FlashAttention: A CUDA implementation

Dhruv Srikanth
Columbia University
ds4399@columbia.edu

Pranav Kumar Kota
Columbia University
pkk2125@columbia.edu

***Abstract*—The time and memory complexity of self-attention mechanisms in transformers is quadratic with respect to sequence length, which makes them slow and demanding in terms of memory when processing long sequences. To address this issue, various approximate attention techniques have been proposed to reduce computational costs, but they often fail to deliver significant improvements in real-time performance. A crucial element often overlooked is making attention algorithms aware of input/output (I/O) operations—specifically, considering the reads and writes between different levels of GPU memory. FlashAttention is an I/O-ware exact algorithm that reduces the number of reads and writes between GPU on-chip memory and HBM.**

***Index Terms*—component, formatting, style, styling, insert.**

I. Introduction

Transformers have established themselves as the prevailing standard in deep learning, driving cutting-edge progress in natural language processing, computer vision, and more. A vital element of the Transformer architecture is the self-attention mechanism [7] which allows for efficient representation of long-range dependencies within data. Nonetheless, as sequence lengths increase, the self-attention mechanism's quadratic time and memory complexity creates a major limitation, restricting its scalability to longer sequences and larger datasets.

A variety of strategies have been suggested to tackle this limitation, such as sparse attention mechanisms, low-rank approximations, and kernel-based techniques. Although these techniques lower the computational load, they frequently diminish accuracy or necessitate complex alterations to the architecture. There is an increasing need for methods that improve the performance of self-attention, all while preserving its accuracy and applicability in real-world scenarios.

Flash Attention is an innovative algorithm aimed at tackling this issue by utilizing memory-efficient methods and optimizing hardware. By executing attention calculations directly in shared memory, Flash Attention optimizes memory utilization and drastically decreases computation costs. In contrast to previous approaches, it maintains the complete accuracy of conventional attention while providing notable speed improvements on contemporary GPUs.

In this document, we introduce the Flash Attention algorithm, elaborating on its theoretical foundations, implementation, and performance evaluations. We show how Flash Attention facilitates efficient training and inference for lengthy sequences, unlocking the capabilities of Transformers in areas where scalability was once a constraint. Additionally, we demonstrate its compatibility with conventional deep learning frameworks, rendering it a viable option for practical applications in the real world.

II. Background

A. Attention Mechanism

Given input sequences $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ where N is the sequence length and d is the head dimension, we want to compute the attention output $\mathbf{O} \in \mathbb{R}^{N \times d}$:

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

where softmax is applied row-wise.

Standard attention implementationstakes $O(N^2)$ memory.

Most operations are memory-bound - large number translates to very slow wall-clock times.

Therefore several attempts have been made to implement efficient softmax involving fusion of kernels. Our 1D implementation focuses on that and will be described below.

To implement it on a GPU for parallel computation, a naive implementation is summarized in the algorithm shown.

The primary issue with this method is the extensive use of HBM for data loading and storing. The number of memory transactions can be significantly reduced if a more efficient method involving tiling and SRAM use (Shared Memory) is performed.

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

B. Hardware

The hardware primarily used to test this implementation is the NVIDIA T4 GPU. The T4 GPU has the following features:

C. Understanding Dependence of Software Performance on Hardware

There are various hardware parameters that decide the performance of a gievn software. For a parallel program, there are two critical aspects for optimal execution:

- Compute Throughput
- Arithmetic Intensity

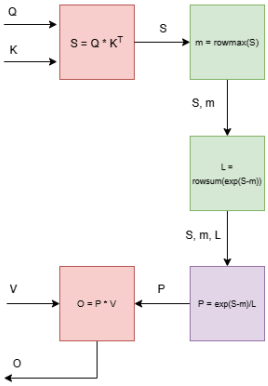


Fig. 1: Standard GPU Attention Pipeline Block Diagram

Device Property	Value
Device Name	Tesla T4
COMPUTE_CAPABILITY_MAJOR	7
COMPUTE_CAPABILITY_MINOR	5
CONCURRENT_KERNELS	1
MAX_REGISTERS_PER_BLOCK	65536
MAX_REGISTERS_PER_MULTIPROCESSOR	65536
MAX_SHARED_MEMORY_PER_BLOCK	49152

TABLE I: Tesla T4 Properties

For a given GPU design, the same software performs differently on different hardware because compute and memory features - maximum FLOPs/s, HBM B/W are different. Therefore, it is possible that the same software can be better optimized to utilize resources available on different platforms.

