Fast Higher-Order Functions for Tensor Calculus with Tensors and Subtensors

Cem Bassoy and Volker Schatz

Fraunhofer IOSB, Ettlingen 76275, Germany,
cem.bassoy@iosb.fraunhofer.de

Abstract. Tensors analysis has become a popular tool for solving problems in computational neuroscience, pattern recognition and signal processing. Similar to the two-dimensional case, algorithms for multidimensional data consist of basic operations accessing only a subset of tensor data. With multiple offsets and step sizes, basic operations for subtensors require sophisticated implementations even for entrywise operations.
In this work, we discuss the design and implementation of optimized higher-order functions that operate entrywise on tensors and subtensors with any non-hierarchical storage format and arbitrary number of dimensions. We propose recursive multi-index algorithms with reduced index computations and additional optimization techniques such as function inlining with partial template specialization. We show that single-index implementations of higher-order functions with subtensors introduce a runtime penalty of an order of magnitude than the recursive and iterative multi-index versions. Including data- and thread-level parallelization, our optimized implementations reach 68% of the maximum throughput of an Intel Core i9-7900X. In comparison with other libraries, the average speedup of our optimized implementations is up to 5x for map-like and more than 9x for reduce-like operations. For symmetric tensors we measured an average speedup of up to 4x.

1 Introduction

Many problems in computational neuroscience, pattern recognition, signal processing and data mining generate massive amounts of multidimensional data with high dimensionality [13,12,9]. Tensors provide a natural representation for massive multidimensional data [10,7]. Similar to matrix analysis algorithms, many recently developed iterative tensor algorithms apply basic tensor operations within subdomains of tensors, i.e. subtensors where their sizes usually depend on induction variables. For instance, the higher-order Jacobi method described in [3] accesses different subtensors of the same tensor in each iteration. In [5], subtensors are used to perform a histogram-based tensor analysis.

While basic tensor operations for multidimensional data have been implemented and discussed in the literature, the design and runtime analysis of algorithms with subtensors have only been sparsely considered. The implementation of entrywise operations for contiguously stored tensors can be efficiently and

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

conveniently implemented with a single loop where the storage format does not influence the runtime. In case of contiguously stored tensors, template functions of the C++ standard algorithm library can be applied. Operating on subtensors is more subtle and requires either index transformations for single loops or algorithms with a complex control-flow for multi-indexed access. Moreover, we have observed that single-index implementations are slower than recursive multi-index approaches in case of subtensors even if the spatial data locality is preserved.

In this work, we discuss optimized implementations of entrywise operations for tensors and subtensors in terms of their runtime behavior. We provide a set of optimized C++ higher-order template functions that implement a variety of map- and reduce-like tensor operations supporting tensors and subtensors with any non-hierarchical storage and arbitrary number of dimensions. The storage format, number of dimensions and the dimensions are latter can be specified at runtime. Our base implementations are based on a single-index approach with a single loop and multi-index approaches using recursion and iteration. We additionally present optimization techniques to minimize the performance penalties caused by recursion including data streaming, parallelization, loop unrolling and parametrized function inling with template meta-programming techniques. Each optimization is separately implemented in order to quantify its effects. Our proposed optimizations can also be applied for more complicated tensor operations such as tensor multiplications or transposition in order to efficiently support subtensors. In summary, the main contributions and findings of our work are:

- Multi-index higher-order functions with subtensors outperform single-index ones on a single core by one order of magnitude and perform equally well for any non-hierarchical storage format using permuted layout tuples.
- Dimension extents corresponding to the loop counts of the inner-most loops can reduce the throughput of recursively implemented higher-order functions by 37%. The runtime can be speed up by a factor of 1.6 using templated-based function inlining.
- Applying data- and thread-level parallelization, our implementations reach 68% of the maximum CPU throughput. Compared to competing implementations described in [14,6,1] the functions yield a speedup of up to 5x for map-like and more than 9x for reduce-like operations.

The remainder of the paper is organized as follows. Section 2 discusses existing libraries that are related to our work. Section 3 introduces the notation and states the problem based on the preliminary. Section 4 provides the research methodology and experimental setup used in this work. The following Section 5 introduces the single- and multi-index implementation. In Section 6 describes optimized multi-index versions of the multi-index implementation. Section 7 discusses and analyzes benchmark results with other libraries. The last Section 8 contains the conclusions.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

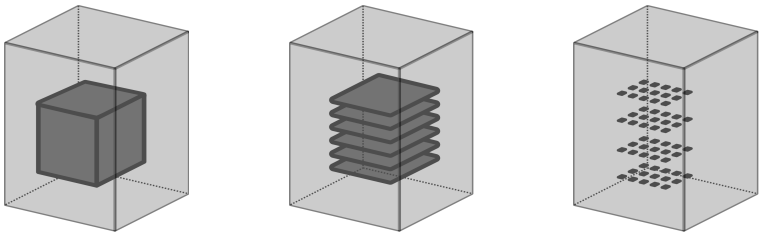

Fig. 1. Subtensors with an index offset $f_r > 1$ where (a) is generated with $t_r = 1$ for all $r \in \{1, 2, 3\}$, (b) with $t_1 > 1$ and $t_2 = t_3 = 1$ and (c) with $t_r > 1$ for all $r \in \{1, 2, 3\}$.

