Fall 2023 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Parallel matrix multiplication for improving memory efficiency and performance for Machine Learning Algorithms

Danyal Maqbool & Wenxuan Tan

December 13, 2023

Abstract

Owing to the increasingly large nature of data used in Machine Learning, GPUs have become a cornerstone of training models. At the heart of this training is matrix multiplication. We propose a solution that splits this multiplication across multiple GPUs for maximum efficiency and utilisation. We were able to successfully create a pipeline for distributing the multiplication of two matrices across multiple GPUs and we were partially successful in creating a toy multi-layer perception that leverages our solution.

Link to Final Project git repo: https://github.com/DanyalMaq/FinalProject

Contents

1 General Information

1. Name: Danyal Maqbool
2. Email: dmaqbool@wisc.edu
3. Home department: L&S
4. Status: Master's Student
5. Teammate: Wenxuan Tan
6. I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2 Problem Statement

As recent foundational models grow larger to handle more intricate tasks and web-scale datasets, the demand for computational power surges because it's impossible to store the model weights on a single device. For example, the most popular open-source language model LLaMA 70B [**2**] takes 280GB to store in fp32 and about 10 times more to train. The necessity of efficient large scale distributed training arises from the necessity to parallelize the training process, sharding the gradient computation and weights across multiple GPUs to accelerate the training time and enhance overall performance. However, orchestrating such parallelization often leads to under-utilization due to fault tolerance, load balancing, and communication overhead between GPUs. In the project we attempt to efficiently parallelize matrix multiplication–the central operation in all ML models and to implement the popular tensor parallel(TP) and pipeline parallel(PP) on a distributed MLP as a proof of concept.

3 Solution description

3.1 Matrix kernel

We first modified the matrix multiplication kernel to be able to handle matrices of different dimensions while also maintaining the use of shared memory to ensure high performance in computation. This was necessary as splitting matrices across GPUs would lead to non-square matrices being multiplied. Our kernel leverages the same tiled approach as was discussed in class but this time handles boundary cases for when the matrices are of two different sizes. We've also made activation function kernels(ReLU and softmax) that apply the named function to the kernel during an MLP forward pass. We also created a transpose kernel and a kernel to extract the first n columns of a matrix using the cuBLAS library as helper functions where needed.

3.2 Memory management

We attempted to use both managed memory and manually allocated device memory as a means to prepare the matrices for the kernel. We found that managed memory matrices had a significantly larger time cost, likely due to the page faults incurred when multiple devices are trying to access different parts of the matrix simultaneously in their kernels. Committing memory to each device the matrix that it would operate on provided the expected speedups in performance so we opted to use manual allocations for our primary tests.

We also aimed to utilise peer to peer communication to further reduce memory movement by avoiding staging data transfers via the CPU and instead transferring bytes via the shortest PCIe path.

For the graphs we use the shorthand described below to indicate which plot refers to which memory management approach

- Managed memory allocation on devices $\rightarrow$ managed
- Manual memory allocation on devices $\rightarrow$ manual
- Asynchronous memory allocation on devices $\rightarrow$ async

We then implemented from scratch an MLP in one of the most popular parallelisms–tensor parallel combined with pipeline parallel. There are 4 common types of parallelisms:

- Model parallel(MP): Splits model weights layer-wise and place layer groups on different GPUs.
- Data Parallel (DP): The full weights are replicated on each device, but data is scattered(split) to each device.
- Tensor Parallel (TP): both the input and weights are scattered to each device, so the computation is distributed evenly.
- Pipeline Parallel (PP): Similar to MP, but removes GPU idleness by pipelining overlapping compute and data copy between layers.

3.3 Multi-Layer Perceptron implementation

We adapted the TP design from Megatron-LM [**1**]. For the first two layers, we split them as in Figure 3. X is copied to each device, the first weight matrix is split column-wise (transpose $\rightarrow$ memcpy to each device $\rightarrow$ transpose) and the second layer is split row-wise, followed by a reduce before the final ReLU activation (we removed dropout and used ReLU instead of GeLU). This design is essential: it removes one reduce operation after the first layer because we can compute the non-linear activation on each device instead of reduce $\rightarrow$ ReLU and feed each shard to the next layer (see Figure 2).