1) Roofline Plots: Roofline plots help estimate the resource utilization of a particular program on a given hardware.

D. Profiling

In this work, we primarily use the **NVIDIA Nsight Compute** [5] tool to profile our kernels. This is provided as part of the NVIDIA CUDA Toolkit. 1D kernels were profiled to analyze the movement of data in hardware. See figure 3 for inference:

- Roofline shows correlation between scale of problem and throughput
- This means kernels do not exploit all resources provided by the hardware. This indicates a great potential for performance improvement.
- Looking at the instruction mix shows that the number of MOV instructions are significantly higher. This makes it evident that memory will be the bottleneck - supported by increasing arithmetic intensity against throughput seen in roofline.
- It is obvious that throughput scaled with data as there is more to be processed. Looking at the block sizes shows an interesting insight - reduced block size produces more throughput. This implies that having a greater number of blocks leads to better utilization of resources. This is not surprising because the profiler indicated that the resource utilization of the 1D kernel is minimal.

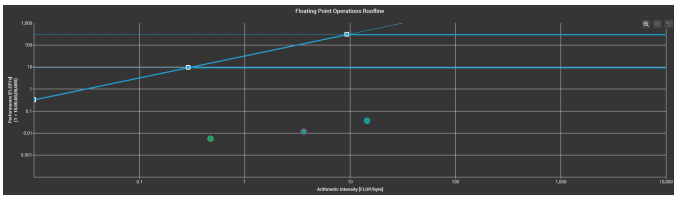


Fig. 2: Roofline Plot

III. FlashAttention: Algorithm and Analysis

The FlashAttention algorithm [6] is I/O aware. It ensures efficient compute of Attention considering HBM bandwidth limitations. The FlashAttention forward pass is described here. Given input sequences $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$, we want to compute the attention output $\mathbf{O} \in \mathbb{R}^{N \times d}$ [1]:

$$\mathbf{S} = \tau \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N},$$
$$\mathbf{P}^{\mathrm{dropped}} = \mathrm{dropout}(\mathbf{P}, p_{\mathrm{drop}}), \quad \mathbf{O} = \mathbf{P}^{\mathrm{dropped}}\mathbf{V} \in \mathbb{R}^{N \times d},$$

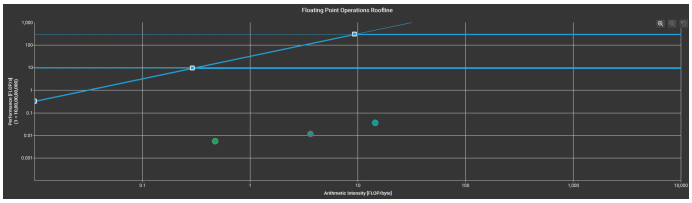
$\tau \in \mathbb{R}$ is softmax scaling (typically $\frac{1}{\sqrt{d}}$) [11],
The full algorithm is in Algorithm 1.

IV. Forward Pass

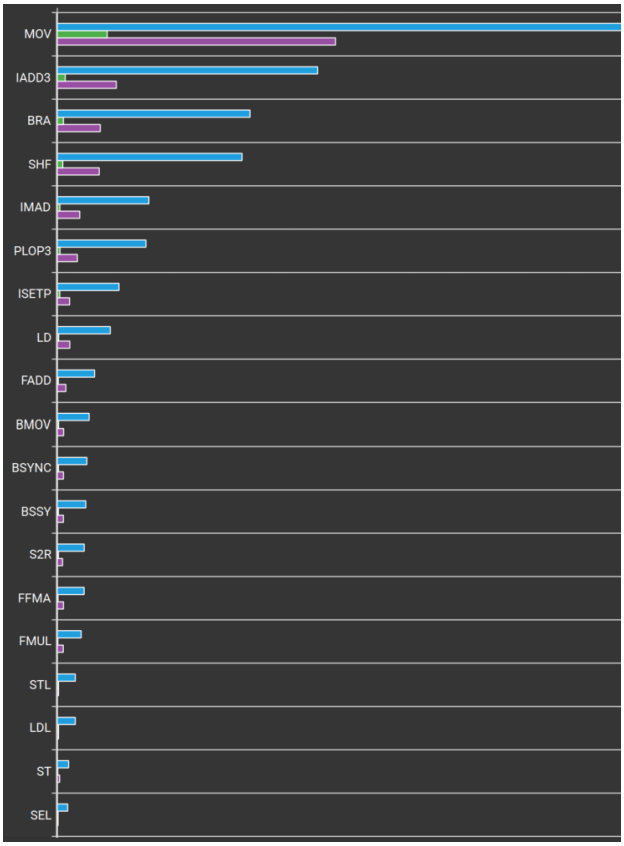
Algorithm 1 FlashAttention Forward Pass [6]

