Parallelizing Multi-Head Attention on GPUs

Tanvi Bhandarkar
UID:705525339
tanvi17@ucla.edu

Hemil Desai
UID:405508015
hemil10@ucla.edu

Kriti Gupta
UID:305491572
kritigupta@ucla.edu

Abstract

Transformers have become ubiquitous for Machine Learning tasks related to Natural Language Processing. They are also being actively researched for Computer Vision applications and have shown some promising results. This is exciting because this leads to the unification of a common model for a majority of Machine Learning tasks. As a result, optimizing Transformer kernels and coming up with novel GPU parallelization for Transformer kernels can have huge payouts. We explore and evaluate different parallelization techniques for Transformers on GPU using CUDA. In particular, we implement a parallel reusable implementation for the multi-head self attention kernel, which forms the core of the transformer model. We find that a vertical strategy utilizing CUDA streams that parallelizes a single operation of the attention kernel achieves speedups between 2x and 4x

1 Introduction

Natural Language Processing (NLP) is advancing quickly, in part due to an increase in available compute and dataset size. The abundance of compute and data enables training increasingly larger language models via unsupervised pre-training. Transformers have become ubiquitous for Machine Learning tasks related to Natural Language Processing ever since the papers Attention is all you Need [1] and BERT [2] were released.They are also being actively researched for Computer Vision applications and have shown some promising results. This is exciting because this leads to the unification of a common model for a majority of Machine Learning tasks. As a result, optimizing Transformer kernels and coming up with novel GPU parallelization for Transformer kernels can have huge payouts. We explore and evaluate different parallelization techniques for Transformers on GPU using CUDA. In particular, we implement a parallel reusable implementation for the multi-head self attention kernel, which forms the core of the transformer model. Self-attention computations are the same, just replicated across multiple heads with different parameters. As a result, this provides many opportunities for data and input reuse. Moreover, the output of each head is concatenated and multiplied by a common matrix, thus necessitating the parallel and timely execution of the different heads.
We basically explore two broad directions of experimentation:

- Combine all CUDA kernels in a single MultiHeadAttention and parallelize across native C++ threads
- Run CUDA kernels at the same level of Attention in parallel using CUDA streams and events

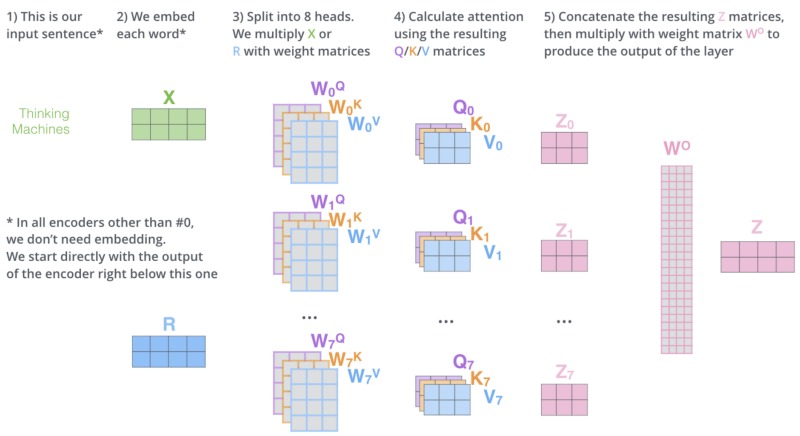

Figure 1: Multi-Head Attention

2 Related Work

2.1 Transformers

The sequence-to-sequence models are based on use of recurrent neural network in encoder-decoder architecture. These architectures have limitations when the input has long sequences and after certain time stamp they cannot entirely retain the information from initial elements when the model starts incorporating new elements into the sequence. To deal with this limitation of decoder new concept of transformer model [Attention is all you need] was introduced. It is entirely built on the mechanism of self-attention without using sequence-aligned recurrent architecture.

