On the Performance Prediction of BLAS-based Tensor Contractions

Elmar Peise(✉), Diego Fabregat-Traver, and Paolo Bientinesi

AICES, RWTH Aachen, Aachen, Germany
{peise,fabregat,pauldj}@aices.rwth-aachen.de

Abstract. Tensor operations are surging as the computational building blocks for a variety of scientific simulations and the development of high-performance kernels for such operations is known to be a challenging task. While for operations on one- and two-dimensional tensors there exist standardized interfaces and highly-optimized libraries (BLAS), for higher dimensional tensors neither standards nor highly-tuned implementations exist yet. In this paper, we consider contractions between two tensors of arbitrary dimensionality and take on the challenge of generating high-performance implementations by resorting to sequences of BLAS kernels. The approach consists in breaking the contraction down into operations that only involve matrices or vectors. Since in general there are many alternative ways of decomposing a contraction, we are able to methodically derive a large family of algorithms. The main contribution of this paper is a systematic methodology to accurately identify the fastest algorithms in the bunch, *without executing them*. The goal is instead accomplished with the help of a set of cache-aware micro-benchmarks for the underlying BLAS kernels. The predictions we construct from such benchmarks allow us to reliably single out the best-performing algorithms in a tiny fraction of the time taken by the direct execution of the algorithms.

1 Introduction

Tensor contractions play an increasingly important role in various scientific computations such as general relativity [1,2] and electronic structure calculations in quantum chemistry [3–5]. Computationally, contractions are generalizations of matrix-vector and matrix-matrix products that involve operands of higher dimensionality. While there are several highly-tuned implementations of the Basic Linear Algebra Subprograms (BLAS) [6–8] for operands with up to 2 dimensions, there are no equivalently standardized high-performance libraries for general tensor contractions. Fortunately, just as matrix-matrix products can computationally be decomposed into a sequence of matrix-vector products, most higher dimensional tensor contractions can be cast in terms of matrix-matrix or matrix-vector BLAS kernels. However, each tensor contraction can be computed via BLAS kernels in many, even hundreds, of different ways, each with its own performance signature. This work addresses the problem of accurately predicting the performance of BLAS-based algorithms for tensor contractions.

S.A. Jarvis et al. (Eds.): PMBS 2014, LNCS 8966, pp. 193–212, 2015.
DOI: 10.1007/978-3-319-17248-4_10

One could argue that only algorithms that use the `gemm` kernel[1] are real candidates to achieve the best performance; while for the most part this observation is true, due to the fact that in practical contractions it is often the case that one or more dimensions are very small (while BLAS is mostly optimized for large dimensions), the difference in performance between two `gemm`-based algorithms can be dramatic. At any rate, with this work we aim at the accurate prediction of any BLAS-based contraction, irrespective of which kernel is used. Our approach, which never resorts to timing a full algorithm, makes use of what we call micro-benchmarks. These benchmarks execute only one BLAS operation in a prescribed memory environment. The idea is to analyze the structure of the code, and determine the state of the cache (precondition) prior to the execution of the kernel; we carefully recreate this state within the micro-benchmark so that the specific kernel can be timed in conditions analogous to those experienced in the actual algorithm. Based on these timings, we extrapolate the total algorithm execution times with sufficient accuracy to single out the fastest algorithms. This micro-benchmark-based prediction proves to be several orders of magnitude faster than executions of the actual algorithms.

Tensor Notation. In the following, we denote tensor contractions by means of the Einstein notation;[2] let us briefly explain said notation by means of an example. In the contraction $C_{abc} := A_{ai}B_{ibc}$, the entries C`[a,b,c]` of the resulting three-dimensional tensor $C \in \mathbb{R}^{a \times b \times c}$ are computed as

$$\forall \texttt{a} \forall \texttt{b} \forall \texttt{c}. C\texttt{[a,b,c]} := \sum_{\texttt{i}} A\texttt{[a,i]} B\texttt{[i,b,c]}.$$

(In this notation, a matrix-matrix product is denoted by $C_{ab} := A_{ai}B_{ib}$.) The indices that appear in both tensors A and B — the summation indices $i, j, \ldots$ — are called *contracted*, while those that only appear in either A or B (and thus in C) — $a, b, c, \ldots$ — are called *free* or *uncontracted*. W.l.o.g., we assume that tensors are stored as Fortran-style contiguous multidimensional double precision arrays: vectors (1D tensors) are stored contiguously, matrices (2D tensors) are stored as sequence of column vectors, 3D tensors (visualized as cubes) are stored as a sequence of matrices (planes of the cube), and so on.

Related Work. The most prominent project targeting the efficient computation of tensor contractions is probably the Tensor Contraction Engine, a compiler built specifically for multi-tensor multi-index contractions to be executed within memory constraints [9]; in light of the wide diffusion and nearly optimal efficiency of the BLAS library, an extension to TCE was proposed to compute

[1] `gemm` is the BLAS-3 routine for matrix-matrix multiplication, which on many systems is optimized within a few percent of peak performance.

[2] For the sake of simplicity and without any loss of generality, we ignore any distinction between covariant and contravariant vectors; this means we treat any index as a subscript.

contractions via BLAS operations [10]. In the same spirit, we provided simple rules to build a taxonomy for all contractions between two tensors, identifying which BLAS routines are usable and how to best exploit them [11].

There also exists a variety of work in the field of performance prediction in the context of dense linear algebra. A notable example is Iakymchuk et al. [12,13], where the authors model the performance of dense linear algebra algorithms analytically based on very detailed models of the occurring cache-misses. Also, in [14], we use measurement-based performance models to predict the behavior of blocked algorithms. However, none of these works target or address high-performance tensor contractions and their peculiarities, i.e., very regular patterns in routine invocation and memory access, but highly skewed dimensionality (tiny sizes for at least one of the dimensions).