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M, softmax scaling constant $\tau \in \mathbb{R}$

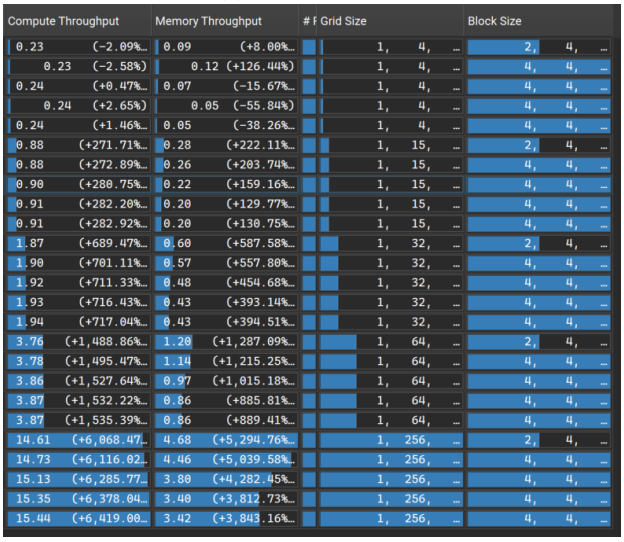
Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  for $1 \le i \le T_r$ **do**
    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$. [10]
    On chip, compute $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij} \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
    Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
  end for
end for
Return $\mathbf{O}, \ell, m, \mathcal{R}$.



(a) Roofline



(b) Instruction Mix



(c) Throughputs

Fig. 3: Profiling 1D kernels

V. Parallelization

The previous sections described a method to break down the attention algorithm into smaller blocks, multiply them, and aggregate results to obtain the exact result of computation [9]. This section describes in detail the parallelization of this algorithm using **CUDA**.

A. Optimization Techniques

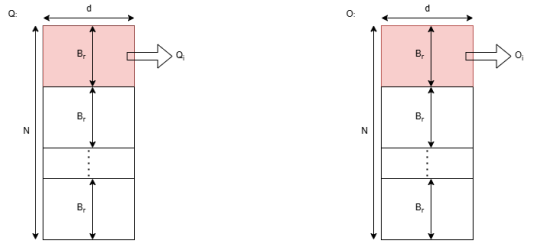
With parallelization, threads have their own register memory and blocks possess their own shared memory. This gives us the opportunity to implement memory-efficient architectures while simultaneously defining certain constraints on accessing the same memory.

1) Tiling: Repeated accesses to global memory are highly time-efficient. Instead, we make use of each block's shared memory by breaking the data into tiles such that each tile contains exactly the amount of information needed by a particular block (abiding by shared memory limitations). Further, this tile is saved to the shared memory of said block. In this new setup, all required memory accesses will be to the shared memory, thus enhancing efficiency [3]. There are 2 possible implementations for tiling based on resource availability and dimensionality of operations, data and grid:

- One-block many-tiles: Due to a limitation on compute or requirement for serialization in the algorithm, multiple tiles can be sequentially fed to the same block via a loop within the kernel. This is called Phasing [4]. Specifics discussed further in Sec. V-B1.
- One-block one-tile: In the event that we have sufficient compute availability and the algorithm is parallelizable to the extent that each tile's computation can run independently, each block is fed a single tile of the aforementioned array. Specifics discussed further in Sec. V-B1.

2) Atomic Operations: When threads executing in parallel seek to update the value of the same memory location at the same time [2], regular assignment operations could overwrite the data between threads due to the time difference between selecting the memory location & reading from it, performing the required operation and updating the new value. To avoid the above, we use Atomic operations, special functions that perform the required update to the memory space while carefully handling/serializing writes such that read-writes for each thread are performed together.

3) Synchronization: In the case of tasks that involve reading and writing to memory locations common to multiple threads - usually all threads in a block when dealing with shared memory or all threads in the grid when dealing with global memory - and are required in the individual execution of each thread, they can be parallelized by instructing different threads to perform different parts of the task and synchronizing the thread execution after. This is inbuilt in the case of syncing threads within a block. Synchronizing threads in all blocks is not as simple.