The Attention function can be described as mapping a query and a set of key-value pairs to an output. Here, the query, keys, values and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. A single head attention kernel can be described as

$$Softmax(\frac{Q \times K^T}{\sqrt{d_k}}) \times V \tag{1}$$

2.1.1 Multi-Head Self-Attention

The multi-head mechanism checks the scaled dot-product attention multiple times in parallel. Instead of only computing once, the output of independent attention are simply concatenated and linearly transformed into expected dimensions. It allows the model to jointly attend to information from different representation subspaces at different positions. All of these similar Attention calculations are then combined together to produce a final Attention score. This is called Multi-head attention and gives the Transformer greater power to encode multiple relationships and nuances for each word. An overview of the multi-head attention kernel is presented in Figure 1.

2.2 NVIDIA's Megatron-LM

Megatron-LM [6] is a powerful transformer that supports model-parallel and multi-node training. A transformer layer consists of self-attention block followed by two-layer multi-layer perceptron(MLP) as shown in figure 2. As shown in figure, the model parallelism is separately introduced in each of these block. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

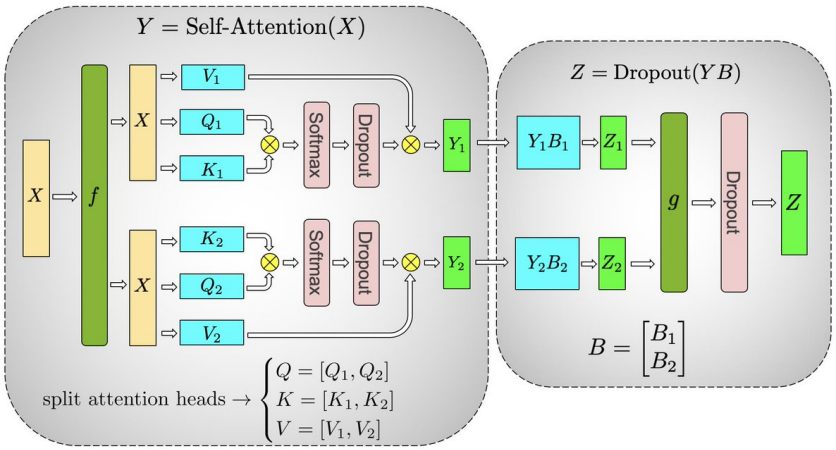

Figure 2: Megatron-LM

The model parallel approach can be characterized as techniques aimed at reducing communication and keeping the GPUs compute bound. Rather than having one GPU compute part of the dropout, layer normalization, or residual connections and broadcast the results to other GPUs, the Megatron duplicates the computation across GPUs. Since all values are either local or duplicated on a GPU, there is no need for communicating updated parameter values int this formulation.

3 Methods

As described in the transformer introduction above, the Multi Head Attention Kernel includes three core operations:

- General Matrix Multiplication (GEMM)
- Softmax
- Transpose

We first start by implementing these kernels natively on CUDA and describe the implementation of each. Next, we look at techniques to parallelize the entire Multi Head Attention Kernel vertically and horizontally using native C++ threads and using CUDA streams and kernels respectively. The core of a multi head attention kernel is the single head attention kernel. The chronological order of a single head attention kernel can be described as:

- $X * W_Q \rightarrow Q \ (Q \ calculation)$
- $X * W_K \rightarrow K \ (K \ calculation)$
- $X * W_V \rightarrow V \ (V \ calculation)$
- $Q * K^T (K^T calculation)$
- $Softmax(\frac{Q*K^T}{\sqrt{d_k}})$
- $Softmax(\frac{Q*K^T}{\sqrt{d_k}}) \times V \ (Softmax * V calculation)$

3.1 CUDA Kernels for Core Operations

GEMM: We utilize shared memory and a tiling approach to implement matrix multiplication. Since shared memory is shared per block, this gives a lot of reuse opportunities. The number of tiles are calculated by $(ROW\_SIZE/BLOCK\_SIZE) \times (COL\_SIZE/BLOCK\_SIZE)$. Each thread calculates one output element. It slides the tiles across the matrix and stores the elements in shared memory. We first initialize a matrix of size $BLOCK\_SIZE \times BLOCK\_SIZE$ (one each for the two matrices being multiplied respectively) in shared memory for each tile. Each thread in

the block then loads one element into the shared memory. The threads are then synced. Then, we calculate the partial dot product of the shared memory matrices corresponding to the thread id. We accumulate these sums into a variable. This consists of one loop iteration. Then we move onto the next tile and repeat the process. We impose error checking to prevent load and store for out of bounds indices. After the tiles have slid over the entire matrix, we store the final sum into the output matrix. This shared memory approach provides great throughput and latency benefits because each thread in a block just does one global access per tile and gets to reuse $BLOCK\_SIZE$ elements from shared memory (for MACs for its partial row and column). Throughput improves because the weights are reused from shared memory across different inputs when increasing the batch size [5]. [1]