2 Related Work

Due to the wide range of applications of tensor calculus, there is a large number of libraries implementing it. In this section we will restrict ourselves to implementations that, like ours, support dense tensor operations. `Blitz`, described in [14], is one of the first `C++` frameworks. It supports tensors up to 11 dimensions including tensor and stencil operations. Multidimensional arrays are generic data types where the number of dimensions are compile-time parameters. The framework supports high-level expressions for entrywise tensor operations and also allows to manipulate subtensors for single core execution. The authors of [6] describe Boost's `MultiArray`. They discuss the design of generic data types for tensors, including the addressing elements and subtensors with first- and last-order storage formats. Users of the library must implement their own higher-order tensor operations using the provided tensor data structures. In [2], the library `MArray` and its implementations are presented. The order and dimensions of the tensor templates are runtime parameters similar to our work. The paper also discusses addressing functions, but only for the first- and last-order storage format. The entrywise tensor operations can also process subtensors. The Cyclops-Tensor-Framework (`CT`) offers a library primarily targeted at quantum chemistry applications. The order and the dimensions of their tensor data structures are dynamically configurable. `LibNT`, discussed in [8], serves a similar purpose. Both frameworks, however, do not allow manipulation of subtensors with entrywise tensor operations. `Eigen`'s tensor library is included in the Tensorflow framework [1]. The runtime of the latter contains over 200 standard operations, including mathematical, array manipulation, control flow, and state management operation. Among other features, the `C++` library framework also provides entrywise tensor operations with tensors and subtensors.

3 Notation

A tensor of order p shall be denoted by $\underline{\mathbf{A}}$ where p is the number of dimensions. Dimension extents are given by a shape tuple $\mathbf{n} = (n_1, \ldots, n_p)$ where $n_r > 1$ for

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

Table 1. Implemented higher-order template functions.

Abbreviation	Function	Description	Example (MATLAB)
scal	`for_each()`	$c_{ij\ldots k} \leftarrow \alpha \odot c_{ij\ldots k}$	`C(1:4,:,2:6) = C(1:4,:,2:6)+3`
copy	`copy()`	$c_{ij\ldots k} \leftarrow a_{ij\ldots k}$	`C(1:4,:,end) = A(2:5,:,end)`
add	`transform()`	$c_{ij\ldots k} \leftarrow \alpha \odot a_{ij\ldots k}$	`C(1:4,:,2:6) = A(2:5,:,3:7)+3`
addc	`transform()`	$c_{ij\ldots k} \leftarrow a_{ij\ldots k} \odot b_{ij\ldots k}$	`C(1:4,:,:) = A(2:5,:,:)+B(3:6,:,:)`
min	`min_element()`	$\min_{ij\ldots k}(a_{ij\ldots k})$	`min(A(2:5,:,3:7)(:))`
equal	`equal()`	$c_{ij\ldots k} \overset{!}{=} a_{ij\ldots k}$	`all(C(1:4,:,:)(:)==A(2:5,:,:)(:))`
all	`all_of()`	$a_{ij\ldots k} \overset{!}{=} \alpha$	`all(C(1:4,:,:)(:)==3)`
acc	`accumulate()`	$\sum_{ij\ldots k} a_{ij\ldots k}$	`sum(C(1:4,:,:)(:))`
inner	`inner_product()`	$\sum_{ij\ldots k} a_{ij\ldots k} \cdot c_{ij\ldots k}$	`dot(C(1:4,:,:)(:),C(2:6:,:)(:))`

$1 \leq r \leq p$. Elements of $\underline{\mathbf{A}}$ are denoted by $a_{i_1,i_2,\ldots,i_p}$, $\underline{\mathbf{A}}(i_1, i_2, \ldots, i_p)$ or $\underline{\mathbf{A}}(\mathbf{i})$ with $i_r \in I_r$ and $I_r = \{1, \ldots, n_r\}$ for all p dimensions. The set of all multi-indices is given by $\mathcal{I} = I_1 \times \cdots \times I_p$.

A subtensor denotes a selection of a multidimensional array $\underline{\mathbf{A}}$ that is defined in terms of p tuples such that $\underline{\mathbf{A}}' = \underline{\mathbf{A}}(\mathbf{s}_1, \ldots, \mathbf{s}_p)$. The r-th tuple $\mathbf{s}_r$ has n'_r elements with $n'_r \leq n_r$ and $s_k \in I_r$ for $1 \leq k \leq n'_r$. Most tensor algorithms use index triplets (f_r, t_r, l_r), where f_r, l_r define the first and last index satisfying $1 \leq f_r \leq l_r \leq n_r$. The parameter t_r with $t_r > 0$ is the step size or increment for the r-th dimension such that $n'_r = \lfloor (l_r - f_r)/t_r \rfloor + 1$. The index sets I'_r and the multi-index set $\mathcal{I}'$ of a subtensor $\underline{\mathbf{A}}'$ is defined in analogy to the index and multi-index set of tensor $\underline{\mathbf{A}}$. Figure. 1 illustrates three types of subtensors with different index triplet configurations.