For mathematical correctness we review basic linear algebra. $X_{b\times m}$ has b vectors in some m-dimensional vector space, and $A_{m\times n}$ has n bases of dimension m that defines a new vector space; the same for $B_{n \times l}$. f(x) = X * A is the **linear** function projecting X from a m-dimensional space to a n-dimensional space. (e.g. the standard 2D space has basis $[\mathbf{1}, \mathbf{0}]$ $[\mathbf{0}, \mathbf{1}]$–the identity matrix. Therefore, we are essentially projecting the data onto half of the basis of the new space (A_1), and then project those n / 2 dimensions to the corresponding dimensions of all bases $B_1 n/2 \times l$. This gives $Z_1 b \times l$–same dims as without splitting but rank-deficient, so we add up $Z_2 b \times l$ for full rank. Had we split A row-wise, we had to all-reduce before ReLU since it's non-linear (2).

However for the 3rd layer, we must reduce and compute softmax on one device anyway, so we split X column-wise to further save computations. All the layer kernels and input & output & intermediate layer output & NCCL reduce buffers are managed asynchronously, which allows efficient pipe lining when batches of data are continuously fed into the model. We implemented most of the logic in network.cuh, trained the model on MNIST in pytorch, and saved the data and weights as numpy arrays from which we can load and do inference using the CNPY library [4]. We also upgrade naive softmax kernel to the highly optimized online softmax kernel [5]

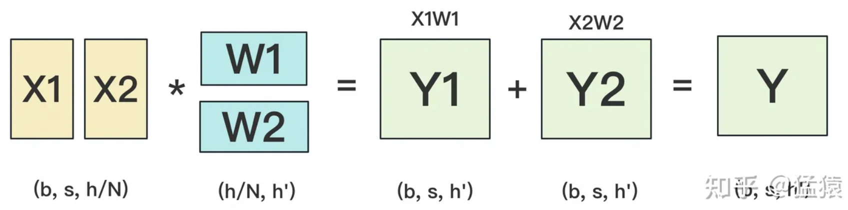

Figure 1: TP recipe 2

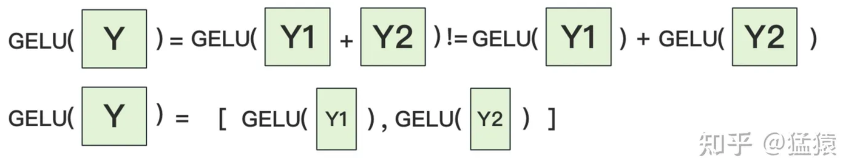
GELU(Y) = GELU(Y1 + Y2) != GELU(Y1) + GELU(Y2)
GELU(Y) = [GELU(Y1), GELU(Y2)]
知乎 @猛猿

Figure 2

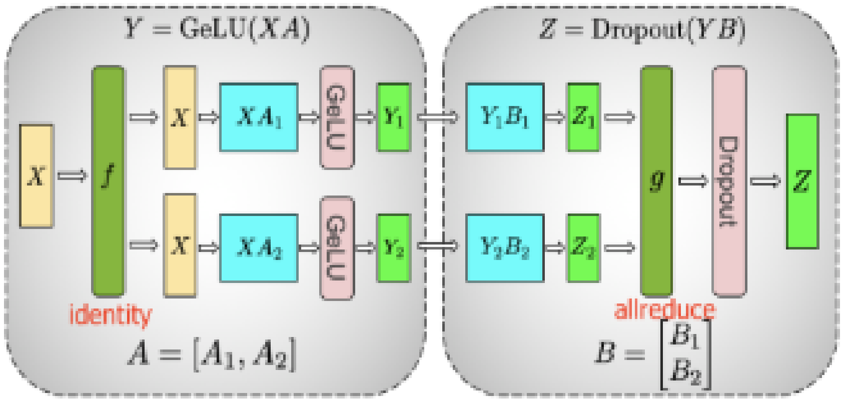

Figure 3: Illustration of tensor parallelism for the MLP component proposed in Megatron-LM. (Image source: Shoeybi et al. 2019)

4 Overview of Results. Demonstration of Project

For the following results, when we compare against a single GPU and multi-GPU setting, we are using the maximum time taken amongst the GPUs in the multiple GPU setting as they are all launched in parallel so taking the maximum time would provide the most generous comparison against a single GPU setting.

