Spring 2020 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Optimizing Einstein Sum Implementation in PyTorch with Specialization, Path Searching, and GPU-Native Contraction

Shawn Zhong
Yuhan Liu
Ziyi Zhang

May 6, 2020

Abstract

In this project, we profiled the current implementation of PyTorch einsum, identified possible reasons that lead to the inefficiency of computation as illustrated by, for example, the differences between computation speed of PyTorch implementations and native ones, and designed and implemented three ways to accelerate the PyTorch einsum including specialization, path optimization, and cuTENSOR based approach.

Link to Final Project `git` repo: https://github.com/ShawnZhong/CS759-Spring-2020-Final-Project

Table of Contents

1. General Information

1. Your home department: Computer Sciences
2. Current status: undergraduate
3. Individuals working on the Final Project (include yourself)
    - Shawn Zhong shawn.zhong@wisc.edu
    - Yuhan Liu yliu738@wisc.edu
    - Ziyi Zhang zzhang765@wisc.edu
4. Choose one of the following three statements (there should be only one statement here):
    - 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. Introduction

We chose to work on this project since we found that the einsum function in PyTorch [1] performs slower than executing the corresponding PyTorch native functions. In this report, we mainly focus on identifying possible reasons why PyTorch einsum is slower than the native functions, and proposing and prototyping different ways to improve PyTorch einsum function.

2.1 Einstein Summation Notation

Einstein summation convention [2] is a notational convention that implies summation over a set of indexed terms in a formula introduced by Albert Einstein. Using the Einstein summation convention, many multi-dimensional, linear algebraic array operations can be represented in a simple fashion and then calculated. We present some examples here.

1. Outer product of vectors `x` and `y` of length n

```
z = torch.einsum('i,j->ij', x, y)
```

$$z_{ij} = x_i y_j \Leftrightarrow z = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \cdots & x_m y_n \end{bmatrix}$$

2. Element-wise multiplication of matrix A of shape $m \times n$ and matrix B of shape $m \times n$

   ```
   C = torch.einsum("ij,ij->ij", A, B)
   ```

   $$C_{ij} = A_{ij} B_{ij} \Leftrightarrow C = \begin{bmatrix} A_{11}B_{11} & \cdots & A_{1n}B_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1}B_{m1} & \cdots & A_{mn}B_{mn} \end{bmatrix}$$

3. Matrix multiplication of matrix A of shape $m \times n$ and matrix B of shape $n \times p$

   ```
   C = torch.einsum('ij,jk->ik', A, B)
   ```

   $$C_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk} \Leftrightarrow C = \begin{bmatrix} A_{11}B_{11} + \cdots + A_{1n}B_{n1} & \cdots & A_{11}B_{1p} + \cdots + A_{1n}B_{np} \\ \vdots & \ddots & \vdots \\ A_{m1}B_{11} + \cdots + A_{mn}B_{n1} & \cdots & A_{m1}B_{1p} + \cdots + A_{mn}B_{np} \end{bmatrix}$$

2.2 PyTorch Implementation

PyTorch implements `einsum` by two stages, input parsing and result calculation. The detailed process of how einsum calculates result can be illustrated by the following example:

Consider the operation `D = torch.einsum("acbk,afek,ace->bf", A, B, C)`, where A, B, C are tensors with dimension $n \times n$. An `einsum` call will first parse the input and then start merging the input from left to right. At each iteration, it merges the current result (which corresponds to a prefix or the operands list) with the next operands in the list.

A single merge will take two tensors, for example, A (`"acbk"`) and B (`"afek"`), and merge them into an intermediate five-dimension tensor M (`"abcfe"`). The five dimensions `"a"`, `"c"`, `"b"`, `"f"`, `"e"` either come from the first tensor or the second. Note that the dimension `"k"` is eliminated from the dimension after calculation because it is no longer needed in the future operation.

As a side-note, `torch.einsum` uses batch matrix multiplication to calculate the results. Consider the previous example, we have dimension `"a"` as the dimension that appears in both operands, `"b"` and `"c"` only on the left, `"fe"` only on the right, and `"k"` merged. As a result, we can see the merge operation of A (`"acbk"`) and B (`"afek"`) as a batch matrix multiplication of a tensor of size (`"a"`, `"bc"`, `"k"`) and another tensor of size (`"a"`, `"k"`, `"fe"`).