Softmax: Softmax is calculated across each row of a N*N matrix. For the CUDA kernel implementation, we use a 1D grid across the row dimension. We utilize parallel sum reduction using threads to efficiently store and calculate the softmax sums [3]. First, we slide the tile across the row of the matrix. Each thread in the tile keeps on accumulating the exponentiated sum. The sum reduction of each thread is then stored in a shared memory array. After the tile has reached the end of the row, the threads are synced and then another level of sum reduction occurs in the shared memory array. In the end, the exponent value of the current element is divided by the sum to get the softmax value.

Transpose: Transpose is also implemented using shared memory. The kernel is optimized to ensure all global reads and writes are coalesced, and to avoid bank conflicts in shared memory. Note that the shared memory array is sized to $(BLOCK\_SIZE + 1) * BLOCK\_SIZE$. This pads each row of the 2D block in shared memory so that bank conflicts do not occur when threads address the array column-wise [4]. [2]

These core kernels are combined in the correct chronological order to get a single head attention kernel. Combining the outputs from multiple single head attention kernels yield the Multi Head Attention Kernel. As evident, Single Head Attention kernels can be parallelized since they are independent operations. We also present a breakdown and analysis of a chronological Single Head Attention Kernel in the results below.

3.2 Parallelization Strategies

We try out two parallelization strategies to parallelize the multiple single head attention kernels in Multi Head Attention. At a higher level, these can be classified into vertical and horizontal parallelization. At a technical level, the two approaches use CUDA streams/kernels and native C++ threads respectively.

Native C++ Threads: For this parallelization strategy, we create a single function for single head attention, and then parallelize that function across native C++ threads. We pass in the parameters that need to remain the same, and initialize other parameters in the function directly. We use std::thread to launch multiple function calls in parallel. This can be considered a horizontal strategy because one function can be considered as one horizontal row in 2 and each thread is running one function. Therefore, multiple C++ threads are running in parallel.

CUDA streams: For this parallelization strategy, we create CUDA streams to execute a single operation for Single Head Attention (like calculating the Q value), but the streams execute this same operation for each of the different heads. In this way, the streams execute the same CUDA kernel but with some shared parameters and some different parameters. This way, each operation in all attention head executes at the same time and then the kernels move onto the next operation. This can be considered as a vertical strategy because in every row of 2 the streams execute an operation like multiplication with weights vertically across all heads, then all the streams move onto the next operation (column) i.e. calculating attention.

[1] This strategy is adapted from mini-project 1

[2] https://github.com/JonathanWatkins/CUDA/blob/master/NvidiaCourse/Exercises/transpose/transpose.cu

4 Methodology and Evaluation

We test the native kernels as well as the two parallelization strategies across different batch sizes, embedding dimensions and attention head dimensions. We also generate a breakdown of the different operations in the single head attention kernel.

4.1 Single Head Attention Kernel Breakdown

First, we analyze each individual operation in Single Head Attention to calculate the CUDA warmup times as well as any bottlenecks. This breakdown is calculated on a single thread. We first present the breakdown for the first CUDA call to incorporate the CUDA warmup times in Table 1 and then present the subsequent average of 5 calls in Table 2. The parameters used for this evaluation are:

- Batch Size - 1
- Sentence Length - 32
- Attention Head size - 64
- Embedding Dimension size - 768

Operation	**Time (s)**
CUDA Malloc and Memcopy	0.150643
Q	2.9e-05
K	0.001255
V	0.001282
K Transpose	1.4e-05
Softmax	0.000301
SoftmaxMatrix * V	0.00579
Total	0.16

Table 1: Single Head Attention Breakdown for first CUDA call

Operation	**Time (s)**
CUDA Malloc and Memcopy	0.002243
Q	1e-05
K	0.001779
V	0.001492
K Transpose	1.3e-05
Softmax	2.1e-05
SoftmaxMatrix * V	7.5e-05
Total	0.0056

Table 2: Single Head Attention Breakdown for subsequent CUDA calls

As we can see, the time difference between the first call and subsequent calls is substantial (0.1544s). This is due to CUDA warmup incurred for the first call. We will now see whether this warmup is replicated across parallel C++ threads.