4.1 Timming Results

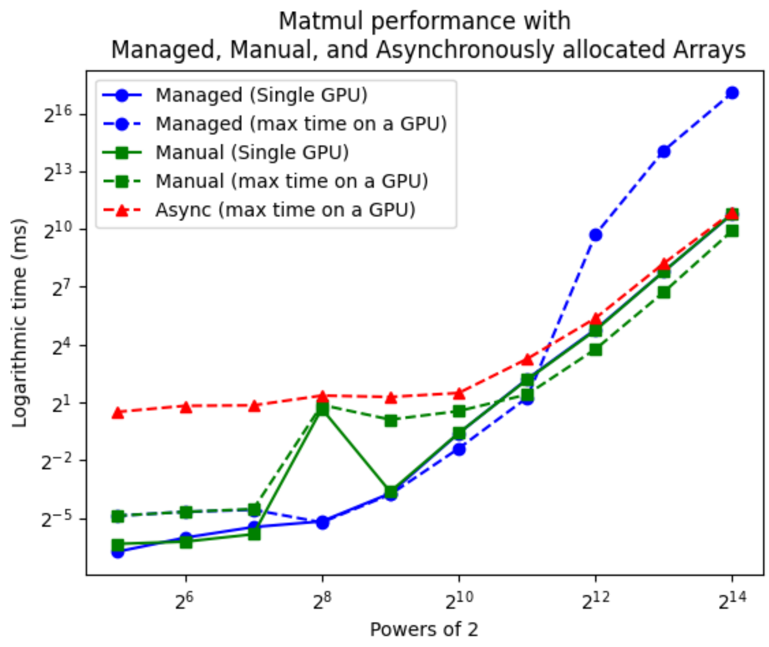

Figure 4: Time comparison of kernels using 1 and 2 GPUs

As can be seen, the maximum time on a single GPU when using multiple GPUs is eventually lower than when the same kernel is called using only one GPU with the same array size. After an array size of 2^{11}, the 2-GPU approach consistently had times that were about half of what the single GPU approach had. The green dotted line represents the maximum time on a GPU when kernels are launched on 2 GPUs simultaneously and achieved the best time performance for large matrices, thus showing the time benefits of splitting computation across multiple GPUs and being consistent with our initial goals.

The async approach also improved for larger matrices and had a lower rate of increase for larger matrices.

The single GPU approach seemed to do better for smaller matrix sizes and we feel this is due to the time cost of launching the kernels overshadowing the actual computation which is negligible for small matrices. For larger matrices this reverses and that we feel matters more since in a real-world setting, large matrix computations would more likely need to be optimised as much as possible.

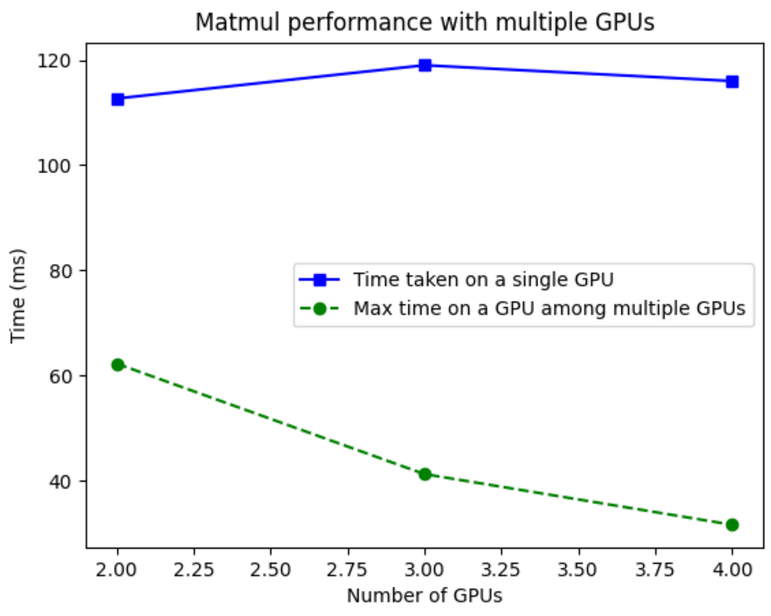

Figure 5: Time comparison of kernels using increasing number of GPUs for square matrices of size 2048 × 2048

As we can see, increasing the number of GPUs also reduces the maximum time taken for a single GPU to complete computation when using manual memory allocation on Euler. Euler did not grant more than 4 GPUs to us in our runs so that was the extent to which we tested it but we imagine a similar trend would hold for more GPUs for matrices as large as the one tested on for the graph.