Structure of the Paper. The rest of this paper is structured as follows. The systematic generation of BLAS-based algorithms for tensor contractions is discussed in Sect. 2. Our performance prediction framework is introduced in Sect. 3, and experimental results for a range of contractions are presented Sect. 4.

2 Algorithm Generation

In this section, we briefly explain how we systematically generate a family of BLAS-based algorithms for a tensor contraction. For a detailed discussion of the topic, we refer the reader to [11].

Aware of the extreme level of efficiency inherent to the best BLAS implementations, our approach for computing a contraction consists in reducing it to a sequence of calls to one of the BLAS kernels. Since BLAS operates on scalars, vectors and matrices (zero-, one- and two-dimensional objects), tensors must be expressed in terms of a collection of such objects. To this end, we introduce the concept of *slicing*: With the help of Matlab's ":" notation,[3] slicing a d-dimensional operand $\mathcal{O}p \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ along the i-th index (or dimension) means creating the n_i $(d–1)$-dimensional slices $\mathcal{O}p[\underbrace{\texttt{:},\ldots,\texttt{:}}_{i-1},\texttt{k},\underbrace{\texttt{:},\ldots,\texttt{:}}_{d-i}]$, where $\texttt{k} = 1,\ldots,n_i$.

Example 1. Consider the matrix-matrix product $C_{ab} := A_{ai}B_{ib}$. Slicing the matrix B along dimension b reduces the matrix to a collection of column vectors; accordingly, the matrix-matrix product is reduced to a sequence of matrix-vector operations:[4]

[3] In the Matlab-like notation used in this paper, `1:`b are the numbers from 1 to b, while an index `:` in a tensor refers to all elements along that dimension, e.g., C`[:,b]` is the b-th column of C.

[4] The pictogram next to the algorithm visualizes the slicing of the tensors that originates the algorithm's sequence of `gemv`s. The red objects represent the operands of the BLAS kernel.

Similarly, a multi-dimensional tensor contraction can be reduced to operations involving solely matrices and vectors.

Depending on the slicing choices, a contraction is reduced to a number of nested loops with one of the following kernels at the innermost loop's body:

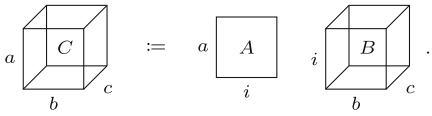
- BLAS-1:
  - `dot` (vector-vector inner product: $\alpha := x^T y$),
  - `axpy` (vector scaling and addition: $y := \alpha x + y$),
- BLAS-2:
  - `gemv` (matrix-vector product: $y := Ax + y$),
  - `ger` (vector-vector outer product: $A := xy^T + A$), and
- BLAS-3:
  - `gemm` (matrix-matrix product: $C := AB + C$).

Notice that to comply with the BLAS interface, the elements in one of the two dimensions of a matrix must be contiguous. Therefore, algorithms that rely on `gemv`, `ger`, or `gemm` as their computational kernel may require a temporary copy of slices before and/or after the invocation of the corresponding BLAS routine.

As a case study, let us consider the contraction

$$C_{abc} := A_{ai} B_{ibc}, \tag{1}$$

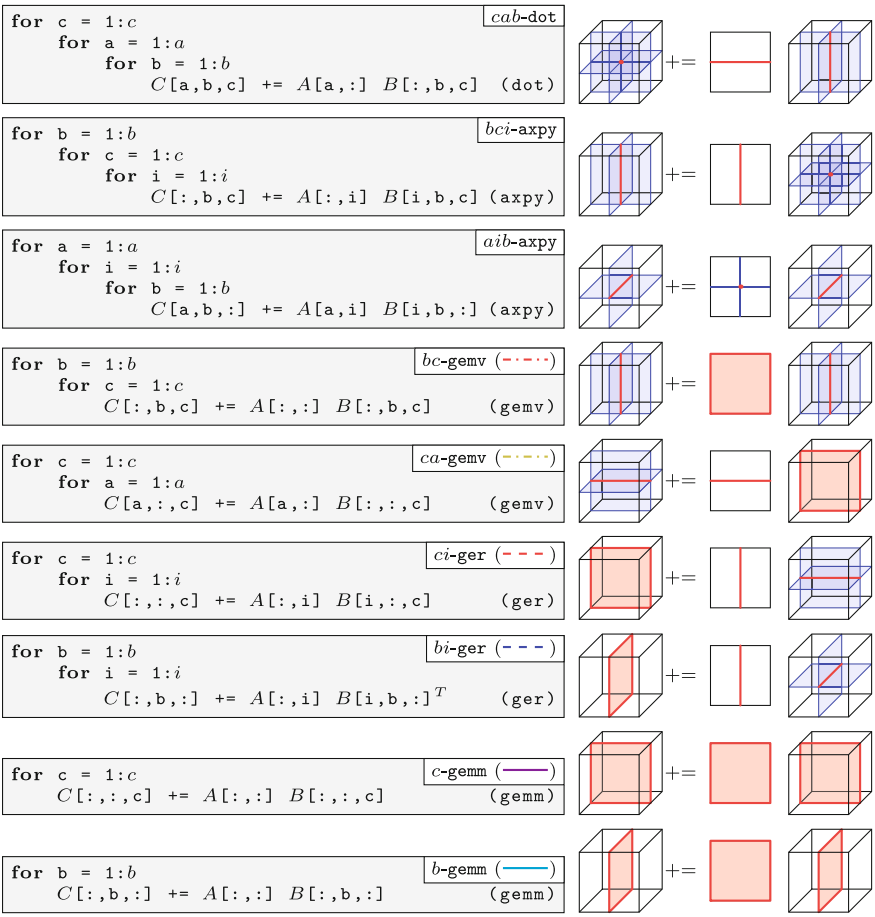
which is visualized as follows:



