Fast Einsum Implementation Using Parallelization, Vectorization, and More

Algorithm Engineering 2025 Project Paper

Jonas Peters
Friedrich Schiller University Jena
Germany
jonas.peters@uni-jena.de

Abstract

(1) **Topic and background:** The topic of this paper is tensor transformations, and how to perform them efficiently. Tensors consist of many dimensions. Take matrices as specialized tensors with two dimensions (rows and columns) for example. When multiplying two matrices, we calculate "row times column" for each position in the resulting matrix. In tensor language, this is equivalent to *contracting* the the second dimension of the first matrix with the first dimension of the second matrix. We can also just rearrange the dimensions in a new order, which is called *transposing*, similar to matrices. Lastly, we can *reduce* entire dimensions by summing over all entries, for example reducing the second dimension of a matrix M is equivalent to

$$M_{reduced} = M \cdot \vec{1} \quad .$$

In order to not always hardcode these operations, *einsum* can be employed by passing in the two operand matrices, together with a format string, which defines how every dimension will be treated. For example `ik,kj->ij` specifies a normal matrix-matrix-multiplication. Here is how these *column identifiers* of `lhs_string` in the eisnum expression `<lhs_string>,<rhs_string>-><target_string>` are interpreted:

Table 1: Column Identifiers of Einsum Expression

Category	rhs_string	target_string
Batch	Yes	Yes
Kept left	No	Yes
Contracted	Yes	No
Summed left	No	No

You have to read the table in the following manner. Take a column identifier of `lhs_string`. Then you check whether it is present in `rhs_string` and `target_string`, and based on these results you assign it to one of the four specified categories, where "Summed left" describes a dimension to reduce.
A similar table can be produced for column identifiers in `rhs_string`. The column identifiers in `target_string` also specify the resulting shape of the operation. All of the necessary computations to achieve the desired transformation is performed by the einsum function.

(2) **Focus:** What is your research question? What are you studying precisely?
(3) **Method:** What did you do?
(4) **Key findings:** What did you discover?
(5) **Conclusions or implications:** What do these findings mean? What broader issues do they speak to?

Keywords

entity resolution, data cleansing, programming contest

1 Introduction

1.1 Background

1.2 Related Work

I used this [1] C++ library

1.3 Our Contributions

1.4 Outline

2 The Algorithm

2.1 Internal Representation of Mock Labels

2.2 Efficient Preprocessing of Input Data

The following findings are important to speed up preprocessing of the input data:

- Reading many small files concurrently, with multiple threads (compared to a single thread), takes advantage of the internal parallelism of SSDs and thus leads to higher throughput [**?**].
- C-string manipulation functions are often significantly faster than their C++ pendants. For example, locating substrings with `strstr` is around five times faster than using the C++ `std::string` function `find`.
- Hardcoding regular expressions with *while, for, switch* or *if-else* statements results in faster execution times than using standard RegEx libraries, where regular expressions are compiled at runtime into state machines.
- Changing strings in place, instead of treating them as immutable objects, eliminates allocation and copying overhead.

3 Experiments

Table 2 shows the running times of the resolution step of the five best placed teams.

Table 2: Comparison of the F-measure and the running times of the resolution step of the five best placed teams. The input data for the resolution step consisted of 29,787 in JSON formatted e-commerce websites. Measurements were taken on a laptop running Ubuntu 19.04 with 16 GB of RAM and two Intel Core i5-4310U CPUs. The underlying SSD was a 500 GB 860 EVO mSATA. We cleared the page cache, dentries, and inodes before each run to avoid reading the input data from RAM instead of the SSD.

Team	Language	F-measure	Running time (s)
PictureMe (**this paper**)	C++	0.99	**0.61**
DBGroup@UniMoRe	Python	0.99	10.65
DBGroup@SUSTech	C++	0.99	22.13
eats_shoots_and_leaves	Python	0.99	28.66
DBTHU	Python	0.99	63.21

4 Conclusions

References

[1] Paul Springer, Tong Su, and Paolo Bientinesi. 2017. HPTT: A High-Performance Tensor Transposition C++ Library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Barcelona, Spain) *(ARRAY 2017)*. ACM, New York, NY, USA, 56–62. https://doi.org/10.1145/3091966.3091968