Integrating Flash Attention with MiniTorch

Nasrin Kalanat, Vedant Bhasin
Carnegie Mellon University

MOTIVATION

Transformers struggle with inefficiency when processing long sequences due to the quadratic scaling of time and memory requirements associated with self-attention. This limitation has led to the development of approximate attention methods aimed at reducing computational complexity. However, these approaches typically fail to achieve significant improvements in real-time performance. A crucial oversight in existing solutions is the lack of attention to input/output (IO) dynamics, specifically the interactions between various levels of GPU memory. To address this gap, the authors introduce FlashAttention, an exact attention algorithm that incorporates an IO-aware strategy. By optimizing memory interactions, particularly between the GPU's high bandwidth memory (HBM) and on-chip SRAM

IMPORTANCE

Improving the latency and memory footprint of the attention mechanism is crucial for several reasons. Firstly, it directly enhances the scalability of models to handle longer sequences, which is essential for tasks involving more extensive context, such as document summarization or complex question answering. By reducing latency, models can process information faster, making them more practical for real-time applications like machine translation and interactive chatbots. Secondly, optimizing memory usage allows for the deployment of more sophisticated models on resource-constrained environments, such as mobile devices and edge computing platforms. This democratizes access to advanced AI capabilities, enabling broader adoption and integration into everyday applications. Furthermore, more efficient memory management contributes to energy conservation and reduces the carbon footprint associated with training and deploying large models. As the demand for AI grows, sustainability becomes increasingly important, making efficient computing an imperative rather than an option. By addressing these aspects, FlashAttention not only pushes the boundaries of model performance but also aligns with the broader goals of making AI more accessible, responsive, and sustainable.

INTEGRATION

Fused CUDA Kernels: The fused kernels written according to the algorithms described in the Flash Attention paper.

Functions in cuda_kernel_ops: This module contains a collection of functions responsible for launching the CUDA kernels. These functions are the interface which connects the python code and the CUDA kernels.

Flash Attention Class in tensor_functions: The Flash Attention functionality is encapsulated within a class in the tensor_functions module. This class defines the high-level interface for applying the Flash Attention mechanism within the MiniTorch framework. It abstracts the underlying CUDA operations in a class, and is called by a tensor method.

Tensor Method: To integrate Flash Attention into the typical workflow of MiniTorch, a specific tensor method is implemented. This method extends the MiniTorch tensor class, allowing users to directly apply Flash Attention to tensors as part of their model architectures. The method internally calls the Flash Attention function from the tensor_functions class, simplifying the process.