4 Methodology and Experimental Setup

Table 1 lists some of the implemented and optimized higher-order functions. The offsets, increments, the number of dimensions and layout of the tensor can be dynamically set. The first four functions read from and write to memory regions. The following three functions read from memory regions and perform a reduce operation returning a scalar result. The last two functions are commonly used in numerical algorithms and also perform a reduce operation. Our implementation of higher-order functions support subtensors and tensors with any non-hierarchical storage format equally well. They can be thought as an extension of higher-order functions that are described by the C++ standard. Being applicable to contiguously stored tensors, they cannot be used to iterate over a multidimensional index set of a subtensor. We have applied multiple approaches and optimizations for all higher-order functions listed in Table 1 each of which is separately implemented. The implementations are as follows:

- *single-index* implementation uses a single loop.
- *multi-index-rec* implementation contains recursive functions calls.
- *multi-index-iter* is an iterative version using multi-indices.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

The following optimizations are based *multi-index* and are denoted as follows:

- *{minindex}* contains less index computations.
- *{inline}* avoids recursive function calls for a given compile-time order.
- *{parallel}* applies implicit data- and explicit thread-level parallelism.
- *{stream}* applies explicit data-parallelism and uses stream instrinsics.

We first quantify the runtime penalties that arise from index transformation within a single loop comparing *single-index* and *multi-index* implementations with subtensors. Based on the unoptimized *multi-index* implementation, we measure a combinations of optimizations, such as *{minindex,inline}*. We have defined multiple setups for the runtime and throughput measurements of our algorithms.

- *Setup 1* contains four two-dimensional arrays $\mathbf{N}_k$ of shape tuples for subtensors with 10 rows and 32 columns where each shape tuple $\mathbf{n}_{r,c}$ is of length $r+1$. The initial shape tuples $\mathbf{n}_{1,1}$ for all arrays are $(2^{15}, 2^8)$, $(2^8, 2^{15})$, $(2^8, 2, 2^{14})$ and $(2^8, 2^{15})$, respectively. The value of the k-th element is given by $\mathbf{n}_{r,c}(k) = \mathbf{n}_{1,1}(k) \cdot c/2^{r-1}$. If $k = 4$, the last element of all shape tuples instead of the fourth is adjusted. The remaining elements are set to 2 such that all shape tuples of one column exhibit the same number of subtensor elements. The subtensor sizes range from 32 to 1024 MB for single-precision floating-point numbers.
- *Setup 2* contains two-dimensional arrays $\mathbf{N}_k$ of shape tuples with 10 rows and 64 columns. The shape tuples are similarly created starting with the same initial shape tuple $(2^4, 2^{19})$. The first shape tuple elements are given by $\mathbf{n}_{r,c}(1) = \mathbf{n}_{1,1}(1) \cdot c$. The second and last dimension are adjusted according to $\mathbf{n}_{r,c}(2) = \mathbf{n}_{1,1}(2)$ / 2^{r-1} and $\mathbf{n}_{r,c}(r+1) = \mathbf{n}_{1,1}(r+1)$ / 2^{r-1}, respectively. The remaining shape tuple elements are set to 2. The subtensor sizes range from 32 to 2048 MB for single-precision floating-point numbers.
- *Setup 3* contains shape tuples that yield symmetric subtensors. The setup provides a two-dimensional array $\mathbf{N}$ of shape tuples with 6 rows and 8 columns where each shape tuple $\mathbf{n}_{r,c}$ is of length $r+1$. Elements of the shape tuples $\mathbf{n}_{r,1}$ for $r = 1, \ldots, 6$ are each 2^{12}, 2^8, 2^6, 2^5, 2^4 and 2^3. The remaining shape tuples for $c > 1$ are then given by $\mathbf{n}_{r,c} = \mathbf{n}_{r,c} + k \cdot (c-1)$ where k is respectively equal to 2^9, 2^5, 2^3, 2^2, 2, 1 for $r = 1, \ldots, 6$. In this setup, shape tuples of a column do not yield the same number of subtensor elements. The subtensor sizes range from 8 to 4096 MB for single-precision floating-point numbers.

The first two configurations with 4×320 and 2×640 shape tuples exhibit an orthogonal design in terms of tensor size and order, where the algorithms are run for fixed tensor sizes with increasing tensor order and vice versa. Varying only one dimension extent for a given order helped us to quantify its influence on the runtime. The last setup contains 48 tuples for symmetric tensors.

Subtensors are created with increments equal to one for all dimensions in order to analyze runtime penalties introduced by index computations and recursive function calls. Each subtensor is selected from a tensor that has a shape

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

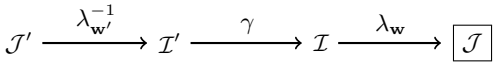

Fig. 2. Accessing contiguously stored elements requires the computation of scalar indices in $\mathcal{J}$. Function $\lambda_{\mathbf{w}}$ is applied if tensors are accessed with multi-indices in $\mathcal{I}$. Function $\lambda_{\mathbf{w}} \circ \gamma$ is applied if subtensor are accessed with multi-indices in $\mathcal{I}'$. Accessing elements subtensors with scalar indices in $\mathcal{J}'$ requires the application $\lambda_{\mathbf{w}} \circ \gamma \circ \lambda_{\mathbf{w}'}^{-1}$.

tuple of the last row of the corresponding two-dimensional array $\mathbf{N}_k$. One extent n_k of the subtensor is chosen smaller than the dimension extents of the referenced tensor. The sizes of the subtensors were chosen greater than the last-level cache to avoid caching effects. Spatial data locality is always preserved meaning that relative memory indices are generated according to storage format. Tensor elements are stored according to the first-order storage format for all setups. All of the following findings are valid for any other non-hierarchical storage format if the optimization in Section 6.2 is applied.

