WebTensor: Towards high-performance raster data analysis in the browser

Lucas Fabian Naumann[1][2]

Abstract: We present WebTensor, a chunked tensor implementation for WebAssembly (Wasm) compiled from a self-written C++ library and designed to efficiently analyze raster data directly in the browser. WebTensor allows loading (chunked) data from various backends, manipulating it by aggregations and forwarding computed results in a zero-copy manner to JavaScript so that they can be further processed or visualized. We demonstrate the performance of WebTensor by benchmarking data access and aggregation operations and compare it against a JavaScript version compiled from the same C++ code.

Keywords: WebAssembly; Raster Data; Tensor Processing; Visual Analytics

1 Introduction

With climate change research becoming increasingly important in the last years, so are raster datasets used in it, for example, from the Copernicus project[3] or the MOSAiC expedition[4]. Analysis of raster data is often done using visual analytics, a method where domain experts analyze the data with interactive visualization and exploration tools [Cu19]. Easy and platform-independent access to such tools could be realized by a browser application that is able to load the desired datasets and process them. However, such applications were not feasible in the past, as processing the data in the browser with JavaScript would be too inefficient due to performance limitations of the language, and doing the processing on the server side instead would introduce a too large overhead regarding requesting and receiving data [LH14]. This infeasibility changed when WebAssembly (Wasm), a binary instruction format for a virtual machine, was launched in 2017 [Ha17]. With being supported by most browser engines and achieving a performance comparable to those of languages like C++ [Ja19], it is suited for high-performance data analysis in the browser.

So far, only a few data processing tools utilizing Wasm have been proposed, like the embeddable SQL database DuckDB-Wasm [Ko22] or a Wasm backend for TensorFlow.js[5]. None of those tools is suitable for analyzing raster datasets. Existing tensor implementations for browsers like TensorFlow.js focus on machine learning and thus lack features needed for analysis tasks. For example, tensors should be chunked to perform aggregations needed for the analysis efficiently and, as the data typically originates from a multitude of different sensors, data of varying backends, types and layouts should be processable in a single tensor.

[1] Technische Universität Dresden, lucas_fabian.naumann@mailbox.tu-dresden.de
[2] German Aerospace Center, Institute of Data Science
[3] https://copernicus.eu
[4] https://mosaic-expedition.org
[5] https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html

doi:10.18420/BTW2023-75

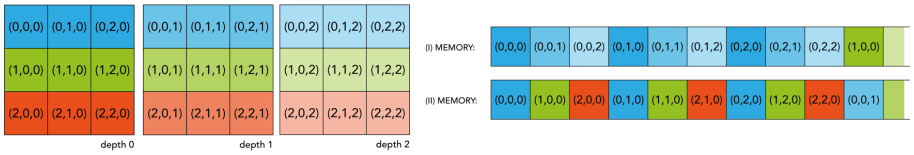


Fig. 1: Memory layouts for a three-dimensional tensor: (I) row-major order, (II) column-major order (row-/column-major order indicate data along the last/first dimension to be contiguous).

In this paper, we present WebTensor, a chunked tensor implementation for Wasm designed for raster data analysis and intended to serve as a backend for JavaScript programs. Furthermore, we evaluate the performance of WebTensor on various data access and aggregation operations and compare the results against an equivalent tensor implementation in JavaScript.

2 Background

Tensor Data Processing Multidimensional data is often represented as a datacube, which, in its most basic form, is a tensor. In order to store multidimensional data, it is mapped onto the one-dimensional index space of storage devices (cf. Figure 1). The resulting memory layout has a significant impact on I/O performance. Due to the performance implications of data locality, multidimensional data is commonly chunked to reduce latency when the data access pattern might change over time[6]. Analyzing such data requires two kinds of queries: data accesses and aggregations. Data is accessed either at a single index of the tensor or along index ranges per dimension. The latter one is commonly called dicing or, if the index is static for one dimension, slicing. Aggregation reduces data along selected dimensions by applying a numeric operation. Consider, for example, a four-dimensional tensor with three spatial and one temporal dimension. Aggregating over the temporal dimension by taking a minimum, results in a three-dimensional tensor containing the minimum value over time at all spatial locations.