The pseudocode for `D = torch.einsum("acbk,afek,ace->bf", A, B, C)` is shown below:

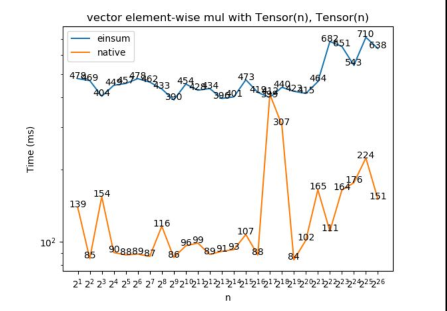
```
A'←reshape(A, "a, bc, k")   # A' is a tensor of shape (a, b * c, k)
B'←reshape(B, "a, k, fe")   # B' is a tensor of shape (a, k, f * e)
M ←bmm(A', B')              # M  is a tensor of shape (a, b * c, f * e)
```

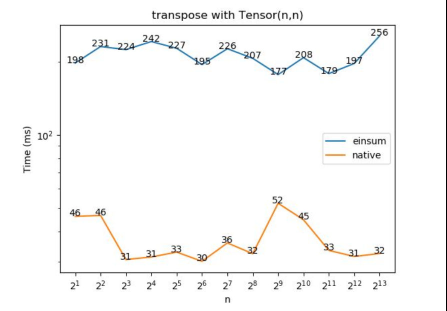
```
M'←reshape(M, "1, bf, ace") # M' is a tensor of shape (1, b * f, a * c * e)
C'←reshape(C, "1, ace, 1")  # C' is a tensor of shape (1, a * c * e, 1)
D'←bmm(M', C')              # D  is a tensor of shape (1, b * f, 1)
D ←reshape(D, "bf")         # D  is a tensor of shape (b * f)
```

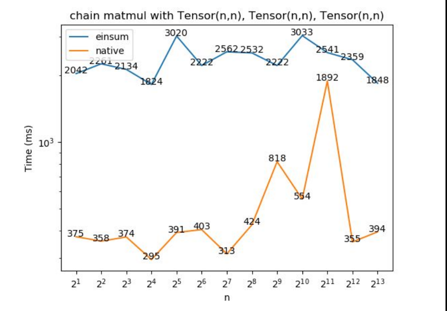
3. Problem Statement

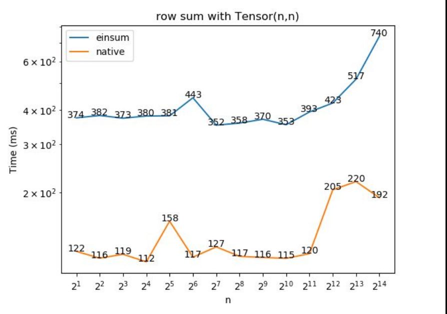
3.1 Comparison with Native Operations

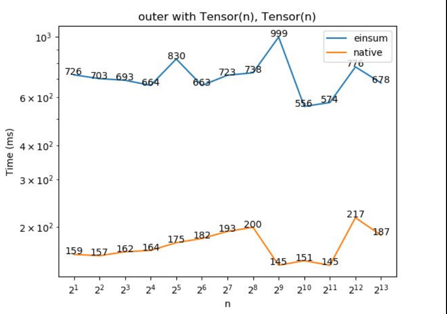
Using PyTorch einsum to perform tensor operations is much slower than manually performing tensor operations. For preliminary benchmark, we compared the time used by `torch.einsum` vs the corresponding native function. The result is attached in the table below:

