CS542200 Parallel Programming

Homework 4: FlashAttention

Due: Dec 25 23:59, 2024

1 Goal

This assignment focuses on implementing the forward pass of the FlashAttention. You will gain an understanding of how efficient attention mechanisms work and how CUDA can be utilized to accelerate them. In this assignment, you will realize the performance benefits of optimizing the attention computation on GPUs. Finally, you are encouraged to explore and apply additional optimization strategies to maximize performance .

2 Requirements

- In this assignment, you are asked to implement a CUDA-based program for the forward pass of the FlashAttention.
  - Your implementation must leverage CUDA to compute the forward pass of FlashAttention.
  - You need to implement using the core concepts of the FlashAttention algorithm, including kernel fusion and tiling techniques.
  - You are encouraged to optimize program performance by adjusting the data replacement and parallelism levels.
  - The program must correctly handle memory management, kernel launches, and ensure the correctness of the attention computation.

3 FlashAttention algorithm

In this assignment, You are asked to implement the FlashAttention algorithm. Below is the algorithm overview:

- Inputs
  - $Q, K, V \in R^{N*d}$: Query, Key, and Value matrices in HBM (High Bandwidth Memory).
- Outputs
  - $O \in R^{N*d}$: Output matrices.

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.

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 $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  end for

end for

Return $\mathbf{O}$.

FlashAttention Algorithm Explanation:

The FlashAttention algorithm is designed to efficiently compute scaled dot-product attention by leveraging GPU memory hierarchy, which includes High Bandwidth Memory (HBM) and on-chip SRAM. This method minimizes memory read and write operations, addressing the bottlenecks associated with memory-bound attention computations.

In FlashAttention, the input matrices Q , K, and V (queries, keys, and values) are divided into blocks of manageable sizes. This ensures that intermediate computations can be performed within the on-chip SRAM, which has significantly higher bandwidth compared to HBM. The algorithm operates in two main loops:

1. **Outer Loop:** Blocks of K and V are copied from HBM to SRAM.
2. **Inner Loop:** Each block of Q is loaded into SRAM, and the attention matrix (QK^T) is computed block-wise. This matrix undergoes row-wise max normalization and exponentiation to ensure numerical stability. The normalized attention scores are then used to compute the weighted sum of V, resulting in the output matrix O.

The key optimization lies in performing all the computationally intensive operations (e.g., matrix multiplications, softmax) directly on SRAM to reduce the frequency of accessing HBM. This optimization not only accelerates computation but also reduces energy consumption.

Paper link:

https://arxiv.org/abs/2205.14135

4 RUN YOUR PROGRAMS

- **Preparation (only need to execute it once at the beginning)**

```
# On Apollo GPU
mkdir -p ~/hw4
cp /home/pp24/share/hw4/seq-attention.c ~/hw4
cp /home/pp24/share/hw4/seq-flashattention.c ~/hw4
cp /home/pp24/share/hw4/Makefile ~/hw4

# On NCHC Container
mkdir -p ~/hw4
cp /tmp/hw4/seq-attention.c ~/hw4
cp /tmp/hw4/seq-flashattention.c ~/hw4
cp /tmp/hw4/Makefile ~/hw4
```

- **Command line specification**

```
# On Apollo GPU
srun -N1 -n1 --gres=gpu:1 ./hw4 INPUTFILE OUTPUTFILE

# On NCHC Container
./hw4 INPUTFILE OUTPUTFILE
```

    - `INPUTFILE`: The pathname of the input file. Your program should read the input matrix from this file.

- OUTPUTFILE: The pathname of the output file. Your program should output the output matrix to this file.

- **Input specification**
  - The input file is a binary file containing 32-bit integers (*B, N, D*) and 32-bit floating-point numbers (*Q, K, V*).
  - The first three integers are *the batch size (B)*, *the sequence length (N)* and *the embedding size (d)*.
  - Then, there are *B* batches. Each batch consists of *Query (Q)*, *Key (K)*, and *Value (V)* matrices:
    1. *Query (Q)*: *N * d* floating-point numbers
    2. *Key (K)*: *N * d* floating-point numbers
    3. *Value (V)*: *N * d* floating-point numbers
  - The ranges for the input are:
    - $2 \leq B \leq 14000$
    - $N \in \{128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768\}$
    - $d \in \{32, 64\}$
    - $-3.0 \leq Q_{i,j}, K_{i,j}, V_{i,j} \leq 3.0$
    - $B * N * d < 56000000$
- **Output specification**
  - The output file is also in binary format.
  - You should output an output file containing $B * N * d$ floating-point numbers.