Instead of a blind search for appropriate slicings, we generate algorithms by following a goal-oriented approach: For each of the five kernels of interest, we know the dimensionality required for each operand; accordingly, we deduce how many slices are needed and which combination of free/contracted indices to slice. Table 1 (left) exhibits, for each kernel, the conditions necessary for a contraction to be computed in terms of that kernel. In particular, the second and the third columns indicate how many contracted and free indices, respectively, appear in each kernel. A and B refer to the first and the second input operand of the kernel; in a contraction between tensors of arbitrary dimension, all indices beyond those indicated in these columns must be sliced.

Example 2. Since `gemm` involves one free index in each of its operands A and B, and one contracted index (common to both A and B), in order to reduce a contraction to a sequence of `gemm` calls, one must slice all free indices of A but one, all free indices of B but one, and all contracted indices but one. With

reference to (1), this is achieved by slicing either dimension b or c, resulting in the two algorithms (b-`gemm` and c-`gemm`)[5] shown in the last two examples of Algorithm 1[6].



Algs. 1: $C_{abc} := A_{ai}B_{ibc}$: 9 exemplary algorithms out of 36.

As already mentioned, given a contraction, there is no obvious a-priori choice of kernel and slicings to attain the highest performance. We therefore generate all possible combinations. Moreover, due to their impact on performance and to further stress our modeling tool, we generate all possible permutations of the loops.

We developed a small algorithm and code generator that produces all such algorithms, constructs for each of them a C-implementation, as well as an abstract

[5] The algorithm names are composed of two parts: The first part is the list of sliced tensor indices iterated over by the algorithm's loops including an apostrophe ′ for each `copy`-kernel; the second part is the BLAS-kernel at the algorithm's core.

[6] For algorithms with more than 1 `for`-loop, all slicings are visualized in blue and only the kernel operands (the slicings' intersections) are in red.

Table 1. Rules for tensor slicing to obtain a given BLAS kernel. Left: how many contracted and free indices appear in a kernel. Right: different slicings make it possible to express one contraction in terms of different kernels. The names in the rightmost column refer to Algorithm 1.

Kernel	Number of indices		Examples from $C_{abc} := A_{ai}B_{ibc}$		
	Contracted	Free	Kernel indices	Sliced indices	Resulting algorithm
`dot`	1	0	i	c, a, b	cab-`dot`
`axpy`	0	(1 in $A \wedge 0$ in B) $\vee$	a	b, c, i	bci-`axpy`
		(0 in $A \wedge 1$ in B)	c	a, i, b	aib-`axpy`
`gemv`	1	(1 in $A \wedge 0$ in B) $\vee$	i, a	b, c	bc-`gemv`
		(0 in $A \wedge 1$ in B)	i, b	c, a	ca-`gemv`
`ger`	0	1 in $A \wedge 1$ in B	a, c	i, b	ib-`ger`
`gemm`	1	1 in $A \wedge 1$ in B	i, a, b	c	c-`gemm`

syntax tree (AST) representing its loop-based structure. The ASTs are then passed to the prediction tool introduced in the following section.

3 Performance Prediction

In this section, we present how to accurately model the performance of algorithms that compute tensor contractions through BLAS kernels. These algorithms consist of one or more nested loops and cast all computation in terms of one single BLAS kernel. Taking advantage of this structure, we aim at estimating the execution time of a target algorithm with the help of only few micro-benchmarks of the kernels, i.e., with no direct execution of the algorithm itself. In order to obtain reliable estimates, the micro-benchmarks need to be executed in a setup that mirrors as closely as possible the computing environment (most importantly the cache) within the contraction algorithm. In the following, we incrementally go through the steps required to build a meaningful "replica" of the computing environment.

Throughout this section, we track the changes in the performance prediction by considering the exemplary contraction $C_{abc} := A_{ai}B_{ibc}$. We chose the tensors $A \in \mathbb{R}^{a \times i}$ and $B \in \mathbb{R}^{i \times b \times c}$ of size $i = 8$ and $a = b = c = 8, \ldots, 1{,}000$ — a deliberately challenging scenario due to the thin tensor dimension i, for which BLAS kernels are generally not optimized. Our generator produces 36 algorithms for the considered contraction, some of which are shown in Algs. 1:

- 6 `dot`-based,
- 18 `axpy`-based,
- 6 `gemv`-based: bc-`gemv` (-·-··), cb-`gemv` (-·-··), ac-`gemv` (-·-··), ca-`gemv` (-·-··), ab-`gemv` (-·-··), ba-`gemv` (-·-··),

– 4 `ger`-based: *ci*-`ger` (- - -), *ic*-`ger` (- - -), *bi*-`ger` (- - -), *ib*-`ger` (- - -), and
– 2 `gemm`-based: *c*-`gemm` (——), *b*-`gemm` (——).

In this section, to focus our attention, we will only consider the BLAS-2 and BLAS-3 based algorithms (i.e., with kernels `gemv`, `ger`, and `gemm`).