The experiments were carried out on an Core i9-7900X Intel Xeon processor with 10 cores and 20 hardware threads running at a base frequency of 3.3 GHz. It has a theoretical peak memory bandwidth of 85.312 GB/s resulting from four 64-bit wide channels with a data rate of 2666MT/s. The examples and tests were compiled with GCC 7.2 using the highest optimization level including the `-march=native` and `-pthread` options. The benchmark results presented below are the average of 10 runs. The throughput is given as number of operations times element size in bytes divided by the runtime in seconds. The comparison were performed with Eigen's tensor library (3.3.4), Boost's multiarray library (1.62.0) and Blitz's library (0.9) that were described in the Section 2.

5 Baseline Algorithms

If tensors are allocated contiguously in memory, a single index suffices to access all elements. The set of scalar indices is denoted by $\mathcal{J}$ with $\mathcal{J} = \{0, \ldots, \bar{n} - 1\}$ where $\bar{n} = \prod_{r=1}^{p} n_r$ with $|\mathcal{I}| = |\mathcal{J}|$. The mapping of multi-indices in $\mathcal{I}$ onto scalar indices in $\mathcal{J}$ depends on the layout of a tensor. The following mappings include any non-hierarchical layouts that can be specified by a layout tuple $\boldsymbol{\pi}$. The most common layouts are the first- and last-order storage formats with their respective layout tuples $\boldsymbol{\pi}_F = (1, 2, \ldots, p)$ and $\boldsymbol{\pi}_L = (p, p\text{-}1, \ldots, 1)$. The layout function $\lambda_{\mathbf{w}}$ with

$$\lambda_{\mathbf{w}}(\mathbf{i}) = \sum_{r=1}^{p} w_r(i_r - 1). \tag{1}$$

maps multi-indices in $\mathcal{I}$ to scalar indices in $\mathcal{J}$ for a fixed stride tuple $\mathbf{w}$ whose elements are given by $w_{\pi_1} = 1$ and

$$w_{\pi_r} = w_{\pi_{r-1}} \cdot n_{\pi_{r-1}} \quad \text{for } 1 < r \leq p. \tag{2}$$

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

Algorithm 1: Recursive algorithm.

Input: $\underline{\mathbf{A}} \in T^{\mathbf{n}}$, $\underline{\mathbf{B}} \in T^{\mathbf{n}}$, $\mathbf{n} \in \mathbb{N}^p$, $\mathbf{i} \in \mathbb{N}^p$, $r = p$
Result: $\underline{\mathbf{C}} \in T^{\mathbf{n}}$

```
1  transform(A, B, C, w, n, i, r)
2      if r > 1 then
3          for i_r ← 1 to n_r do
4              transform(A, B, C, w, n, i, r-1)
5      else
6          for i_1 ← 1 to n_1 do
7              j ← λ_w(i)
8              C[j] ← A[j] ⊙ B[j]
```

Algorithm 2: Iterative version.

Input: $\underline{\mathbf{A}} \in T^{\mathbf{n}}$, $\underline{\mathbf{B}} \in T^{\mathbf{n}}$ with $\mathbf{n} \in \mathbb{N}^p$, $\mathbf{i} \in \mathbb{N}^p$
Result: $\underline{\mathbf{C}} \in T^{\mathbf{n}}$

```
1  transform(A, B, C, w, n, i)
2      r ← 1
3      while r ≤ p do
4          for k ← 2 to r do
5              i_k ← 1
6          for i_1 ← 1 to n_1 do
7              j ← λ_w(i)
8              C[j] ← A[j] ⊙ B[j]
9          for r ← 2 to p do
10             if i_r < n_r then
11                 break;
12         i_r ← i_r + 1
```

The inverse layout function $\lambda_{\mathbf{w}}^{-1} : \mathcal{J} \to \mathcal{I}$ of $\lambda_{\mathbf{w}}$ is given by

$$\lambda_{\mathbf{w}}^{-1}(j) = \mathbf{i}, \quad \text{and} \quad i_r = \left\lfloor \frac{k_r}{w_r} \right\rfloor + 1, \tag{3}$$

with $k_{\pi_r} = k_{\pi_{r+1}} - w_{\pi_{r+1}} \cdot i_{\pi_{r+1}}$ for $r < p$ and $i_{\pi_p} = \lfloor j / w_{\pi_p} \rfloor + 1$. We can analogously define a scalar index set $\mathcal{J}'$ for a subtensor with $\bar{n}'$ elements where $\bar{n}' = \prod_{r=1}^{p} n'_r$. Note that λ can only be applied if $1 = f_r$, $l_r = n_r$ and $1 = t_r$ such that $n'_r = n_r$. In any other case, each multi-index in $\mathcal{I}'$ needs be mapped onto an multi-index in $\mathcal{I}$. The mapping $\gamma : \mathcal{I}' \to \mathcal{I}$ with $\gamma(\mathbf{i}') = \mathbf{i}$ is given by

$$\gamma_r(i'_r) = f_r + (i'_r - 1) \cdot t_r = i_r, \tag{4}$$

for $1 \leq r \leq p$. Subtensor elements can be accessed with single indices in $\mathcal{J}'$ by applying the function $\lambda_{\mathbf{w}} \circ \gamma \circ \gamma_{\mathbf{w}'}$ such that

$$j = \lambda_{\mathbf{w}} \left(\gamma \left(\lambda_{\mathbf{w}'}^{-1} \left(j' \right) \right) \right), \tag{5}$$

where $\mathbf{w}'$ and $\mathbf{w}$ are stride tuples of a subtensor and tensor. Figure 2 illustrates how a single-loop approach for subtensors requires scalar indices to be transformed according to Eq. (5).