5 REPORT

Answer the questions below. You are recommended to use the same section numbering as they are listed.

1. Implementation

a. Describe how you implemented the FlashAttention forward pass using CUDA. Mention the algorithm's key steps, such as matrix blocking, SRAM usage, and how intermediate results like scaling factors (ℓ and m) were calculated.

b. Explain how matrices Q, K, and V are divided into blocks and processed in parallel.

c. Describe how you chose the block sizes B_r and B_c and why.

d. Specify the configurations for CUDA kernel launches, such as the number of threads per block, shared memory allocation, and grid dimensions.

e. Justify your choices and how they relate to the blocking factors and the SRAM size.

2. Profiling Results

Provide the profiling results of following metrics on the kernel of your program using NVIDIA profiling tools. NVIDIA Profiler Guide.

- occupancy
- sm efficiency
- shared memory load/store throughput
- global load/store throughput

> Note:
>
> To run nvprof on Apollo GPU with flags like --metrics, please run on the slurm partition nvidia. e.g. srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --metrics gld_throughput ./hw4 /home/pp24/share/hw4/testcases/t01 t01.out
>
> Please note that nvprof is only available for the GTX 1080. For more details, please refer to the Nvprof Transition Guide.

3. Experiment & Analysis

a. System Spec

State which platform you used to conduct your experiment. If you didn't use our Apollo-GPU server or NCHC container for the experiments, please show the CPU, RAM, disk of the system.

b. Optimization

Any optimizations after you port the algorithm on GPU, describe them with sentences and charts. Here are some techniques you can implement:

- Coalesced memory access
- Shared memory
- Handle bank conflict
- CUDA 2D alignment
- Occupancy optimization
- Streaming

c. Others

Additional charts with explanation and studies. The more, the better.

4. **Experience & conclusion**
    a. What have you learned from this homework?
    b. Feedback (optional)

6 GRADING

1. **[60%] Correctness**

A total of 30 test cases, from t01 to t30, will be used to test your implementation.

- You get 60 points if you passed all the test cases, $max(0, 60 - 3k)$ points if there are k failed test cases.
- Each test case has a time limit.
    - Apollo GPU (GTX1080)
        - t01 - t10: 2 seconds.
        - t11 - t20: 6 seconds.
        - t21 - t30: 10 seconds.
    - NCHC Container (RTX3070)
        - t01 - t10: 1 seconds.
        - t11 - t20: 3 seconds.
        - t21 - t30: 5 seconds.
- The floating-point numbers in the output file must have an error margin within 5e-3 compared to the correct answer to be considered correct. If any number exceeds this margin, it will be judged as a wrong answer.
- You can run your program on either the Apollo GPU or the NCHC Container. Successfully passing on either platform will be considered as completing the assignment.
- We will test your program on both platforms and take the better result as your final grade, so you only need to complete the assignment on one platform.

2. **[20%] Demo**

- A demo session will be held remotely. You'll be asked questions about the homework.

3. **[20%] Report**

- Grading is based on your evaluation, discussion and writing. If you want to get more points, design or conduct more experiments to analyze your implementation.

7 Submission

Upload the files below to eeclass. (**DO NOT COMPRESS THEM**)

- hw4.cu
- Makefile (optional)
- hw4_{student_ID}.md

8 Final Notes

- Type `hw4-judge` to run the test cases on Apollo GPU.
- Type `hw4-remote-judge` to run the test cases on NCHC Container.
- If you want to submit your Makefile, **please combine it into a single file**. All of your code should be compiled by `make hw4` with the same Makefile.
- Scoreboard:
  - Apollo GPU: https://apollo.cs.nthu.edu.tw/pp24/scoreboard/hw4/
  - NCHC Container: https://apollo.cs.nthu.edu.tw/pp24/scoreboard/hw4-remote/
- Resources are provided under `/home/pp24/share/hw4/` on Apollo-GPU and `/tmp/hw4/` on NCHC Container.
  - `seq-attention` - a sequential code of original attention implementation
  - `seq-flashattention` - a sequential code of flashattention implementation
  - `Makefile` - example Makefile
  - `testcases/` - sample test cases
- Contact TA via pp@lsalab.cs.nthu.edu.tw or eeclass if you find any problems with the homework specification, judge scripts, example source code or the test cases.
- You are allowed to discuss and exchange ideas with others, but you are required to write the code on your own. You'll get **0 points** if we found you cheating.