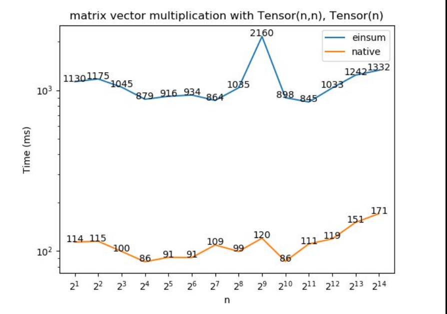


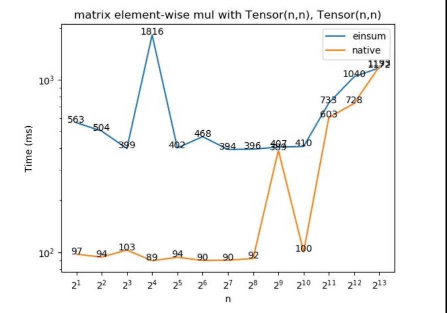


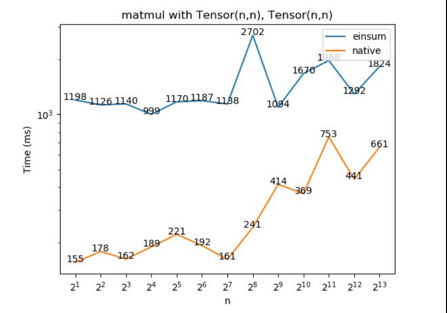


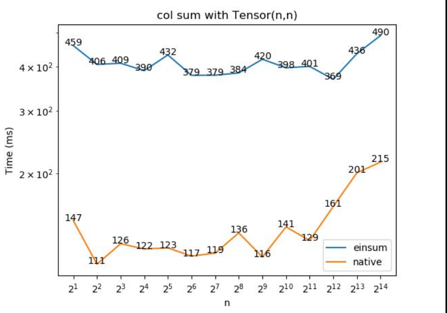












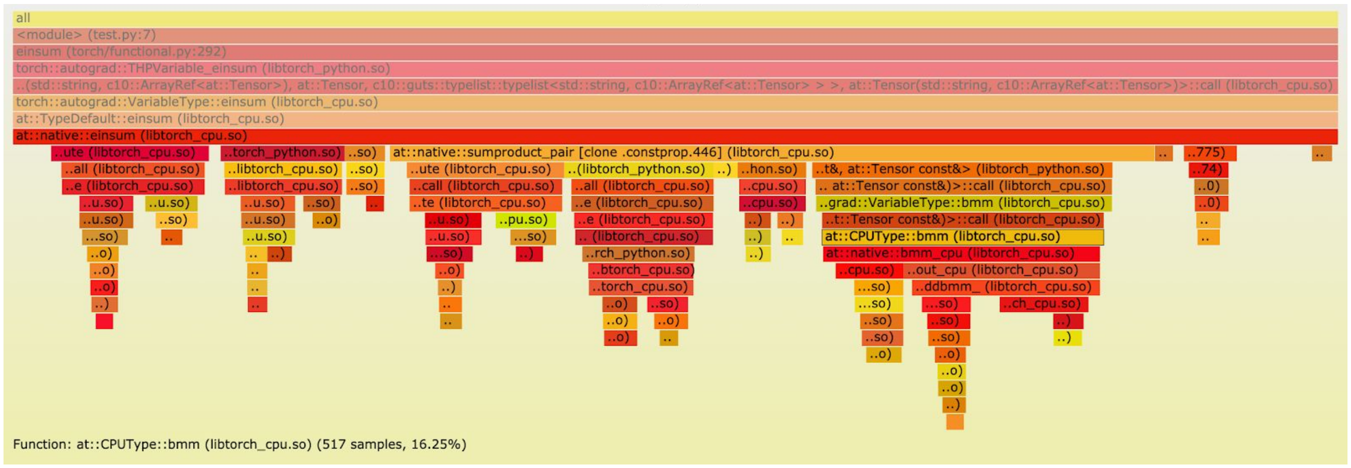
We expect that the execution time of PyTorch einsum should not be longer than performing PyTorch native functions.

3.2 Algorithmic Complexity

On the other hand, the order that Pytorch einsum merges the tensor list is not always the optimal way in regard to running time, which will lead to a very large intermediate. Consider one example. Suppose `torch.einsum("acb,afe,ace->bf", [A, B, C])`. Suppose all three tensors have dimension (n, n, n). the calculation complexity of the first merge is $O(n^5)$, and the second merge will cost the same. Consider another equivalent operation `torch.einsum("acb,ace,afe->bf", [A, C, B])`. The first merge will have calculation complexity $O(n^4)$ and the subsequent merge will also have the same complexity, so the total complexity will be $O(n^4)$, which is better than the original order. Therefore, we want to

3.3 Batch Matrix Multiplication Overhead

We identified the problem of PyTorch `einsum` as: for tasks other than batch matrix multiplication, spending time on reshaping, executing batch matrix multiplication, and permuting introduce overhead to the computation compared with executing specific PyTorch native functions that accomplish the tasks. On the flame graph below, we can see that only 16.25% of the execution time is spent on batch matrix multiplication, and the rest is used by preprocessing.