5.1 Recursive Multi-Index Algorithm *multi-index-rec*

The baseline algorithm `transform` in Algorithm 1 exemplifies an implementation of entrywise operation for tensors and subtensors where $\odot$ is be a binary operation. It is a nonlinear recursive algorithm and has variable number of recursive function calls in each recursion level. The first input arguments denote a tensor or subtensor of order p all exhibiting the same shape tuple $\mathbf{n}$. Before each recursive function call, an element of the multi-index is incremented and passed to the following function instance for $r > 1$. The inner-most recursion level for $r = 1$ computes the first element i_1 of the multi-index and applies the

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

layout function defined in Eq. (1). Once the memory indices for all data structures are computed, the binary operation in line 7 is applied. Using equations (1) and (4), the layout function of subtensor is given by $\lambda' = \lambda_{\mathbf{w}'} \circ \gamma$ such that $\lambda'_{\mathbf{w}'}(\mathbf{i}') = \lambda_{\mathbf{w}}(\mathbf{f}) + \lambda_{\mathbf{w}''}(\mathbf{i}')$ where $\mathbf{f}$ is the tuple of the first indices of the subtensor and $\mathbf{w}''$ is a modified stride tuple with $w''_r = w'_r t_r$. The first summand $\lambda_{\mathbf{w}}(\mathbf{f})$ is an offset with which the pointer to the tensor data is shifted to set the position of the first subtensor element. In this way, the algorithm is able to process tensors and subtensors equally well.

5.2 Iterative Multi-Index Algorithm *multi-index-iter*

The basline algorithm provides an elegant solution for traversing the multi-index space and generating unique multi-indices with no redundant computation. However, recursion may introduce runtime overhead due to stack operations for saving the callers state [4]. In our case, $p - 1$ stack frames are repeatedly created before the inner-most loop is executed.

Nonlinear recursive algorithms can be transformed into an iteration using a software stack [15,11]. With no processing in between the recursive calls except the adjustment of the multi-index, we applied the method described in [15] and eliminated function calls which resulted in a much simpler control flow. We further simplified the algorithm, by only storing multi-index elements and to use a single array, where the r-th entry stores the r-th multi-index element.

The resulting iterative version of the recursive baseline algorithm is given in Algorithm 2. The multi-indices are modified in lines 3 to 5 and 8 to 11 just as it is done in line 2 to 4 in Algorithm 1. A multi-index element i_r is reset in lines 3 to 5 if any i_k with $k > r$ has reached the loop count n_k.

5.3 Single-Index Algorithm *single-index*

Higher-order functions for tensors can be implemented with one loop where tensor elements are accessed with a single index j. The memory address of the j-th element is given by the addition $k + \delta \cdot j$ where δ is the size of an element and k the memory location of the first element. We have used higher-order functions of the C++ standard library to perform elementwise operations for tensors. However, they cannot be used in case of subtensors where the values of the induction variable are in $\mathcal{J}'$. Each memory access with a scalar index needs a transformation according to Eq. (5). . The loop body for first-order storage format first increments the scalar j with $w_r \cdot i$ and updates k and i with $k \leftarrow \bar{k} - w'_r \cdot \bar{i}$ and $i \leftarrow k/w'_{r-1}$ where $\bar{k}$ and $\bar{i}$ are previously computed values.

6 Optimizing the Multi-Index Algorithm

In this section we will turn to algorithms optimized for accessing subtensors. The first subsection will present an iterative algorithm for higher-order functions. The following sections describe successive optimizations of the multi-index recursive

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

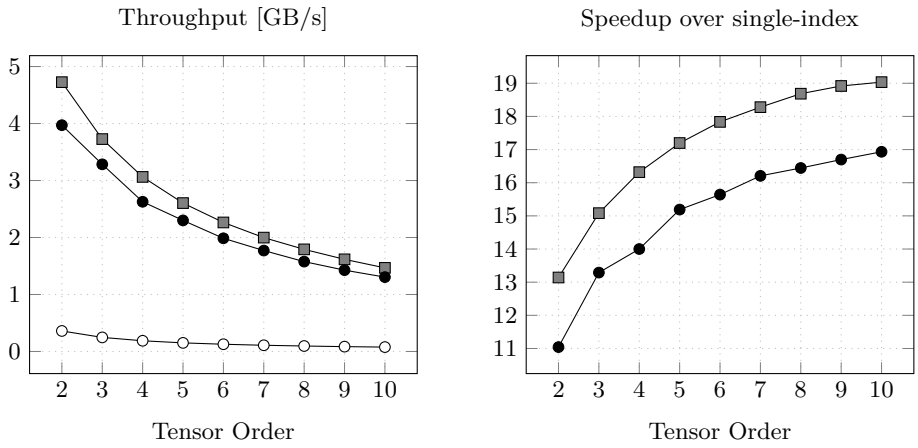

Fig. 3. Comparison of —○— *single-index*, —■— *multi-index-rec* and —●— *multi-index-iter* implementations of the entrywise addition of two subtensors. Data is stored in single precision. Tests are run on a single core with shape tuples of *Setup 1*. *Left*: Mean throughput. *Right*: Mean speedup of *multi-index-rec* and *multi-index-iter* over *single-index*.

algorithm from Section 5.1. Three of those subsections explain methods that can be applied in order to reduce the runtime penalties caused by the recursive approach. The last subsection discusses data- and thread-level parallelization with which bandwidth utilization can be maximized.