We execute these algorithms on 1 core of an Intel Penryn E5450 (Harpertown) CPU[7] linking with the OpenBLAS library [15]. Figure 1a displays the performance, in terms of computed floating point operations per clock cycle (flops/cycle), measured for each algorithm; our goal is to accurately reproduce, without executing the algorithms, such performance profiles. While it is evident that only two of the algorithms — the `gemm`-based *c*-`gemm` (——) and *b*-`gemm` (——) — are competitive, we aim at predicting the behavior of all algorithms to develop and demonstrate the broad applicability of our methodology.

3.1 Repeated Execution

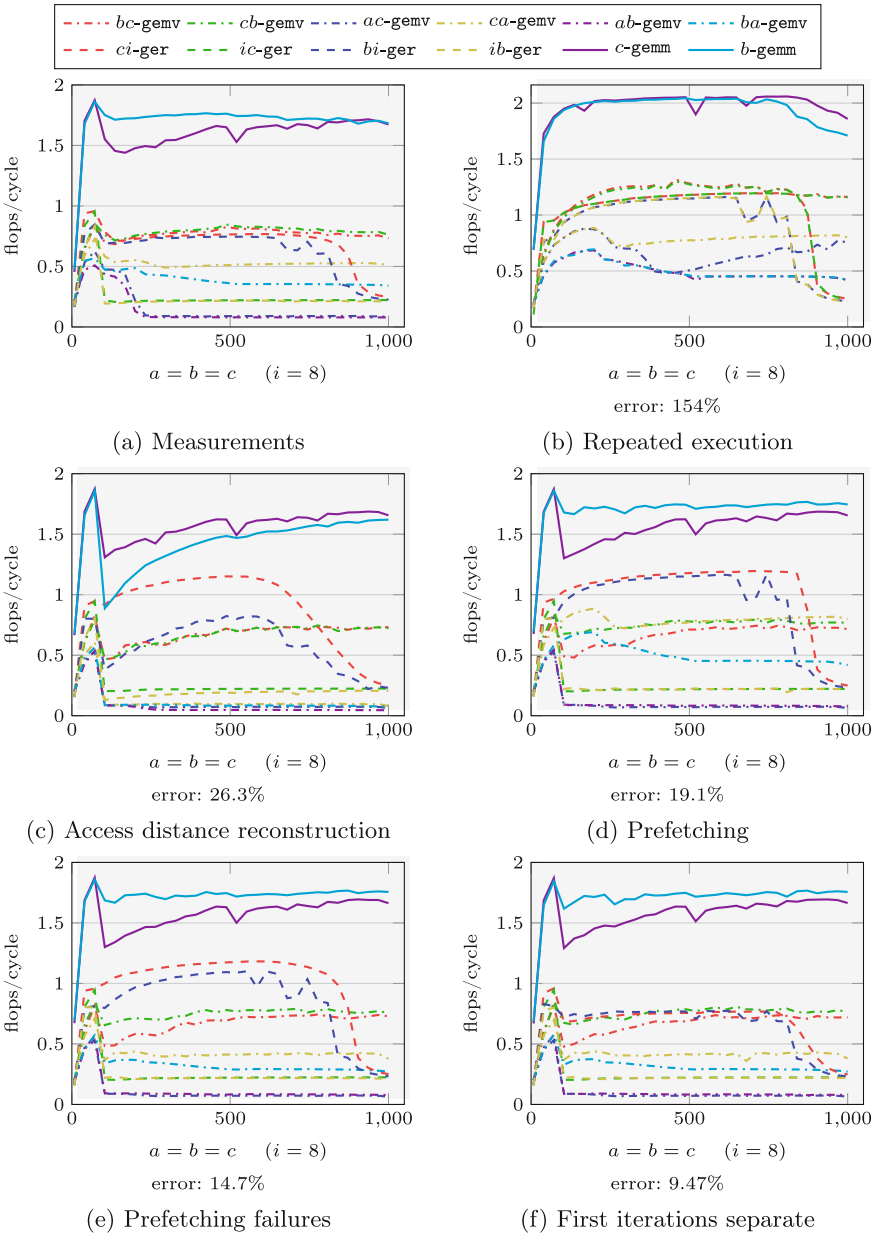
The first, most intuitive, attempt to predict the performance of an algorithm relies on the isolated and repeated measurement of its BLAS kernel's performance. We implemented this approach by executing each kernel ten times and extracting the median execution time; the corresponding estimate is then obtained by multiplying the median by the number of kernel invocations within the algorithm. In our example, this boils down to multiplying the kernel execution time with the product of all loop lengths.

The performance profiles predicted by this first, rough approach are shown in Fig. 1b. By comparing this figure with the reference Fig. 1a, it becomes apparent that while the two top algorithms are already correctly identified, the performance of almost all algorithms is consistently overestimated — the average absolute error with respect to the measured performance is 154 %. In other words, when executed as part of the algorithms, the BLAS kernels take longer to complete than in the isolated micro-benchmarks. The reason for this discrepancy is that the micro-benchmarks invoke the kernels repeatedly with the same memory regions as operands, i.e., they operate on warm data (the operands remain in the CPU's cache). Within the algorithm, by contrast, at least one, and potentially even all of the operands, vary from one invocation to the next, i.e., the kernels operate at least partially on cold data.

3.2 Operand Access Distance

In order to improve the accuracy of the predictions, the idea is to first identify the state of the cache in the algorithm prior to the invocation of the BLAS kernel ("precondition"), and to then replicate this status in the micro-benchmark. For this purpose, each algorithm is symbolically analyzed to reconstruct the order of memory accesses involving the kernel's operands. For each operand, we determine the set of memory regions M that were loaded into cache since its last access, and define the *access distance* as the sum of the size of these regions M.