4. Solution Description

4.1 Specialization

We do pattern matching on the einsum string, and pre-define a mapping between Pytorch native functions and the parsed string. If the parsed string can be mapped to a predefined native function, our specialized einsum function will just call the corresponding native function which is faster than the more general approach using batched matrix multiplication in einsum.

This approach helps us prototype the solution of calling corresponding PyTorch native functions so as to reduce the gap between using native functions and PyTorch einsum.

4.2 Path Optimization

An intuitive approach to find the optimal order to merge the results, is to try all possible orders and find the time complexity of them. This approach is given in the function `find_opt_path_brute_force`. However, the time complexity of this approach is $O(n!)$, where n is the number of operands in the list. This implies that this approach will only work for n up to 8, which is sufficient for einsum in most of the cases.

For a strong support of finding the optimal path, we implemented a dynamic programming approach which has a complexity of $O(2^n n)$ [5], and work for n up to 18. The idea is that for each subset of operands S, we calculate the worst scale to merge those together in all possible orders. We enumerate from smaller subsets to larger ones and keep new values to the larger subsets. Then the final answer is $dp[S_{all}]$ where S_{all} is the set of all operands, and we can do backtracking to find the order.

4.3 GPU-Native Contraction

In the existing implementation, permute and reshape is called on the input tensors to make the data fit the requirement for batch matrix multiplication. The preprocessing stage takes a considerable amount of time in einsum, and also creates a copy of the tensor, which occupies the precious GPU memory. In addition, batch matrix multiplication is called multiple times, adding the overhead of launching kernel functions.

cuTENSOR [4] is a CUDA library provided by Nvidia that supports tensor reduction, contraction, and element-wise operations. cuTENSOR supports tensor contraction of the following form.

$$D_{modes_D} \leftarrow \alpha A_{modes_A} B_{modes_B} + \beta C_{modes_C}$$

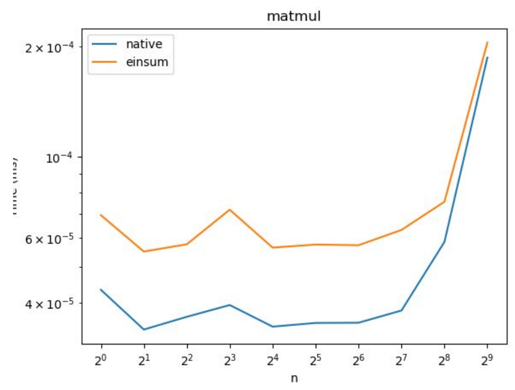
We used it as a replacement for batch matrix multiplication so that we can perform einsum with minimum preprocessing. A code snippet for computing `torch.einsum("abcd,aefd->aefbc", A, B)` using cuTENSOR is shown below:

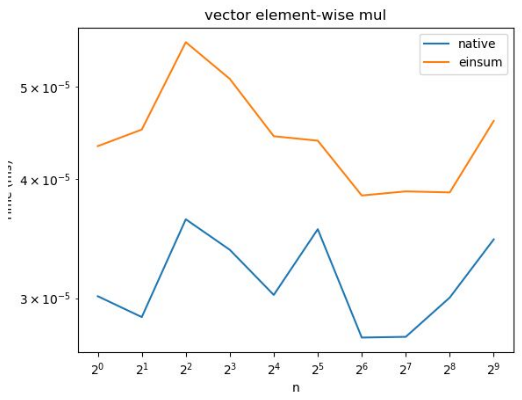
```
// Initialize context and tensors
cutensorInit(&handle);
cutensorInitTensorDescriptor(&handle, &descA, ... );
cutensorGetAlignmentRequirement(&handle, dA, &descA, &alignmentRequirementA);
// Create contraction descriptor
cutensorInitContractionDescriptor(
    &handle, &desc,
    &descA, { 'a', 'b', 'c', 'd' },       alignmentRequirementA,
    &descB, { 'a', 'e', 'f', 'd' },       alignmentRequirementB,
    &descC, { 'a', 'e', 'f', 'b' , 'c' }, alignmentRequirementC,
    &descC, { 'a', 'e', 'f', 'b' , 'c' }, alignmentRequirementC,
    CUTENSOR_R_MIN_F32);
// Set the algorithm and plan for execution
cutensorInitContractionFind(&handle, &find, ... );
cutensorInitContractionPlan(&handle, &plan, &desc, &find, ... );
// Perform contraction
cutensorContraction(
    &handle, &plan,
    &alpha, dA, dB,
    &beta,  dC, dC,
    workspace, workspaceSize, stream);
```