WebAssembly & Compilation Toolchain In the past, JavaScript has been the only programming language natively supported in browsers. Consequently, developing applications for the web required its usage, restricting the feasibility of computationally intensive applications because of its limited performance [Ha17]. In order to overcome this issue, Wasm, a binary instruction format for a virtual machine usable in browsers, was introduced as a compilation target for high-level languages like C++ [Ha17]. Jangda et al. showed that Wasm is not only faster than JavaScript but even comparable to code executed natively on x86 [Ja19]. Currently, the Wasm heap is limited to 4 GiB in size as a 32-bit addressing space model is used [Ha17]. Furthermore, the interaction between JavaScript and Wasm is one-sided since the Wasm heap can be accessed by JavaScript but accessing JavaScript memory by Wasm is not possible.

[6] https://www.unidata.ucar.edu/blogs/developer/en//entry/chunking_data_why_it_matters

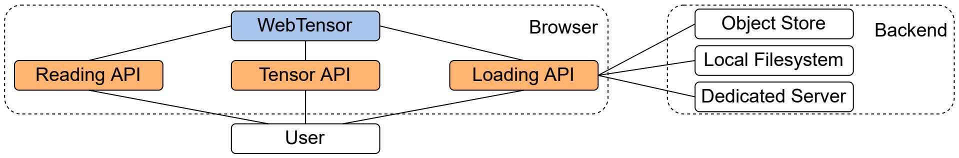


Fig. 2: Architecture of WebTensor (orange: JavaScript components, blue: Wasm components).

There are two major toolchains for compiling C++ code to Wasm: Cheerp[7] (commercial) and Emscripten[8] (non-commercial, open source), with Emscripten achieving a better performance as reported by Yan et al. [Ya21]. Once installed, the Emscripten compiler frontend `em++` can be used as a drop-in replacement for regular C++ compilers. For the compilation process, Emscripten uses Clang and LLVM. Additionally to compiling to Wasm, Emscripten allows to compile C++ to JavaScript.

3 Raster Data Analysis in the Browser

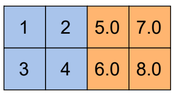
Figure 2 depicts an overview of WebTensor and subsequent components (JavaScript components in orange, Wasm components in blue). We compiled WebTensor from a self-written C++ library to Wasm using Emscripten. JavaScript programs can interact with it by utilizing three APIs, which were initially written in C++ and then compiled to Wasm and bound to JavaScript methods using Embind[9]. These APIs enable WebTensor to be used straightforwardly in JavaScript programs and allow arbitrary post-processing or visualization of tensor data, making it a flexible tool for various applications.

3.1 WebTensor

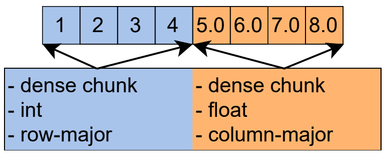
Memory Layout To maximize spatial data access locality, WebTensor stores raw data in a binary buffer with data of fixed-sized rectangular chunks lying contiguously in it. For accessing the values corresponding to the bytes stored in this buffer, the tensor stores chunk objects, each one having pointers to one of those contiguous blocks and additional metadata, e.g., the chunk type, the value type of the data and the internal memory layout (cf. Figure 3). This decoupling of raw data and metadata has additionally the advantage that different chunk types (e.g., dense and sparse chunks), varying data types (e.g., float and int) and arbitrary internal memory layouts (e.g., row- and column-major) can be freely combined in a single tensor.

Data Access As the user should not have to care about the internal memory layout of WebTensor, accessing its data is done by specifying indices regarding its dimensions, which are then transformed into offsets within the binary buffer. Such a transformation requires using the chunks as only these contain necessary metadata, like the type of the stored values. Since all chunks have a fixed shape, the one containing a user-provided index can easily be obtained and, with that, also an offset to the first element in it. In a second step, the metadata of the chunk can be used to determine the offset from its beginning.

[7] https://leaningtech.com/cheerp
[8] https://emscripten.org
[9] https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html



(a) Two-dimensional data.



(b) Buffer-Chunk structure of WebTensor.

Fig. 3: Chunked data representation in WebTensor (orange and blue color mark different chunks).