[7] 2 GHz, 4 cores, 4 double precision flops/cycle/core, 6 MB L2 cache/2 cores.



(a) Measurements (b) Repeated execution

(c) Access distance reconstruction (d) Prefetching

(e) Prefetching failures (f) First iterations separate

Fig. 1. $C_{abc} := A_{ai}B_{ibc}$: Performance measurements and various stages of performance predictions (BLAS-2 and BLAS-3). The presented errors for the predictions (b) – (f) are the average absolute difference with respect to the measurements (a).

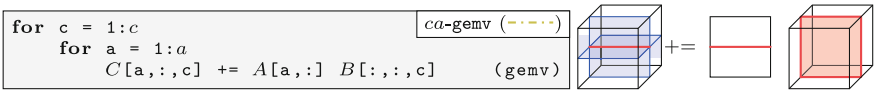
Once the access distances for all operands of a kernel are determined, we can create an artificial sequence of memory accesses to reconstruct the cache precondition. Based on this cache setup, the BLAS kernels are timed in a micro-benchmark that closely resembles the actual execution of the algorithm. As before, these micro-benchmarks are repeated and timed ten times to yield a stable median. From the median, the performance of the algorithm is again obtained based on the number of kernel invocations per algorithm execution.

To predict which memory regions are in cache, we assume a fully associative Least Recently Used (LRU) cache replacement policy[8] and sum up the size of all memory regions accessed since an operand's last use, yielding the access distance. In first instance, we also assume that all loops surrounding the kernel are somewhere in the middle of their traversal (i.e., not in their first iteration); this assumption will be lifted later.

We now describe how to obtain the access distance for each operand. The presented method is general and allows for any combinations of loops and multiple kernels within the abstract syntax tree (AST), however for the sake of clarity, we limit the discussion to ASTs that only consist of a series of loops with a single call to a BLAS kernel at their core.

For each operand $\mathcal{O}p$, we examine the algorithm's AST (see Sect. 2) with the kernel of interest as a starting point. The AST is traversed backwards until the previous access to $\mathcal{O}p$ (or the AST's root) is found, thereby collecting all other operands involved in kernels in the initially empty set M. Going up the AST, three different cases can be encountered.

1. **$\mathcal{O}p$ does not vary across the surrounding loop.**
   Example 3. In algorithm *ca*-`gemv` (-·-··), repeated below, the operand b`[:,:,c]` does not depend on the surrounding loop's iterator `a`. Hence, $M = \emptyset$ and b`[:,:,c]`'s access distance is 0.



   $\mathcal{O}p$ refers to the same memory region as in the previous iteration of the surrounding loop. The back-traversal therefore terminates and the memory regions collected in M so far determine the access distance.
2. **$\mathcal{O}p$ varies across the surrounding loop.**
   Example 4. In algorithm *ca*-`gemv` (-·-··), the operand A`[a,:]` depends on the surrounding loop's iterator `a`.
   $\mathcal{O}p$ referred to a different memory region in the previous iteration of the loop. As a result, it is safe to assume that at least all memory regions covered by all kernel operands throughout this loop's iterations were accessed since the last access to $\mathcal{O}p$. Hence, all operands are added to M and the memory regions are symbolically joined along the dimensions the loop iterates over.

[8] Due to the regular storage format and memory access strides of dense linear algebra operations such as the considered tensor contractions, this simplifying assumption does not affect the reliability of the results.

Example 4 (continued). The algorithm's kernel operates on A`[a,:]`, B`[:,:,c]`, and C`[a,:,c]`. Joining these operands across the index `a` yields the memory regions $M = \{A\texttt{[:,:]}, B\texttt{[:,:,c]}, C\texttt{[:,:,c]}\}$.
Since a previous access to $\mathcal{O}p$ was not yet detected, the traversal proceeds by going up one level in the AST, and applying the method recursively: the surrounding loop now takes the role of the starting node and we look for a previous access to $\mathcal{O}p$ joined across this loop.
Example 4 (continued). The back-traversal now looks for a previous access to A`[:,:]` (A`[a,:]` joint across `a`) on the second-innermost loop. This time, the region is independent of the surrounding loop's iterator `c`; therefore, in this second step, case 1. above applies and the access distance is computed from the previously collected set $M = \{A\texttt{[:,:]}, B\texttt{[:,:,c]}, C\texttt{[:,:,c]}\}$.

3. **The parent node is the AST's root.**
Example 5. In algorithm *ca*-`gemv` (-·-·-), the operand C`[a,:,c]` depends on both of the surrounding loops' iterators `a` and `c`. Therefore, the back-traversal encounters case 2. above in both its first and second step, joining the kernel's operands A`[a,:]`, B`[:,:,c]`, and C`[a,:,c]` across first `a` and then `c`, yielding $M = \{A\texttt{[:,:]}, B\texttt{[:,:,:]}, C\texttt{[:,:,:]}\}$. In the third step of the back-traversal, the outermost loop is already the starting point — the algorithm's root is reached.
In this case, the considered region is accessed only once (and for the first time). Since we do not know how the contraction is used (within a surrounding program), we can generally not make any assertions on the access distance. For the purpose of this paper, in which we execute the contraction repeatedly to measure its performance, however, we assume that no further memory regions were loaded since the last invocation of the contraction — i.e., we compute the access distance from the previously collected memory regions in M.

Based on the such obtained access distance for each operand of an algorithm's kernel, we now construct a list of memory accesses that emulates the accesses within the algorithm prior to the kernel's execution. This list consists of accesses to the kernel's operands, interleaved with accesses to remote memory regions, in order to flush portions of the cache corresponding to the access distances: First, we access the operand with the largest access distance, then a remote region that accounts for the difference to the next smaller access distance, followed by the next operand, and so on until the operands with the smallest access distance followed by a remote access of this size. If the access distances to the first operands in this list are larger than $\frac{5}{4}$ times the cache size, the list is truncated down to this limit at the front.

