framework, that aims to retrieve high-quality in-context examples for large language models. Given an initial set of retrieved candidates, our framework ranks them based on the conditional LLM log probabilities of the ground-truth outputs. Subsequently, a cross-encoder-based reward model is trained to capture the fine-grained ranking signals from LLMs. Finally, a bi-encoder-based dense retriever is trained using knowledge distillation. The training process of LLM-R comprises three stages: generating training data based on an initial retriever and LLM feedback, reward modeling, and training dense retrievers by distilling the knowledge from the reward model. At inference time, the trained dense retriever is employed to retrieve in-context examples from the pool $\mathbb{P}$ and the retrieved examples are fed to the LLM to generate the output.

4 Experiment

4.1 Applying FlashAttention to LLaMA

We aim to replace all multi-head attention layers in the original LLaMA model without changing any parameters. This involves rewriting the forward and backward functions and adapting the FlashAttention CUDA file to fit the sequence length, dimension, and head size of LLaMA. The process may also involve modifications to FlashAttention arguments such that it is able to support Turing architecture GPUs.

4.2 Equip Model with LLM-R

As instructed by Wang et al. (2023), LLaMA-7B is the base model used for candidate ranking and task evaluation. We aim to equip LLaMA with LLM-R and redo the three-phase training step for document retrieval training and evaluation.

4.3 Experiment Setup

We will train and evaluate our model on one NVIDIA RTX 2080 GPU so that it can be deployed on local machines in the future. The base model we are using is LLaMA-7B, the reward model is ELECTRA-base (Clark et al., 2020), and the retriever is E5-base (Wang et al., 2022).

5 Expected Result

The experiment is expected to make sure LLaMA-7B, in the precision of float 16, is able to deploy, train, and evaluate small VRAM GPUs, including Turing GPUs such as NVIDIA RTX 2080 and NVIDIA Tesla T4. The model is expected to achieve comparable performance to original LLaMA-7B with float 32 precision and normal multi-head attention on tasks including Close QA, Commonsense Reasoning, Coreferencing, NLI, Paraphrasing, Reading Comprehension, Sentiment Analysis, Data-to-text, and Summarization.

References

Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed Badawy. 2022. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind

Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators.

Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. 2023. Why can GPT learn in-context? language models secretly perform gradient descent as meta-optimizers. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 4005–4019, Toronto, Canada. Association for Computational Linguistics.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training compute-optimal large language models.

Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the nvidia volta gpu architecture via microbenchmarking.

Noam Shazeer. 2020. Glu variants improve transformer.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2023. Roformer: Enhanced transformer with rotary position embedding.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text embeddings by weakly-supervised contrastive pre-training.

Liang Wang, Nan Yang, and Furu Wei. 2023. Learning to retrieve in-context examples for large language models.

Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2022. An explanation of in-context learning as implicit bayesian inference.

Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization.

Algorithm 1 FlashAttention Forward Pass

```
Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$, softmax scaling constant
    $\tau \in \mathbb{R}$, masking function MASK, dropout probability $p_{\text{drop}}$.
 1: Initialize the pseudo-random number generator state $\mathcal{R}$ and save to HBM.
 2: Set block sizes $B_c = \left\lfloor \frac{M}{4d} \right\rfloor, B_r = \min\left(\left\lfloor \frac{M}{4d} \right\rfloor, N\right)$.
 3: Initialize $O = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
 4: Divide $Q$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $Q_1, \dots, Q_{T_r}$, of size $B_r \times d$ each, and divide $K, V$ into $T_c = \left\lceil \frac{N}{B_c} \right\rceil$
    blocks $K_1, \dots, K_{T_c}$ and $V_1, \dots, V_{T_c}$, of size $B_c \times d$ each.
 5: Divide $O$ into $T_r$ blocks $O_1, \dots, O_{T_r}$, of size $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_1, \dots, \ell_{T_r}$, of
    size $B_r$ each, divide $m$ into $T_r$ blocks $m_1, \dots, m_{T_r}$, of size $B_r$ each.
 6: for $j = 1$ to $T_c$ do
 7:     Load $K_j, V_j$ from HBM to on-chip SRAM.
 8:     for $i = 1$ to $T_r$ do
 9:         Load $Q_i, O_i, \ell_i, m_i$ from HBM to on-chip SRAM.
10:         On chip, compute $S_{ij} = \tau Q_i K_j^T$ in $\mathbb{R}^{B_r \times B_c}$.
11:         On chip, compute $S_{ij}^{\text{masked}} = \text{MASK}(S_{ij})$.
12:         On chip, compute $m_{ij} = \text{rowmax}(S_{ij}^{\text{masked}})$ in $\mathbb{R}^{B_r}$, $P_{ij} = \exp(S_{ij}^{\text{masked}} - m_{ij})$ in $\mathbb{R}^{B_r \times B_c}$
    (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(P_{ij})$ in $\mathbb{R}^{B_r}$.
13:         On chip, compute $m_{\text{new}} = \max(m_i, m_{ij})$ in $\mathbb{R}^{B_r}$, $c_{\text{new}} = e^{m_i - m_{\text{new}}} \ell_i + e^{m_{ij} - m_{\text{new}}} \tilde{\ell}_{ij}$ in $\mathbb{R}^{B_r}$.
14:         On chip, compute $\hat{P}_{ij}^{\text{dropped}} = \text{dropout}(P_{ij}, p_{\text{drop}})$.
15:         Write $O_i \leftarrow O_i + \left(\text{diag}\left(c_{\text{new}}^{-1}\right) \text{diag}\left(e^{m_i - m_{\text{new}}}\right) \hat{P}_{ij}^{\text{dropped}} V_j\right)$ to HBM.
16:         Write $\ell_i \leftarrow c_{\text{new}}, m_i \leftarrow m_{\text{new}}$ to HBM.
17:     end for
18: end for
19: return $O, \ell, m, \mathcal{R}$.
```