Features WebTensor provides methods for accessing data at single indices as well as for dicing and slicing operations. Slicing and dicing operations thereby only return a view to existing data, which offers the same functionalities as a tensor regarding data access and aggregations and can be materialized to a new independent tensor at a later stage. Regarding aggregates, the computation of basic statistics, i.e., minimum, maximum, mean and standard deviation, on arbitrary dimensions is supported. Furthermore, it is possible to rechunk a tensor, thereby changing the in-memory order of its data, and thus favor access patterns and aggregations over specific dimensions.

3.2 APIs

Loading The Loading API enables JavaScript programs to load data from various backends to WebTensor. For this, parts of Apache Arrow[10] (compiled to Wasm) are used to process data in the Arrow IPC and Parquet format. Since Wasm programs cannot access JavaScript memory, data should be loaded directly from the backend onto the Wasm heap to avoid unnecessary copies. Currently, WebTensor offers loading data in this way from a dedicated server using a WebSocket connection and the IPC format for serialization. Loading data in the Parquet format from object stores or the local file system is also possible, but at the moment, only by loading the data first in JavaScript and then copying it to Wasm using the API, thus having an increased overhead.

Tensor This API binds the slicing, dicing and materializing, as well as the aggregate functions of WebTensor to JavaScript methods using Emscriptens Embind. Hence, it enables the manipulation of tensor data and the construction of new views and tensors from the JavaScript side.

Reading Using this API, WebTensor data can be accessed from JavaScript programs, where further processing or visualization (e.g., with the library D3[11]) is possible. Accessing the data is done by first obtaining its begin and end addresses in the corresponding chunks, as well as information about the chunk types, data types and memory layouts. Then, zero-copy, typed views of the data on the Wasm heap are created with this information and returned to the JavaScript side through the API.

[10] https://arrow.apache.org
[11] https://d3js.org/

Dice Shape	Wasm [ms]	JS [ms]
[400, 20, 20, 20]	11.01	27.41
[20, 20, 20, 20]	1.34	2.78
[20, 30, 72, 72]	1.00	1.55
[20, 30, 20, 72]	0.94	1.49

Tab. 1: Runtimes for dicing varying shapes.

Aggregated Dimensions	Wasm [ms]	JS [ms]
time, alt, lat, lon	444.4	2439.0
time, alt, lat	1136.4	2439.0
time, alt	980.4	2777.8
time	574.7	3030.3

Tab. 2: Runtimes for aggregating the minimum over varying dimensions.

4 Experimental Evaluation

Setup & Methodology We compare WebTensor against a JavaScript baseline implementation, compiled from the same C++ code using Emscripten, for various data access and aggregation operations. As baseline to compare against, we use a compiler-generated JavaScript implementation as it has been shown to consistently outperform equivalent manual implementations [Ya21]. We executed all benchmarks with benchmark.js[12] on a machine with an Intel i5-6440HQ CPU @2.6 GHz and 32 GiB RAM using a Firefox browser (version 107.0). As noted before by Yan et al., the specified optimization options for the compilation of the Wasm and JavaScript code sometimes show unexpected behaviour, e.g., building with -O1 results in more efficient code than with -O3 [Ya21]. We used, in all cases, the -Os optimization flag, as it led to consistently good performance results.

Dataset For our evaluation, we use a space weather dataset provided by the German Aerospace Center. This dataset has four dimensions: time, altitude (alt), longitude (lon) and latitude (lat), along with multiple variables. The data is organized in row-major order, i.e., the values for varying latitudes lie contiguously in memory. With about 13 GB, the dataset is too large to be processed at once in Wasm with its 32-bit addressing space model. It is planned to support processing datasets larger than 4 GiB by implementing a lazy loading strategy for dataset chunks and replacing the least recently used one when no further memory is available. However, this has not been implemented yet. To still show the performance of WebTensor, we restrict ourselves to a data size of 250 MB by only regarding a [400,30,72,72]-shaped region of the dataset and one of its 32-bit floating point variables. The original dataset is not chunked, but for the benchmarks, we rechunk the data into 12 chunks of shape [100,30,40,40] (all dense chunks, with row-major order and floats as value type) having a size of approximately 21 MB.