6.1 Reducing Index Computation *{minindex}*

The baseline algorithm for higher-order functions computes relative memory indices in the inner-most loop. We can further reduce the number of index computations by hoisting some of the summands to the previously executed loops. In each recursion level r, line 3 and 6 only modify the r-th element i_r of the multi-index $\mathbf{i}$. Moreover, the k-th function call at the r-th level adds k to i_r, i.e. increments the previously calculated index. We can therefore move the r-th summand $w_r \cdot i_r$ of Eq. (1) to the r-th recursion level. In this way, unnecessary index computations in the inner-most loop can be eliminated allowing to pass a single index j that is incremented by w_r. Algorithm 1 therefore needs to be modified in line 3, 6 and 7 to substitute j. At the recursion level r, the single index j is incremented n_r times with w_r until the stride $(r+1)$-th element of the stride tuple $\mathbf{w}$ is reached. The last element of the stride tuple $\mathbf{w}$ is given by $w_p \cdot n_p$. As j denotes a memory index, we can manipulate pointers to the data structures in the same manner. In this way only a dereferencing in the inner-most loop is necessary. The same holds for subtensor.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

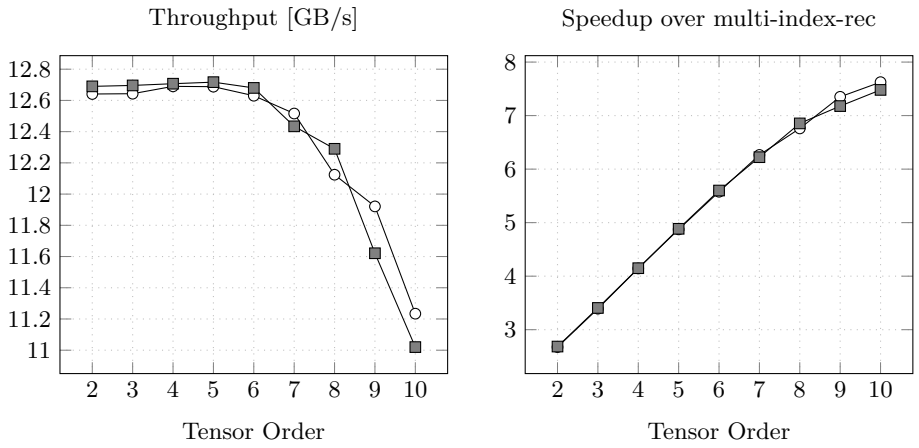

Fig. 4. Comparison of the *multi-index-rec{minindex}* —■— and *multi-index-iter {minindex}* —○— implementations of the entrywise addition of two subtensors. Data is stored in single precision. Tests are executed on a single core with shape tuples of *Setup 1*. *Left*: Mean throughput. *Right*: mean speedup over *multi-index-rec* implementation.

6.2 Preserving Data Locality

The spatial data locality of Algorithm 1 is always preserved for the first-order storage as the inner-most loop increments the first multi-index by $w_1 = 1$. For any other layout tuple, the elements are accessed with a stride greater than one. This can have a greatly influence the runtime of the higher-order function. In order to access successive element, we can reorder the loops or stride tuple according to the layout tuple. However, the modification of a stride tuple can be performed before the initial function call. Using the property $1 \leq w_{\pi_q} \leq w_{\pi_r}$ for $1 \leq q \leq r \leq p$, a new stride tuple $\mathbf{v}$ with $v_r = w_{\pi_r}$ for $1 \leq r \leq p$ can be computed. The runtime penalty for the permutation of the stride tuple becomes then negligible.

6.3 Reducing Recursive Function Calls *{inline}*

The recursion for the multi-index approach consists of multiple cases where each function call contains multiple recursive function calls, see [11]. Inlining can be guaranteed if a separate function is implemented and called for each order. This can be accomplished with class templates and partial specialization with a static member function containing a loop in order to reduce the number of function implementations. The order of the tensor and subtensor is a template parameter that allows the compiler to generate jump instructions for the specified order and to avoid recursive function calls. In order to leave the order runtime flexible, the static function is called from a switch statement. If the runtime-variable order is larger than the specified template parameter, the standard