Example 6. For algorithm *ca*-`gemv` (-·-·-), the following table summarizes the operands $\mathcal{O}p$, their sizes s, the corresponding collections M and the implicated access distances d for contraction sizes $a = b = c = 400$ and $i = 8$ (all sizes in doubles = 8 bytes):

$\mathcal{O}p$	s	M	d
B[:,:,c]	3,200	$\emptyset$	0
A[a,:]	8	$\{A$[:,:], B[:,:,c], C[:,:,c]$\}$	166,400
C[a,:,c]	400	$\{A$[:,:], B[:,:,:], C[:,:,:]$\}$	65,283,200

From these distances, we get the following list of memory accesses as a setup for the gemv-kernel, where $[s]$ correspond to remote memory accesses of size s:

$$C[\mathtt{a},:,\mathtt{c}], [65{,}116{,}792], A[\mathtt{a},:], [163{,}200], B[:,:,\mathtt{c}].$$

Note, that the remote accesses do not directly correspond to the access distances; instead, this size is reached for each operand as the sum of the sizes of all accesses to its right in this list. (e.g., the access distances of A[a,:] is reached as $163{,}200 + \text{sizeof}(B[:,:,\mathtt{c}]) = 166{,}400$).

Now, the largest access distance is at 65,283,200 considerably larger than 983,040 ($\frac{5}{4}$ times the cache size of $\frac{6\,\text{MB}}{8} = 786{,}432$ doubles). Hence, the list is cut at this size, yielding the final setup for this algorithm's micro-benchmark:

$$[816{,}632], A[\mathtt{a},:], [163{,}200], B[:,:,\mathtt{c}].$$

The thus obtained benchmark, consisting of the setup followed by the kernel invocation, is once more executed ten times. The median of the kernel run-times of these ten benchmarks is then used to compute our second execution time estimate.

In Fig. 1c, we present the flops/cycle performance of our new estimates. These predictions are much closer to the measured performance (Fig. 1a) than the first rough estimates (Fig. 1b): the average error is reduced to 26.3 %. For several algorithms (such as *ic*-ger (- - -), Algs. 1), the error is already within a few percent; for many others instead, the predictions are still off. In particular, the performance of some algorithms — for instance, *bi*-ger (- - -) (see Algs. 1) — is now underestimated; this is due to the fact that based on the access distance, certain operands are placed out of cache, while in practice they are (partially) brought into cache through either prefetching or because they share cache-lines across the innermost loop's iterations. We address this discrepancy by further refining our micro-benchmarks.

3.3 Cache Prefetching

In the considered type of tensor contraction algorithms, prefetching of operands or sharing of cache lines across loop iterations occur frequently.

Example 7. In algorithm *bi*-ger (- - -), the operand A[:,i] points to a different memory location in each iteration of the inner loop across i. However, these vectors-operands are consecutive in memory; thus, when reaching the end of A[:,i], the prefetcher will likely already load the next memory elements, which

constitute A[:,i] in the next iteration. Likewise, operand B[i,b,:] varies across inner loop iterations; however, since this loop iterates over the region's first dimension i, 8 consecutive operands[9] B[i,b,:] will occupy the same cache-line.

Such prefetching situations occur when a certain set of conditions are met, namely:

1. the operand varies across the directly surrounding loop, and
2. the iterator of this loop indexes
   - either the first dimension of the operand,
   - or its second dimension, while the first is accessed entirely, or fits in a single cache-line.

As part of our AST-based algorithm analysis, these conditions are tested; when both of them are met, we can use a slight modification of the previously introduced method to compute the *prefetch distance*, i.e., how long ago the prefetching occurred. These prefetch distances are then integrated into the micro-benchmark's setup list just like the access distances, only that for prefetch accesses the access is limited to one cache-line along an operand's first dimension.

Example 8. In algorithm *ca*-gemv (-·-··), for which we explicitly constructed the setup list in the previous section, both operands A[a,:] and C[a,:,b] meet both prefetching conditions: 1. they vary across the surrounding loop iterator a and 2. a indexes their first dimensions (sharing of cache-lines). As a result, their prefetch distances are 0 and the prefetching access will load the entire operands since their extension along the first, contiguously stored dimension is 1. Since the remaining operand B[:,:,c] has an access distance of 0, all operands are now accessed immediately before the kernel invocation; the setup list is reduced to

$$C[\mathtt{a}, :, \mathtt{c}], A[\mathtt{a}, :], B[:, :, \mathtt{c}].$$

(Since this setup consists only of accesses to the operands, it becomes redundant in our micro-benchmarks, because each of the ten repetitions will already touch all operands for the next repetition; hence, in such a case, we omit the setup altogether.)

Now accounting for prefetching, we obtain the performance estimates shown in Fig. 1d. Here, several algorithms, such as *ba*-gemv (-·-··), are estimated closer to their measured performance, leading to an improved average error of 19.1 %. However, several other algorithms, including *ca*-gemv (-·-··), are overestimated in performance (i.e., underestimated in execution time). There are two separate causes for this discrepancy.

- In several algorithms, such as *ca*-gemv (-·-··), where prefetching implicitly happens due to sharing of cache-lines, the prefetcher fails once a new cache-line is reached.

[9] The cache-line size is 64 B = 8 doubles.