(a) Tiling of Q (b) Tiling of O

(c) Tiling of K (d) Tiling of V

Fig. 4: Q, O, K, V tiling visualized

B. Different Implementations

Visual representation of how the data is tiled can be seen in Figures 4(a)-(d). How the tiles are being used is described below.

1) 1D Grid: An approach is to parallelize one loop of the two required in the algorithm. This is beneficial as it helps avoid grid-synchronization discussed later. We parallelize the inner loop, and have each O_i, Q_i loaded into consecutive blocks across $blockDim.y$. The grid shape is $(1, T_r, 1)$ to maintain consistency with notation. After loading O_i, Q_i, m_i, l_i, we process each K_j, V_j pair across all blocks in a for loop. In an instance of the loop, all grid blocks will have loaded the same K_j, V_j and will continue execution for $j \in [0, 1, .., T_c - 1]$.

2) 2D Grid: The other approach is to parallelize both loops of the two required in the algorithm. This is expected to be more time-efficient but we run into grid-synchronization (Sec. VIII-A). When considering the 2D grid, We parallelize both the inner and outer loops, and have each O_i, Q_i loaded into consecutive blocks across $blockDim.y$ while each K_j, V_j are loaded into consecutive blocks across $blockDim.x$. The grid shape is $(1, T_r, T_c)$ to maintain consistency with notation. Each block will perform a single independent pass of the pipeline described in Fig. 1 (with the exception of Step 2: m/rowmax will be calculated globally).

VI. Block Diagrams

This section provides the strategies used to compute 1D/2D attention in a visual form.

A. 1D block grid

Blocks are indexed by i. Each block updates its final calculated output into the corresponding tile in O as shown in Fig. 5. Tiles T_i are loaded along rows into corresponding blocks. Tiles of K and V are phased sequentially into all blocks via native looping. This allows for sequential computation for maximum in softmax.

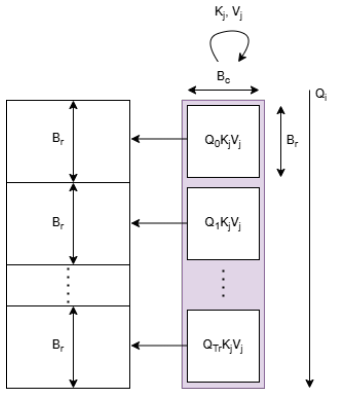


Fig. 5: Tile-Block visualization for 1D Grid Architecture

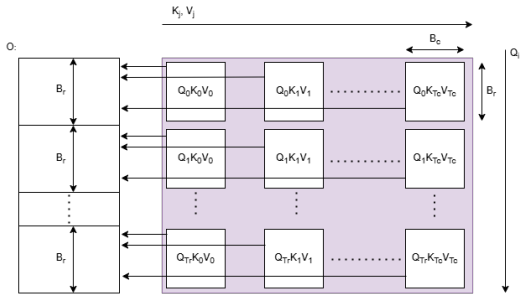


Fig. 6: Tile-Block visualization for 2D Grid Architecture

B. 2D block grid

Blocks are indexed by i for rows and j for columns. Each block updates its final calculated output into the tile in O corresponding to the same row as shown in Fig. 6. Tiles T_i are loaded along rows into corresponding blocks. Tiles of K and V are loaded along columns into corresponding blocks. Computation for maximum in softmax must now be done via synchronization.

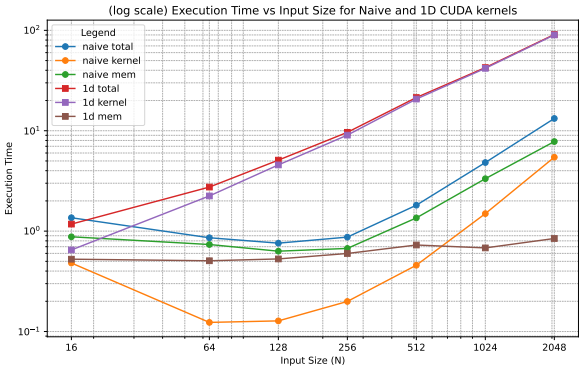
VII. Experiments and Analysis

The source code for our system and experiments is available at https://github.com/eecse4750/e4750-2024fall-project-cuda-ds4399-pkk2125.