4.2 FLOP usage

Matrix Size	**Single GPU (GFLOPS/s)**	**2-GPUs (GFLOPS/s)**
32 x 32	5.333	1.939
64 x 64	39.385	13.474
128 x 128	240.941	97.524
256 x 256	21.071	18.351
512 x 512	3338.085	249.12
1024 x 1024	3237.907	1467.47
2048 x 2048	3804.681	6459.53
4096 x 4096	5118.937	10118.662
8192 x 8192	4919.074	10346.57
16384 x 16384	4908.647	9043.583

Table 1: FLOPS comparison between the two approaches

The main thing we wish to highlight here is that for large matrices, we are achieving a higher FLOPS rate which shows more computation is achieved which is the primary goal for machine learning models with large matrix computations.

4.3 Multi-Layer Perceptron

We had trained a toy model separately on PyTorch on the MNIST dataset to obtain high accuracy weights and loaded those weights using the CNPY library [**4**] onto our network on the GPU to perform the forward pass across two GPUs. We were able to create and compile a the weights into the model successfully but faced some runtime issues in our MLP class and highlight this to be a future work we attempt.

5 Deliverables: Building & Running the Project

We've included a Makefile that builds the executables. Please refer to its outputs to execute and test. You can also build manually as below.

Allocation Type	**Compilation Command**
Managed Allocation	`nvcc test_time.cu matmul.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -lcurand -lcublas -o managed`
Manual Allocation	`nvcc test_manual.cu matmul.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -lcurand -lcublas -o manual`
Async Allocation	`nvcc test_async.cu matmul.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -lcurand -lcublas -o async`

Table 2: Compilation Commands for Different Memory Allocation Strategies

Usage for managed and manual (after compilation to your desired output executable name)

```
./executable nRowsA  nColsA  nColsB  num_gpus
```

Usage for async (after compilation to your desired output executable name)

```
./executable nsize  num_gpus
```

Both manual and managed allocations work on Euler. However, Async allocation uses APIs that require an NVLINK and thus we were not able to run them on Euler. Instead we ran them on the Ovli research cluster with A100s.

There is also a bash script named ”test.sh” that you can use to recreate our experimental result for timmings on Euler. Recall that the async version will not work on Euler's instructional node, but the managed and manual versions will.

The main files are the ones mentioned in the compile commands above. The matmul.cu file has the main kernel logic and helper functions. The util.cuh file has additional utility functions. The network.cuh file contains the main logic for the MLP class. The files named as ”test_*.cu” are the files we used for testing various memory allocation styles. The final test files we used are those mentioned in the table.

6 Conclusions and Future Work

We've shown that distributing computations across multiple GPUs is a key way to increase the computations of matrix multiplication and have created a toy example to demonstrate its correctness. There are few key areas we wish to expand on inn future works.

- We aim to perform futher optimisations of existing code. We found lots of timing disparity with 2 GPUs between async and manual matmul depending whether testing on Euler or the Olvi cluster of A100s (see Figure 4) on Euler the manual approach is much faster manual, but when we tested on Olvi, async turned out to be faster.
- We would also like to implement backpropagation using multiple GPUs.
- Due to time limits we haven't benchmarked the MLP in terms of speed and FLOPS, but we will. We will continue to optimize the MLP by combining more paralleism and further overlapping of memory management and compute after the finals. Due to time limits and the complexity of TP MLP, there's still a lot of room for improvement.
- We intend to test distribution on more GPUs (8, 16, etc.) when we fix the timing issue.

References

[1] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, Bryan Catanzaro, *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*, *CoRR*, **abs/1909.08053**, 2019, http://arxiv.org/abs/1909.08053.

[2] 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, Guillaume Lample, *LLaMA: Open and Efficient Foundation Language Models*, https://arxiv.org/abs/2302.13971, arXiv preprint arXiv:2302.13971, Primary Class: cs.CL, Year: 2023.

[3] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, Björn Ommer, *High-Resolution Image Synthesis with Latent Diffusion Models*, https://arxiv.org/abs/2112.10752, arXiv preprint arXiv:2112.10752, Primary Class: cs.CV, Year: 2022.

[4] C. E. Rogers, *cnpy: A Library for Handling Numpy Data in C++*, https://github.com/rogersce/cnpy, Year of access: 2023.

[5] Maxim Milakov and Natalia Gimelshein, *Online normalizer calculation for softmax*, CoRR, Volume: abs/1805.02867, Year: 2018, http://arxiv.org/abs/1805.02867, eprinttype: arXiv, eprint: 1805.02867, Timestamp: Mon, 13 Aug 2018 16:48:49 +0200, BibURL: https://dblp.org/rec/journals/corr/abs-1805-02867.bib