ALGORITHM

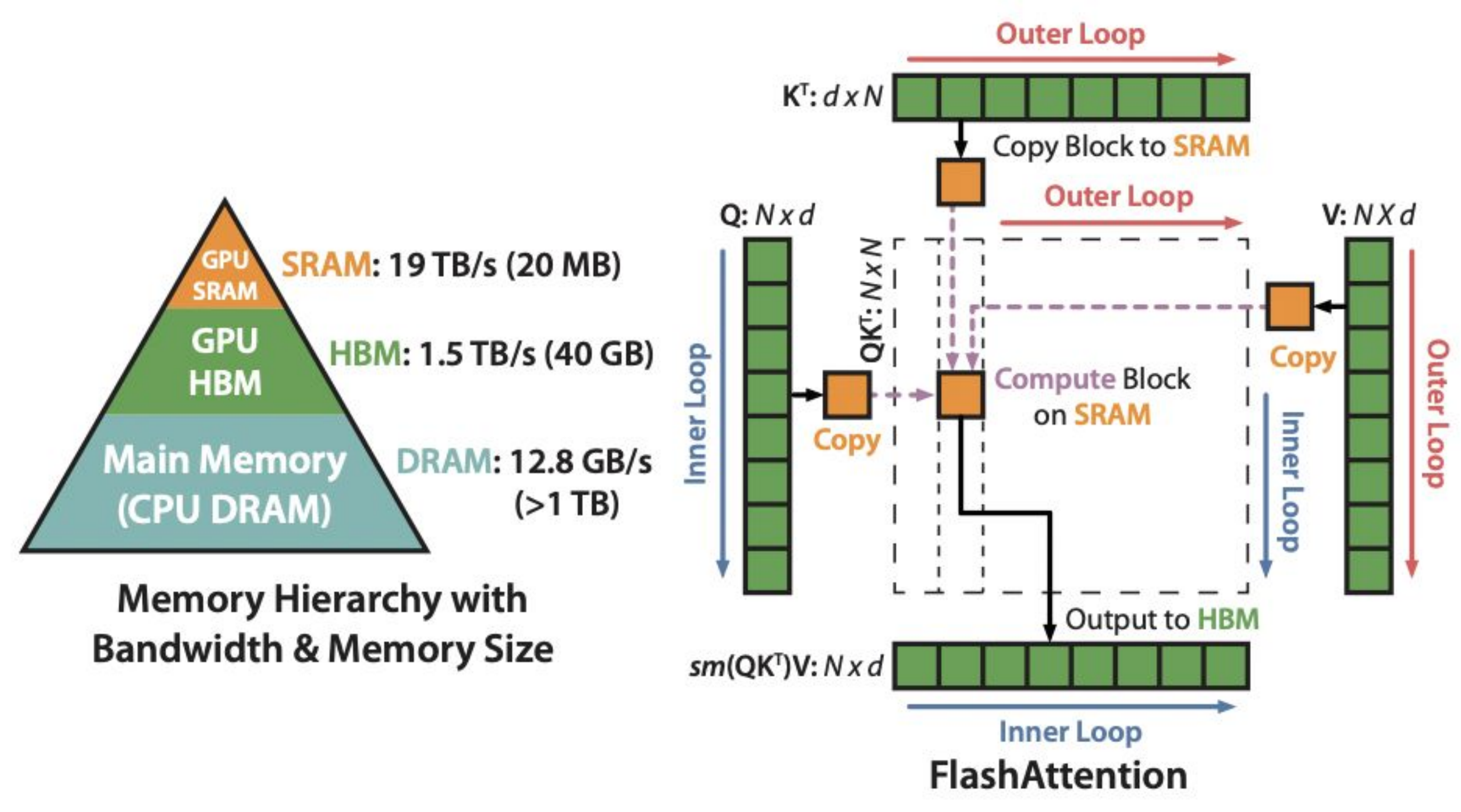

Figure 1: Visualization of Flash Attention forward algorithm with the memory hierarchy

The Flash Attention algorithm integrates two strategies tiling and recomputation to enhance computational efficiency by reducing both latency and the memory footprint. Tiling involves segmenting the input sequence into smaller, more manageable blocks or tiles. This partitioning facilitates more strategic memory usage, as each tile is processed sequentially rather than loading the entire sequence into the memory all at once. Such an approach not only drastically reduces the memory load but also optimizes cache utilization on GPUs. Enhanced cache efficiency means that data can be accessed and processed quicker, which, in turn, accelerates the overall computation speed and improves the throughput of processing large sequences.

On the other hand, recomputation is employed to further boost system efficiency by recalculating intermediate values as needed, rather than persistently storing them. This technique effectively decreases the volume of memory required at any moment, liberating vital computational resources to manage more extensive models or longer sequences. This strategic choice of recalculating data rather than storing extensive intermediate states helps in maintaining a lower memory footprint, crucial for scaling the models to longer sequences or deploying them on memory-constrained platforms.

Algorithm 1 FLASHATTENTION

```
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM, on-chip SRAM of size $M$.
1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: 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.
3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks
   $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_i, \dots, \ell_{T_r}$ of size $B_r$ each,
   divide $m$ into $T_r$ blocks $m_1, \dots, m_{T_r}$ of size $B_r$ each.
5: for $1 \le j \le T_c$ do
6:   Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7:   for $1 \le i \le T_r$ do
8:     Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9:     On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
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), $\tilde{\ell}_{ij} =$
       $\mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11:    On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12:    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.
13:    Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
14:  end for
15: end for
16: Return $\mathbf{O}$.
```