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

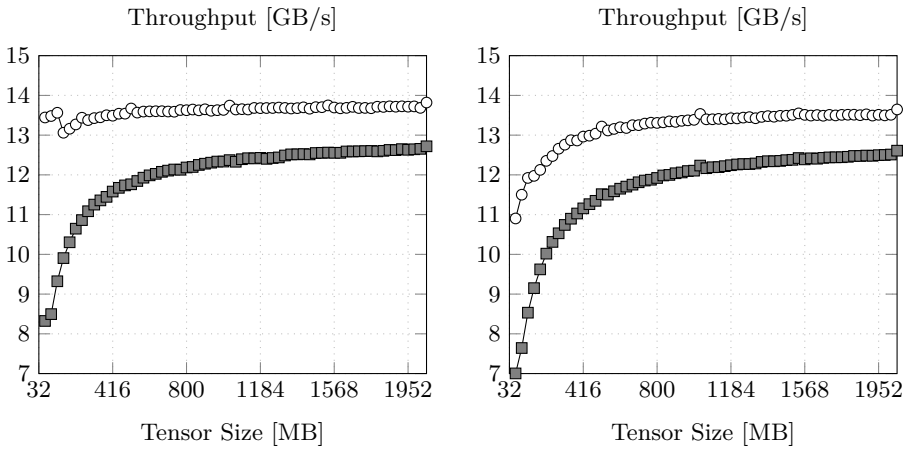

Fig. 5. Comparison of the recursive multi-index implementations of the entrywise subtensor addition with *{minindex,inline}* —○— and *{minindex}* —■— optimizations. Data is stored in single precision. Tests are executed on a single core with shape tuples of *Setup 2*. Left and right plots contain mean throughputs of the implementations executed with the first $\mathbf{N}_1$ and second shape tuple array $\mathbf{N}_2$, respectively.

recursive implementation is called. In order to prevent a code bloat of multiple implementations for different orders, we chose the order to 20.

6.4 Data- and Thread-Parallelization *{parallel,stream}*

By data-level parallelization we refer to the process of generating a single instruction for multiple data. The inner-most loop of the higher-order function is an auto-vectorizable loop if the stride of a tensor or subtensor is equal to one. In such a case, the compiler generates vector instructions with unaligned loads and regular store operations. In order to yield a better memory bandwidth utilization, we have explicitly placed Intel's aligned load and streaming intrinsics with the corresponding vector operation in the most inner loop. Note that pointers to the data structures must be aligned and the loop count must be set to a multiple of the vector size.

By thread-level parallelization we refer to the process of finding independent instruction streams of a program or function. Thread-level parallel exuction is accomplished with C++ threads executing the higher-order function in parallel where the outer-most loop is divided into equally sized chunks. Each thread executes its own instruction stream using distinct memory addresses of the tensor or subtensor. In case of reduction operations such the inner product with greater data dependencies, threads perform their own reduction operation in parallel and provide their results to the parent thread. The latter can perform the remaining reduction operation.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

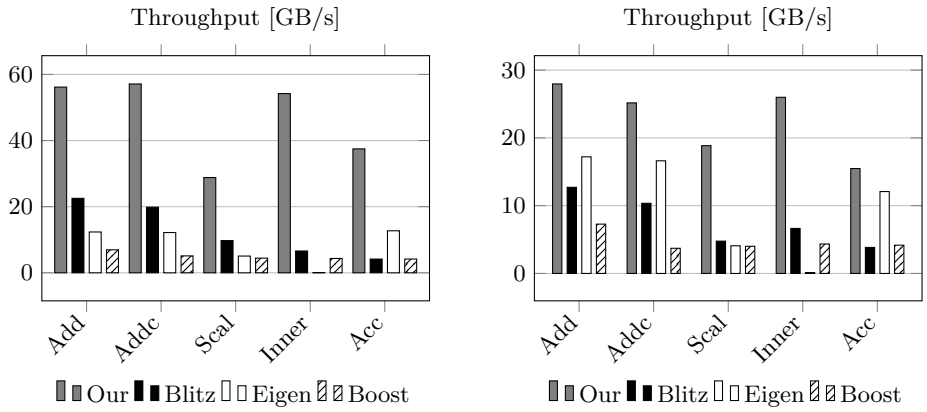

Fig. 6. Comparison of our implementation ▮▮ to Blitz ▮▮, Eigen ▯▯ and Boost ▨▨. Entrywise operations are executed with subtensors in single precision on all available cores. Function descriptions are found in Table 1. *Left:* Setup 1 was used for this benchmark. Our refers to *multi-index-rec {minindex,inline,parallel,stream}* reaching 68% of the theoretical peak memory bandwidth. *Right:* Setup 3 was used for this benchmark. Our refers to *multi-index-rec {minindex,inline,parallel}*.

7 Results and Discussion

A comparison of the *single-index*, *multi-index-rec* and *multi-index-iter* implementations for entrywise addition for subtensors is provided in Figure 3. The throughput for a given order is constant over all subtensor sizes and only varies between two different orders. The throughput of the *multi-index-rec* and *multi-index-iter* implementations speed up with increasing order and run up to 20 times faster than the *single-index* version. Note that both multi-index implementations perform equally well. We observed similar speedups for all other implemented higher-order functions.

Hoisting the index computation of the recursive *multi-index* approach with the *{minindex}* optimization significantly reduces the number of index computations, as shown in Figure 4. The recursive and iterative *multi-index {minindex}* linearly speed up with increasing order outperforming the unoptimized versions by almost a factor of 8. We measured a slowdown of at most 10% for any tuple shape of *Setup 1*.

Without the *{inline}* optimization, the recursive multi-index implementation runs slower for tensors and subtensors with a small extent of the first dimension. We did not observe similar behavior for shape tuples belonging to *Setup 1* where the extents of the first dimension is kept always greater than 256. Figure 5 illustrates the impact of the first dimension where only the first dimension is increased from 16 to 1024. Decreasing the first dimension causes a slow down almost up to a factor of 2. The positioning of the large dimension has a minor

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

impact on the throughput. The *{inline}* optimization reduces the runtime up to a factor of 1.6.