4.2 Horizontal Parallelization

Now that we have seen the breakdown of a single attention kernel, let's compare the times if we run the single head attention kernels in parallel versus synchronously. We want to see whether the CUDA warm-up times are replicated across threads. Table 3 shows the result for synchronous execution of Multi Head Attention (executing the 12 heads in sequence) vs Asychronous execution using C++

Batch Size	Sentence Length	Attention Head Size	Embedding Dimension	Synchronous Time (s)	Asynchronous Time (s)
1	32	64	768	0.045244	0.417439
8	64	64	768	0.05776	0.401976
8	524	64	768	1.7438	2.05031
16	32	64	768	0.057411	0.391909

Table 3: Time comparison for C++ Threads Parallelization

threads. We compare the times across a variety of parameter sets as shown in the table. We notice that the synchronous version performs much better than the asynchronous version indicating that in the async version, each thread is incurring a CUDA warmup time. For smaller parameter sizes, the time difference is substantial as the warmup time overpowers rest of the operations. However, for bigger parameter sizes the warmup time is not so dominant. We conclude that the high CUDA warmup times do not warrant the use of native threads to parallelize CUDA kernels. Further investigation is needed into the possible reasons and resolutions of the warmup time.

Batch Size	Sentence Length	Attention Head Size	Embedding Dimension	Synchronous Time (s)	Asynchronous Time (s)
1	32	64	768	0.000612	0.000306
8	32	64	768	0.001656	0.000346
16	64	64	768	0.004788	0.001127
16	128	64	768	0.005832	0.002216

Table 4: Time comparison for CUDA streams Parallelization for calculation of Q

4.3 Vertical Parallelization

Since the warmup times pose a substantial overhead in the previous strategy, we try a vertical strategy using CUDA streams and kernels. We try to vertically parallelize each operation of the single head attention kernel. For comparison purposes, we calculate the comparison times across a single operation of the kernel. In Table 4 we show the execution times of matrix multiplication across CUDA streams and see the comparison for synchronous vs asyncrhronous execution. Since there are 12 attention heads in most multi-head attention kernels, we use 36 CUDA streams to calculate the Q, K and V values for each head in parallel. We see speedups of between 2x to 4x for the single matrix multiplication operation across the single head attention kernels. We conclude that vertical parallelization is the preferred choice for parallelizing multi head attention on CUDA. We can synchronize each operation in the kernel to operate in parallel and then proceed chronologically across the single head attention kernel.

5 Conclusions

In comparing different parallelization strategies, we notice that using native C++ threads incur a CUDA warm-up time across each thread which hinders the parallelization of the single head attention kernel. In comparison, a vertical strategy utilizing CUDA streams that parallelizes a single operation of the attention kernel works best. A single operation using CUDA streams achieves speedups of up-to 2x using CUDA streams. Parallelizing all of the operations of the kernel using CUDA stream would thus achieve substantial speedups in total ($\approx$ 2x * 5 -> 10x)

6 Statement of Work

- Tanvi Bhandarkar - Experimented with CUDA kernels at the same level of Attention in parallel using CUDA streams and events.
- Hemil Desai - Implemented native CUDA kernels for the GEMM, Softmax and Transpose operations.
- Kriti Gupta - Combined all CUDA kernels in a single function and experimented with parallelization across native C++ threads

The rest of the work including Project Presentation and Project report was shared among all the authors.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL `http://arxiv.org/abs/1810.04805`.

[3] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4): 1–39, 2007.

[4] Dmitry I Lyakh. An efficient tensor transpose algorithm for multicore cpu, intel xeon phi, and nvidia tesla gpu. *Computer Physics Communications*, 189:84–91, 2015.

[5] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. A coordinated tiling and batching framework for efficient gemm on gpus. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 229–241, 2019.

[6] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.