– In other algorithms, such as *bi*-`ger` (- - -), the innermost loop is so short (here: 8 iterations) that each first iteration of the loop significantly impacts performance.

These two causes are treated separately in the following sections.

3.4 Prefetching Failures

For those algorithms in which certain operands are identified as prefetched because they share cache lines across iterations (i.e., the surrounding loop indexes their first dimension), the CPU would need to prefetch the next cache-line every 8 iterations (1 cache-line = 8 doubles). However, as a detailed analysis of hand-instrumented algorithms has shown, it fails to do so. As a result, in every 8th iteration of the innermost loop, the operand is not available and the kernel may take significantly longer.

We account for this prefetching-artifact by performing two separate micro-benchmarks: one simulating the 7 iterations in which the operand is available in cache as before, and one for the 8th iteration, where we account for the failure to prefetch and eliminate the emulated prefetching from our setup-list. The prediction for the total execution time is now obtained from weighting these two benchmark timings according to their number of occurrences in the algorithm and summing them up.

Example 9. In algorithm *ca*-`gemv` (-·-·-), the memory regions of both A`[a,:]` and C`[a,:,c]`, respectively, share cache-lines across iterations of the innermost loops over `a`. Hence, affecting not one but two of the kernel's operands, in every 8th iteration the kernel execution time increases drastically by a factor of about 4.5. To account for these "prefetching failures", we introduce a second set of micro-benchmarks, where the prefetching emulating accesses are removed from the setup list, resulting for $a = b = c = 400$ and $i = 8$, as without prefetching, in:

$$[816{,}632], A[\mathtt{a}, :], [163{,}200], B[:, :, \mathtt{c}].$$

Fig. 1e shows the predictions obtained after this improvement: the error is further reduced to 14.7 %. Most apparent in *ca*-`gemv` (-·-·-), the overestimation of algorithms where iterations share cache-lines are now corrected.

3.5 First Loop Iterations

The predictions for several algorithms, such as *ci*-`ger` (- - -), are still severely off, because the innermost loop of these algorithms is very short (in our example 8 iterations long). In such a case, the predictions are very accurate for all but the first iteration. Due to vastly different cache preconditions for this first iteration, however, its performance can be significantly different (in our case, up to 10× slower). Combined with the low total iteration count, this results in predictions that are off by a factor of up to 2.

To treat such situations, we introduce separate benchmarks to predict the performance of the first iteration of the innermost loop (and further loops if their first iterations account for more than 1 % of the total kernel invocations). For this purpose, the access distance evaluation method is slightly modified: instead of the kernel itself, the starting point is now the loop whose first iteration is considered, and the set M already contains all of the kernel's memory regions joined across this loop.

Example 10. In algorithm *ci*-`ger` (- - -), the innermost loop over i is in our example only 8 iterations long. For all but the first iteration, the operand C`[:,:,c]` stays the same, while A`[:,i]` and B`[i,:,c]` are prefetched, leading to optimal conditions for performance. In the first iteration (i.e., the next `c` iteration) however, C`[:,:,c]` refers to a different memory location and prefetching fails for both A`[:,i]` and B`[i,:,c]`, leading to severely lower performance.

From these improved access distances, the cache setup and micro-benchmark are performed just as before. As for the "prefetching failures", the prediction for the total execution time is now obtained from weighting of all relevant benchmark timings with the corresponding number of occurrences within the algorithm.

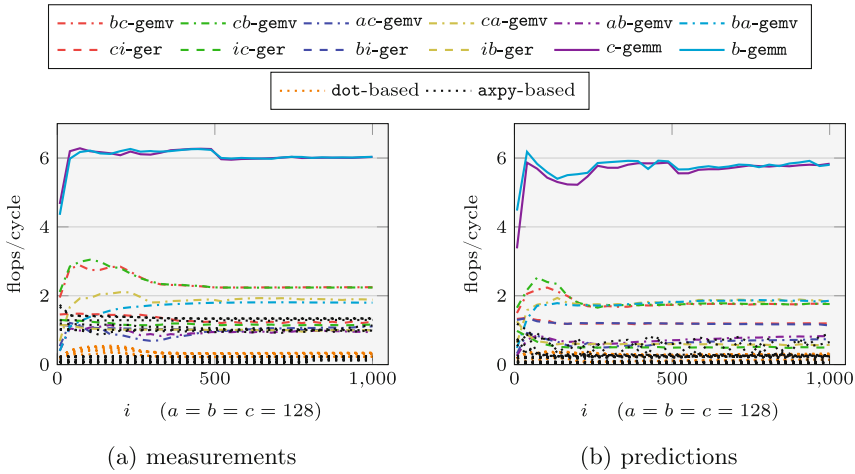
In Fig. 1f, we present the improved performance predictions obtained from this modification. The performance of all algorithms is now predicted with satisfying accuracy — the average absolute error is 9.47 %.

4 Results

In order to showcase its applicability and effectiveness, in this section we apply our technique for performance prediction to a range of contractions. We consider three test cases: In Sect. 4.1, we use different hard- and software, as well as changing the problem sizes. In Sect. 4.2, we consider a contraction that only allows the use of BLAS-1 and BLAS-2. Finally, in Sect. 4.3, we consider a more complex contraction with numerous alternative algorithms and multithreading.