Comparing the throughput of the recursive implementations in Figures 4 and 6 using *{minindex,inline}* and *{minindex,inline,parallel,stream}* reveals that data- and thread-parallel execution with stream operations almost quadruples the sustained throughput from almost 14 GB/s on a single-core to almost 58 GB/s reaching about 68% of the theoretical peak performance. In comparison to other `C++` libraries our fastest implementation performs up to 5x faster for map-like operations and 9x for reduce operations with unsymmetric subtensors. Using implicit vectorization with unaligned access and store instructions reduces the throughput by 37%. With shape tuples of *Setup 3*, we did not use explicit vectorization and vector streaming instructions. In this case, higher-order functions utilized at most 34% of the theoretical peak memory bandwidth. Despite the lower throughput, all our functions yield speedups between 2x and 4x compared to the competing implementations as shown in Figure 6. While higher-order functions had almost a constant throughput with *Setup 1*, we measured a linear decrease of the throughput with increasing order for all implementations with *Setup 3*. This observation coincides with measurements based on the previous description shown in Figure 5.

Although *Blitz* does not support parallel execution, some of its entrywise operations perform almost as fast as our implementations on a single core. A parallelized version could also compete with our functions in terms of runtime. However, the implementation only supports a fixed number of dimensions which allows to provide an optimized implementation for each version. We implemented all entrywise operations for subtensors using *Boost*'s data structures and executed them on a single core. *Eigen*'s implementation executes entrywise operations in parallel. We observed a decrease of the throughput with increasing order as well. The performance of the reduce-like operations provided by the libraries is lower compared to the map-like operations.

8 Conclusion and Future Work

We have investigated the runtime behavior of higher-order functions for subtensors and showed that the recursive multi-index implementation linearly speeds up with increasing order over the single-index approach. Experimenting with a large amount of shape tuples we have shown that the dimension extents corresponding to the loop counts of the inner-most loops can reduce the throughput by 37%. Hoisting index computation and inlining function calls, the multi-index approach can be optimized, minimizing the performance penalties introduced by the recursion.

Applying explicit data-level parallelism with stream instructions and thread-level parallelism, we were able to speed up higher-order functions reaching 68% of the theoretical peak performance. In comparison to other `C++` libraries our fastest implementation performs up to 5x faster for map-like and 9x for reduce-like operations with unsymmetric subtensors. For symmetric subtensors we measured

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49

an average speedup of up to 4x. The findings of our work are valid for any type of tensor operation that includes recursive implementations.

In future work, we intend to create fast implementations of other tensor operations such as the tensor transposition and contractions and analyze the impact of the recursion.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., ..., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 265–283. OSDI'16, USENIX Association, Berkeley, CA, USA (2016)
2. Andres, B., Köthe, U., Kröger, T., Hamprecht, F.A.: Runtime-flexible multi-dimensional arrays and views for c++98 and c++0x. CoRR **abs/1008.2909** (2010)
3. Brazell, M., Li, N., Navasca, C., Tamon, C.: Solving multilinear systems via tensor inversion. SIAM Journal on Matrix Analysis and Applications pp. 542–570 (2013)
4. Cohen, N.H.: Eliminating redundant recursive calls. ACM Transactions on Programming Languages and Systems (TOPLAS) **5**(3), 265–299 (1983)
5. Fanaee-T, H., Gama, J.: Multi-aspect-streaming tensor analysis. Knowledge-Based Systems **89**, 332 – 345 (2015)
6. Garcia, R., Lumsdaine, A.: Multiarray: a c++ library for generic programming with arrays. Softw., Pract. Exper. **35**(2), 159–188 (2005)
7. Hackbusch, W.: Numerical tensor calculus. Acta numerica **23**, 651–742 (2014)
8. Harrison, A.P., Joseph, D.: Numeric tensor framework: Exploiting and extending einstein notation. Journal of Computational Science **16**, 128 – 139 (2016)
9. Kolda, T.G., Sun, J.: Scalable tensor decompositions for multi-aspect data mining. In: Proceedings of the 8th IEEE International Conference on Data Mining. pp. 363–372. IEEE, Washington, DC, USA (2008)
10. Lim, L.H.: Tensors and hypermatrices. In: Hogben, L. (ed.) Handbook of Linear Algebra. Chapman and Hall, 2 edn. (2017)
11. Liu, Y.A., Stoller, S.D.: From recursion to iteration: what are the optimizations? ACM Sigplan Notices **34**(11), 73–82 (1999)
12. Savas, B., Eldén, L.: Handwritten digit classification using higher order singular value decomposition. Pattern recognition **40**(3), 993–1003 (2007)
13. Suter, S.K., Makhynia, M., Pajarola, R.: Tamresh - tensor approximation multiresolution hierarchy for interactive volume visualization. In: Proceedings of the 15th Eurographics Conference on Visualization. pp. 151–160. EuroVis '13, Eurographics Association (2013)
14. Veldhuizen, T.L.: Arrays in blitz++. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) ISCOPE. Lecture Notes in Computer Science, vol. 1505, pp. 223–230. Springer (1998)
15. Ward, M.P., Bennett, K.H.: Recursion removal/introduction by formal transformation: An aid to program development and program comprehension. Computer Journal **42**(8), 650–650 (1999)

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_49