Figure 2: The Flash Attention algorithm

PERFORMANCE

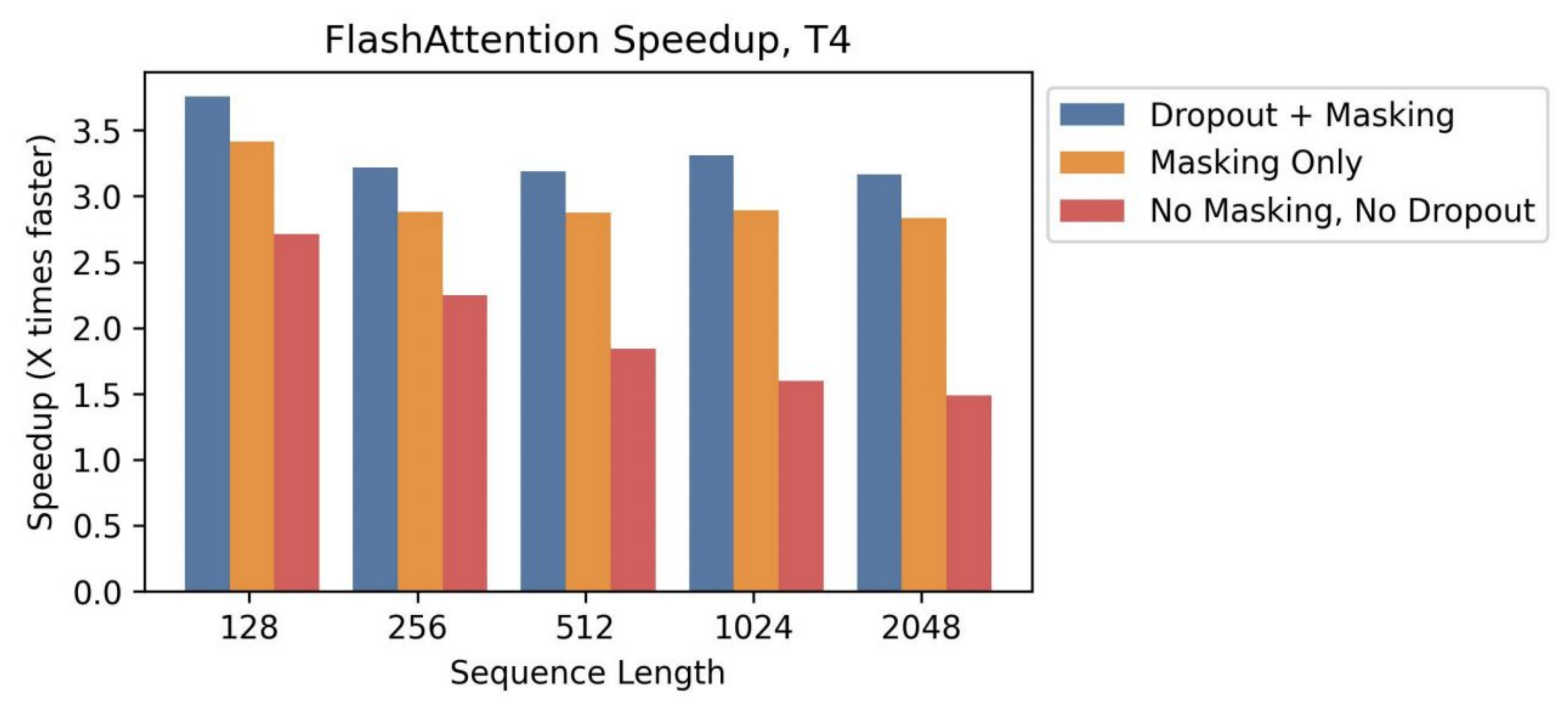

Figure 3: Reported performance improvement for different sequence lengths on a T4 GPU