5. Overview of Results

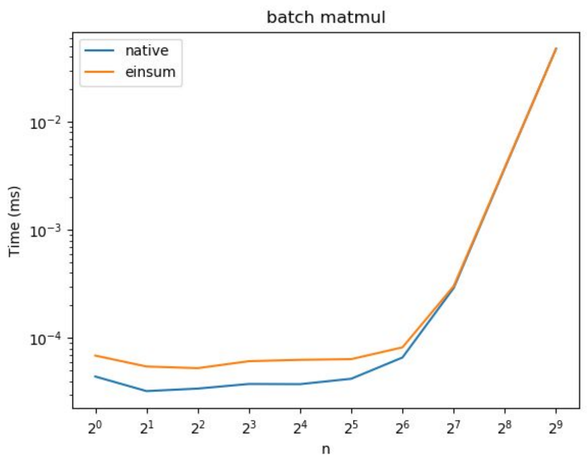
5.1 Specialization

We implemented pattern matching on the input string, and created a mapping between parsed results and native functions. For example, we parsed the input of `"ij,jk->ik"` into `"ab,bc->ac"`, which is then mapped to the torch.matmul function. We experimented the design by first parsing the einsum string and then calling the specific native function by searching the mapped table. We observed that our specialization approach actually accelerates the einsum computation. One possible reason is that parsing and forming inputs to batch matrix multiplication takes more time than directly passing inputs to the specific Pytorch native functions. We present some of the comparisons between our specialization approach and torch.einsum on GPU tensors here.





Furthermore, from the follow plot, we saw that both specialization and torch.einsum perform nearly the same on batch matrix multiplication, which further illustrates the fact that preparing the inputs to different kinds of operands (e.g., matrix transpose, matrix multiplication, vector dot product, etc.) into the format required by batch matrix multiplication (e.g., reshape, permute, etc.) takes considerable amount of time, which introduces the gap between passing inputs into Pytorch native functions and calling Pytorch einsum. However, if we are doing batch matrix multiplication, the input tensors are already prepared into the input to the bmm operand, which saves the cost of reshaping and permuting, so we are getting nearly the same cost.

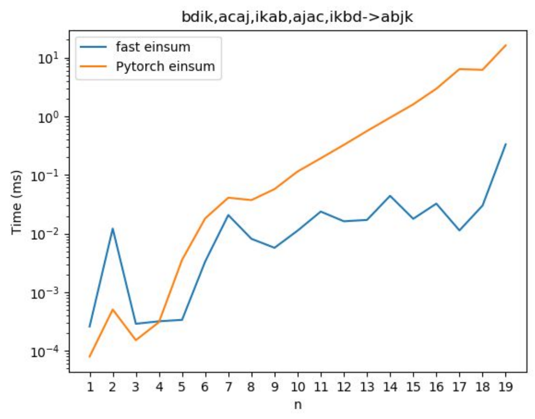


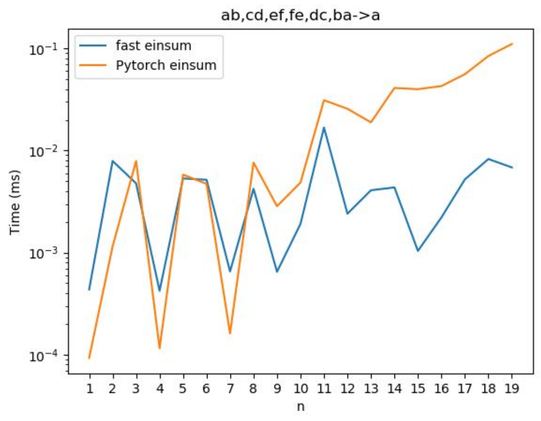
One possible future direction is to create a more comprehensive mapping between input string and specific native functions.

5.2 Path Optimization

We used a bitmask dynamic programming to implement the process of finding the optimal path, and then recover the path from the terminal dynamic programming states.