Benchmarks First, we measured the access times for single indices. For this, we generated 100 random indices, measured their mean access times individually, and afterwards, took the mean over the 100 values received in this way. The resulting mean access time amounts to $1.35 \cdot 10^{-2}$ ms for Wasm and $1.63 \cdot 10^{-2}$ ms for JavaScript, with a standard error of 0.1% for both.

[12] https://benchmarkjs.com

After the access of single values, we evaluated the performance of dicing operations. We did this by specifying four fixed dice shapes and, in a similar fashion as before, randomly generated 100 concrete dices (with varying start and end points) for each of those shapes, measured their mean runtimes individually and then computed the mean of the resulting 100 values. The received runtimes are shown in Table 1, the standard error is omitted in the table as it amounted to less than 0.16% for all shapes and is thus neglectable.

Lastly, we measured the execution times of aggregation operations. Table 2 shows the execution times for aggregating the minimum over the tensor dimensions specified by the "Aggregated Dimensions" column of the table. Again, the standard error is with at most 1.6% neglectable and not shown. The results for computing the maximum, mean and standard deviation are similar, hence we omit them here due to lack of space.

Discussion The Wasm version of WebTensor always achieves better results than its JavaScript counterpart. For point data access, Wasm outperforms JavaScript by 21%, and for dicing and aggregating, it is, on average, faster by 92% and 294%, respectively. The performances differ thereby not by a constant factor but vary. Furthermore, while the results for the Wasm and JavaScript columns in Table 1 are expected due to data locality and the number of operations to be performed, this is only the case for the Wasm values in Table 2 and not for the ones of JavaScript. As the number of aggregated dimensions in Table 2 decreases from top to bottom, so does the number of operations needed to aggregate over them. Thus, the runtimes are expected to decrease too, besides when aggregating over all dimensions, as in this case, the data to be considered lies contiguously in memory, allowing optimizations. This expected behaviour can be observed for Wasm. Regarding JavaScript, however, equal runtimes were obtained when aggregating over all four dimensions as when only considering time, altitude and latitude. For fewer dimensions, the runtimes increased even further. Currently, no satisfying explanation for this discrepancy could be found, but as the same observation was made in multiple repetitions of the benchmarks, it should be subjected to further studies in the future.

5 Summary & Next Steps

We presented WebTensor, a chunked tensor implementation for Wasm capable of efficient raster data analysis in the browser. Our initial experimental results indicate that a Wasm-based tensor implementation can significantly outperform comparable JavaScript implementations on raster data access and aggregation operations. As a next step, we plan to extend WebTensor such that larger datasets and more complex data analysis tasks become possible. More specifically, we plan to implement lazy loading of chunks from the backend such that complete datasets can be analyzed, as well as loading data from other backends without making unnecessary copies. Additionally, we intend to implement more aggregation operations (e.g., resampling, computing histograms) and a data cube layer on top of WebTensor to provide more metadata information (e.g., physical coordinates of tensor indices).

Bibliography

[Cu19] Cui, Wenqiang: Visual Analytics: A Comprehensive Overview. IEEE Access, 7, 2019.

[Ha17] Haas, Andreas; Rossberg, Andreas; Schuff, Derek L.; Titzer, Ben L.; Holman, Michael; Gohman, Dan; Wagner, Luke; Zakai, Alon; Bastien, JF: Bringing the Web up to Speed with WebAssembly. SIGPLAN Not., 52(6), 2017.

[Ja19] Jangda, Abhinav; Powers, Bobby; Guha, Arjun; Berger, Emery D.: Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code. CoRR, abs/1901.09056, 2019.

[Ko22] Kohn, André; Moritz, Dominik; Raasveldt, Mark; Mühleisen, Hannes; Neumann, Thomas: DuckDB-Wasm: Fast Analytical Processing for the Web. Proc. VLDB Endow., 15(12), 2022.

[LH14] Liu, Zhicheng; Heer, Jeffrey: The Effects of Interactive Latency on Exploratory Visual Analysis. IEEE Transactions on Visualization and Computer Graphics, 20(12), 2014.

[Ya21] Yan, Yutian; Tu, Tengfei; Zhao, Lijian; Zhou, Yuchen; Wang, Weihang: Understanding the Performance of Webassembly Applications. In: Proceedings of the IMC'21. 2021.