Algorithm 2 FlashAttention Backward Pass

```
Require: Matrices $Q, K, V, O, dO \in \mathbb{R}^{N \times d}$ in HBM, vectors $\ell, m \in \mathbb{R}^N$ in HBM, on-chip SRAM of size $M$, softmax scaling constant $\tau \in \mathbb{R}$, masking function MASK, dropout probability $p_{\text{drop}}$, pseudo-random number generator state $\mathcal{R}$ from the forward pass.
 1: Set the pseudo-random number generator state to $\mathcal{R}$.
 2: Set block sizes $B_c = \left\lfloor \frac{M}{4d} \right\rfloor, B_r = \min\left(\left\lfloor \frac{M}{4d} \right\rfloor, N\right)$.
 3: Divide $Q$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $Q_1, \dots, Q_{T_r}$, of size $B_r \times d$ each, and divide $K, V$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $K_1, \dots, K_{T_c}$ and $V_1, \dots, V_{T_c}$, of size $B_c \times d$ each.
 4: Divide $O$ into $T_r$ blocks $O_1, \dots, O_{T_r}$, of size $B_r \times d$ each, divide $dO$ into $T_r$ blocks $dO_1, \dots, dO_{T_r}$, of size $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_1, \dots, \ell_{T_r}$, of size $B_r$ each, divide $m$ into $T_r$ blocks $m_1, \dots, m_{T_r}$, of size $B_r$ each.
 5: Initialize $dQ = (0)_{N \times d}$ in HBM and divide it into $T_r$ blocks $dQ_1, \dots, dQ_{T_r}$, of size $B_r \times d$ each. Initialize $dK = (0)_{N \times d}, dV = (0)_{N \times d}$ in HBM and divide $dK, dV$ into $T_c$ blocks $dK_1, \dots, dK_{T_c}$, and $dV_1, \dots, dV_{T_c}$, of size $B_c \times d$ each.
 6: for $j = 1$ to $T_c$ do
 7:     Load $K_j, V_j$ from HBM to on-chip SRAM.
 8:     Initialize $dK_j = (0)_{B_c \times d}, dV_j = (0)_{B_c \times d}$ on SRAM.
 9:     for $i = 1$ to $T_r$ do
10:         Load $Q_i, O_i, dO_i, \ell_i, m_i$ from HBM to on-chip SRAM.
11:         On chip, compute $S_{ij} = \tau Q_i K_j^T$ in $\mathbb{R}^{B_r \times B_c}$.
12:         On chip, compute $S_{ij}^{\text{masked}} = \text{MASK}(S_{ij})$.
13:         On chip, compute $P_{ij} = \text{diag}(\ell_i) \exp(S_{ij}^{\text{masked}} - m_i)$ in $\mathbb{R}^{B_r \times B_c}$.
14:         On chip, compute dropout mask $Z_{ij} \in \mathbb{R}^{B_r \times B_c}$ where each entry has value $\frac{1}{1-p_{\text{drop}}}$ with probability $1 - p_{\text{drop}}$ and value 0 with probability $p_{\text{drop}}$.
15:         On chip, compute $P_{ij}^{\text{dropped}} = P_{ij} \odot Z_{ij}$ (pointwise multiply).
16:         On chip, compute $dV_j = dV_j + (P_{ij}^{\text{dropped}})^T dO_i$ in $\mathbb{R}^{B_c \times d}$.
17:         On chip, compute $dP_{ij}^{\text{dropped}} = dO_i V_j^T$ in $\mathbb{R}^{B_r \times B_c}$.
18:         On chip, compute $dP_{ij} = dP_{ij}^{\text{dropped}} \odot Z_{ij}$ (pointwise multiply).
19:         On chip, compute $D_i = \text{rowsum}(dO_i - O_i)$ in $\mathbb{R}^{B_r}$.
20:         On chip, compute $dS_{ij} = P_{ij} \odot (dP_{ij} - D_i)$ in $\mathbb{R}^{B_r \times B_c}$.
21:         Write $dQ_i \leftarrow dQ_i + \tau dS_{ij} K_j$ in $\mathbb{R}^{B_r \times d}$ to HBM.
22:         On chip, compute $dK_j \leftarrow dK_j + \tau dS_{ij}^T Q_i$ in $\mathbb{R}^{B_c \times d}$.
23:     end for
24:     Write $dK_j \leftarrow dK_j, dV_j \leftarrow dV_j$ to HBM.
25: end for
26: return $dQ, dK, dV$.
```