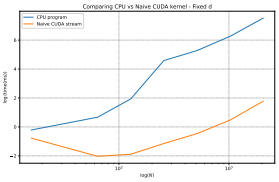
We analyze kernel performances against sequential operations, and also among each other. The goal is to identify improvements in performance made by design choices.

A. Performance Scaling with N (fixed d)

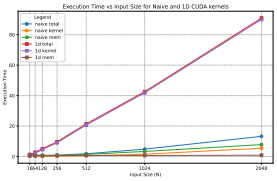
In Fig. 7, we see that for fixed dimension d, CPU execution time increases exponentially with N. However, all CUDA kernels achieve minima in log scale before increasing exponentially. Minima is obtained at optimal N in context of dimension d and grid dimensions. CUDA naive appears to outperform CUDA 1D.



(a) CPU, CUDA naive & CUDA 1D



(b) CPU, CUDA Naive



(c) CPU, CUDA naive & 1D

Fig. 7: CPU, CUDA naive and CUDA 1D kernels with N, for fixed d. (a), (b) - log time, (c) - non-log time.

B. *Performance Scaling with d (fixed N)*

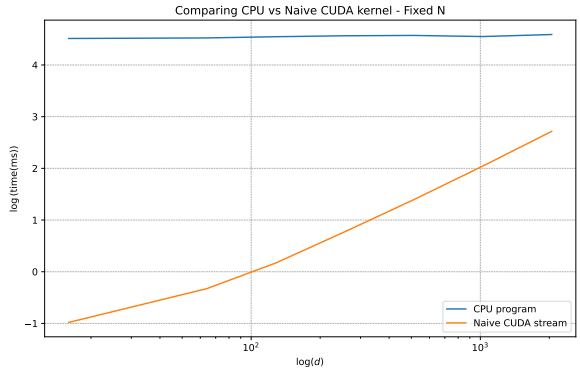
In Figs. 8 and 9, we see that for fixed dimension N, CPU execution time increases exponentially with d similar to the previous flipped case. However, with d, only CUDA naive achieves minima in log scale (minima is obtained at optimal d in context of dimension N and grid dimensions) while CUDA 1D only increases. Again, CUDA naive appears to outperform CUDA 1D but the difference is more significant.

C. *Comparison of Kernels*

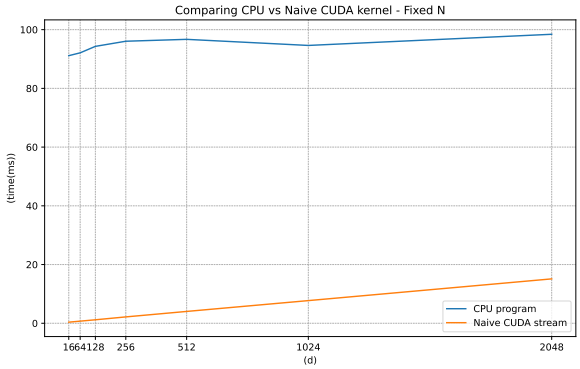
It is unexpected that our naive implementation outperforms the 1D kernel. While this implies that optimizations made are not sufficient, it provides insight into the working of each kernel. The fundamental bottleneck in the attention mechanism is the lack of efficient I/O. This is evident from the execution times of memory operations in the kernels. The 1D grid takes lower time to transfer data. While we do not analyze the internal data transfer times of the GPU kernels, it is evident from the design of algorithms that the 1D grid will outperform the Naive kernel as it performs a lower number of HBM reads of the same data compared to the Naive kernel.

VIII. Challenges Faced

Implementing this algorithm was a very fruitful experience. With deep learning being employed across domains, the scale of computation is increasing and there is an evergrowing need for efficiency [8]. Studying and developing efficient algorithms like this helps save both time and cost of inference, which is very beneficial to users. We faced multiple challenges along that way, some with resolutions and some without.



(a) CPU, CUDA naive



(b) CPU, CUDA naive

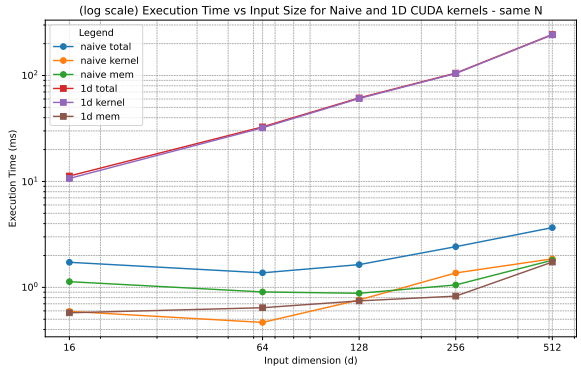
Fig. 8: CPU, CUDA naive with d, for fixed N. (a) - log(time), (b) - time.