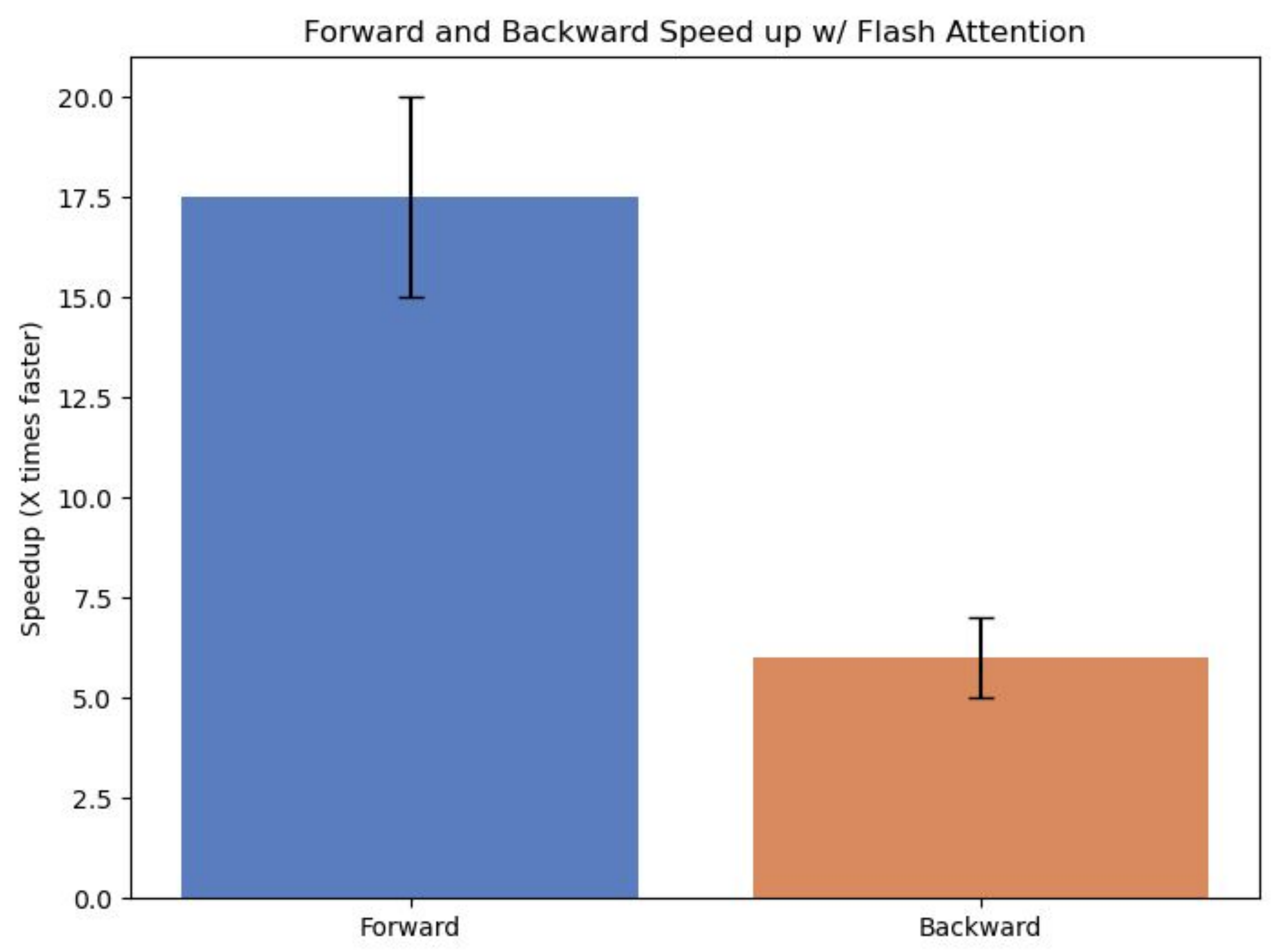

Figure 4: Our measured performance improvements using Flash Attention with minitorch

CONCLUSIONS

The Flash Attention algorithm marks a pivotal advancement in transformer optimizations, particularly in reducing latency and memory usage through IO-aware strategies such as tiling and optimized memory access. A key insight from this project is the critical role of IO-awareness in enhancing computational efficiency, which has proven essential for managing large-scale models effectively. Integrating advanced mechanisms like Flash Attention into autodiff frameworks such as MiniTorch has shown that sophisticated computational techniques can be seamlessly adopted, broadening their application and facilitating innovation. This integration not only supports the development of more efficient AI models but also underscores the importance of aligning algorithm design with hardware capabilities. These developments suggest a promising direction for future AI technologies, emphasizing sustainable, efficient, and powerful systems.