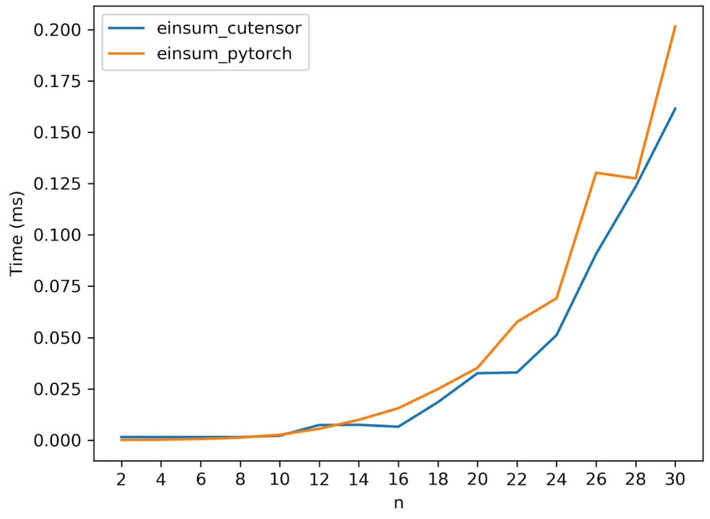
We did our benchmark on several different equations (the results of two of them are shown below) We can observe that when the tensor sizes are small, the two different approaches have similar performance. However, when the size of the tensors goes larger, the advantages of path optimization will get greater as shown below.





5.3 GPU-Native Contraction

We use `"abcd,aefd->aefbc"` as an example for the benchmark. We randomly generate two tensors of shape (a, n, n, n) as input where $a = 200$ is the batch size, and vary n to measure the time.



As we can see, the execution time for cuTENSOR remains is shorter than the PyTorch implementation for $n \geq 15$.

6. Deliverables

The repository can be found at: https://github.com/ShawnZhong/CS759-Spring-2020-Final-Project

Commands to run the optimized versions of einsum:

- `python -m speclication.main`

- `python -m path_opt.main`
- `python -m cutensor.main`

We also include plots for benchmarking in `benchmark/results`, and comparisons between specialization approach and Pytorch einsum approach in `specialization/results` , and comparisons between path optimization einsum and Pytorch einsum approach in `path_opt/results`.

7. Conclusions and Future Work

7.1 Conclusions

Currently, we prototyped the possible solutions that help improve Pytorch einsum, and proved that our proposals actually accelerate the computation. To sum up, we tried:

- Specializing Einsum operation by parsing einsum input string and invoking the pre-mapped Pytorch native functions so as to avoid the overhead of preparing input to and performing Batch Matrix Multiplication
- Conducting path optimization (reordering the operations) by performing exhaustive search or Dynamic programming to get better asymptotic complexity
- Using cuTENSOR contradiction to reduce the preprocessing overhead involved in Pytorch einsum

7.2 Future Work

Our overall goal is to incorporate our proposals into Pytorch einsum implementation. To be more specific, we will improve our current designs in the following aspects:

- Specialization: Include a more comprehensive mapping between parsed input string and PyTorch native functions.
- Path optimization: Other than reordering the operation and merging them one by one, we can possibly merge them in a tree-like fashion to get better asymptotic behavior.
- cuTENSOR: Integrate cuTENSOR based einsum into Pytorch einsum implementation.

References

[1] Paszke, Adam, et al. "PyTorch: An imperative style, high-performance deep learning library." Advances in Neural Information Processing Systems. 2019.

[2] Wikipedia contributors. (2020, February 9). Einstein notation. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Einstein_notation&oldid=939837748

[3] Daniel G. A. Smith and Johnnie Gray, opt_einsum - A Python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, *2018*, 3(26), 753. DOI: https://doi.org/10.21105/joss.00753

[4] Andrew Kerr and Paul Springer, cuTENSOR: High-performance Tensor Operations in CUDA. GTC *Silicon Valley-2019*. https://developer.nvidia.com/gtc/2019/video/S9593

[5] Pfeifer, Robert NC, Jutho Haegeman, and Frank Verstraete. "Faster identification of optimal contraction sequences for tensor networks." *Physical Review E* 90.3 (2014): 033315. Retrieved from https://arxiv.org/abs/1304.6112