A. *Grid synchronization*

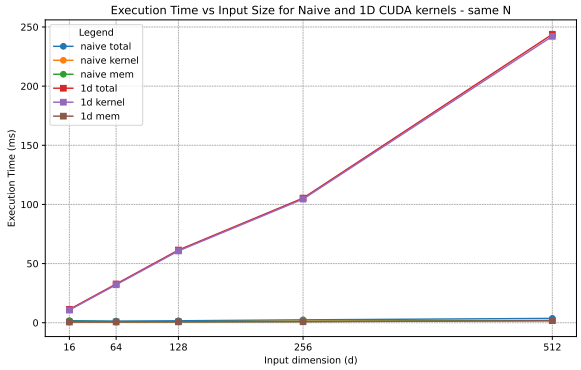
For our 2D grid implementation, it is imperative that the grid is synchronized while computing global row-wise maxima. This is to ensure that blocks using max values to compute Softmax wait for the true global max to be updated in the memory. We circumvent this by splitting our algorithm into separate kernels since completion of a kernel's execution forces full grid synchronization. But this is infeasible for complex kernels and involves wastage of read-writes and memory. Although CUDA offers cooperative grouping primitives, it has not yet been incorporated into PyCUDA. The CUDA solution involved a small change:

```
#include<cooperative_groups.h>

__global__ void kernel(*args){
    /*
    Kernel definitions and code
    */
    cooperative_groups::grid_group grid =
cooperative_groups::this_grid();
    // code
    grid.sync();
}
```



(a) CPU, CUDA 1D



(b) CPU, CUDA 1D

Fig. 9: CPU, CUDA 1D with d, for fixed N. (a) - log(time), (b) - time.

Another limitation of this feature is that all executing blocks need to be active to prevent deadlock type situations. While feasible for small scale problems, it is not possible when dealing with a large number of tokens while performing inference on transformers.

B. Scaling block size

The source paper [6] defines the block sizes as $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$. Given a GPU with M amount of shared memory per block, the choice of block sizes is tied closely to the model dimension d. Also, in our implementation (1D grid) the shared memory can be exactly calculated as

$$M_{req} = \text{sizeof}(\texttt{float32}) \times (2B_r d + 2B_c d + 4B_r + 2B_r B_c + 2B_r)$$

Resources are restricted by scaling B. We further need to analyze what tuples of N, d, B are efficient.

IX. Limitations and Future Work

This work investigated implementations of the forward pass for the FlashAttention algorithm. While a preliminary analysis of the backward pass was done, and efficient implementation is lacking. The next steps would include work in that direction. Also, the system developed fails for some dimensions of inputs - primarily some non-powers of two and odd numbers. This is not a huge concern as most deep learning practitioners ensure a uniform choice of dimensions - powers of two. However for robustness it is neccessary to ensure that all input sizes can be handled.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R. Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, and et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[3] Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, François-David Collin, and Ghislain Durif. Kernel operations on the gpu, with autodiff, without memory overflows. *Journal of Machine Learning Research*, 22(74):1–6, 2021.

[4] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[5] NVIDIA Corporation. Nvidia nsight compute. https://developer.nvidia.com/nsight-compute, 2024. Accessed: 2024-12-17.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[7] Giannis Daras, Nikita Kitaev, Augustus Odena, and Alexandros G. Dimakis. Smyrf-efficient attention using asymmetric clustering. *Advances in Neural Information Processing Systems*, 33:6476–6489, 2020.

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

[9] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[10] Christopher De Sa, Albert Gu, Rohan Puttagunta, Christopher Ré, and Atri Rudra. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1060–1079. SIAM, 2018.

[11] Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(16):14138–14148, May 2021.

X. Appendix

Contributions

A. Code

- Dhruv: 2D kernel, Utils for Shared Memory loading, Matrix Multiplication
- Pranav: 1D kernel, Naive kernel, Scripts to compare CPU/GPU operations, util scripts for plotting performance, profiling

B. Report

Equal Contributions

C. Presentation

Equal Contributions

D. Total

Equal Contributions