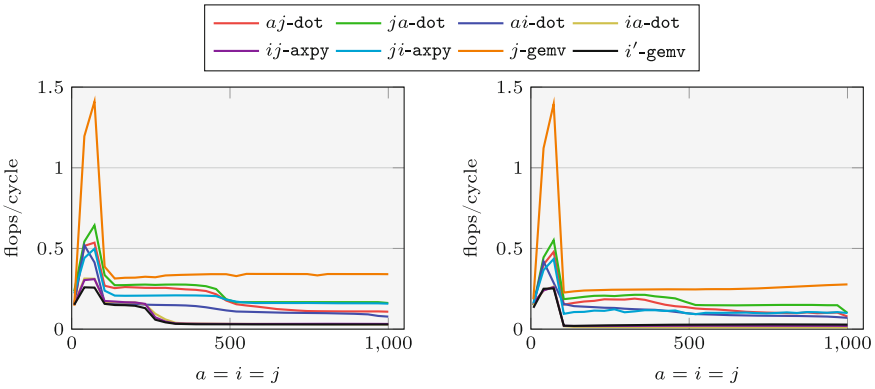
4.1 Test 1: $C_{abc} := A_{ai}B_{ibc}$, Different Setup

We commence with the same contraction used as a case study in the previous section, yet with an entirely different setup: the sizes a, b, and c are now fixed to 128, while i ranges from 8 to 1,000. As experimental environment, we use a 10-core Intel Ivy Bridge-EP E5-2680 v2 processor running at 3.6 GHz (Turbo Boost) and 25 MB of L3 cache. Each core can execute 8 double precision flops/cycle. The routines for both the actual measurements and the micro-benchmarks were linked to the BLAS implementation of Intel's Math Kernel Library (MKL, version 11.0). Figure 2 contains the performance measurements and corresponding predictions for all 36 algorithms (see Algs. 1). Although everything, ranging from the problem size to the machine and BLAS library was changed in this setup, the predictions are of equivalent quality and our tool correclty determines that the `gemm`-based algorithms (*c*-`gemm` (——) and *b*-`gemm` (——)) perform best and equally well.



(a) measurements (b) predictions

Fig. 2. $C_{abc} := A_{ai}B_{ibc}$, different setup: Performance measurements and predictions.



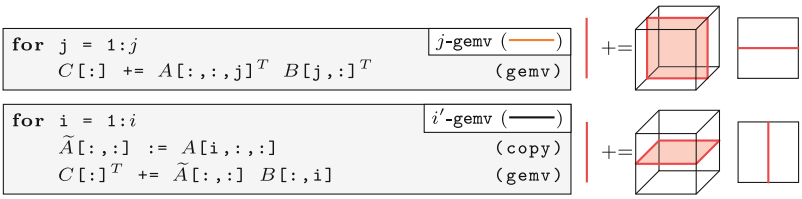
(a) measurements (b) predictions

Fig. 3. $C_a := A_{iaj}B_{ji}$: Performance measurements and predictions.

4.2 Test 2: $C_a := A_{iaj}B_{ji}$, only BLAS-1 and BLAS-2

For certain contractions (e.g., those involving 1D tensors), `gemm` cannot be used as a compute kernel, and only algorithms based on BLAS-2 or BLAS-1 are possible. One such scenario is encontered in the contraction $C_a := A_{iaj}B_{ji}$, for which our generator yields 8 algorithms:

- 4 `dot`-based: aj-`dot` (——), ja-`dot` (——), ai-`dot` (——), ia-`dot` (——),
- 2 `axpy`-based: ij-`axpy` (——), ji-`axpy` (——), and
- 2 `gemv`-based (see Algs. 2): j-`gemv` (——), i'-`gemv` (——).



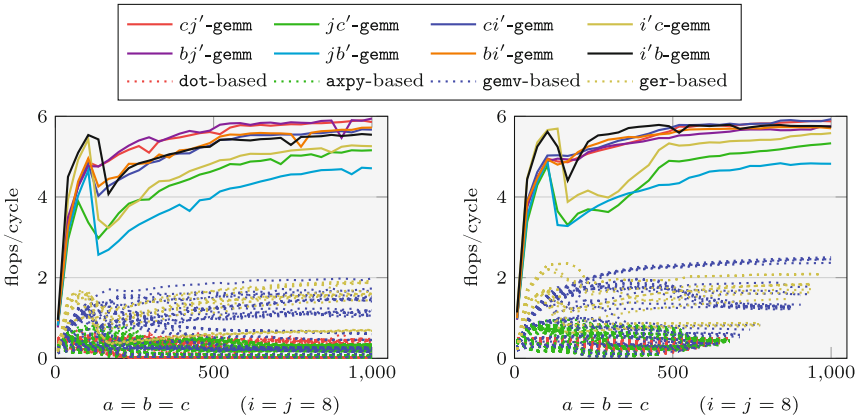
Algs. 2: $C_a := A_{iaj}B_{ji}$, gemv-based.

The measured and predicted performance for these algorithms is shown in Fig. 3. Our predictions clearly discriminate the fastest algorithm j-gemv (——) across the board. Furthermore, the next group of four algorithms is also correctly identified and the low performance of the second gemv-based algorithm i'-gemv (——) (due to the overhead of the involved matrix-copy operation) is predicted too.

4.3 Test 3: $C_{abc} := A_{ija}B_{jbic}$, Challenging Contraction

We now turn to a more complex example inspired by space-time continuum computations in the field general relativity [1]: $C_{abc} := A_{ija}B_{jbic}$. For this contraction, we generated a total of 176 different algorithms:

- 48 dot-based (·····),
- 72 axpy-based (·····),
- 36 gemv-based (·····),
- 12 ger-based (·····), and
- 8 gemm-based:
  cj'-gemm (——), jc'-gemm (——), ci'-gemm (——), $i'c$-gemm (——), bj'-gemm (——), jb'-gemm (——), bi'-gemm (——), $i'b$-gemm (——).



(a) predictions (b) measurements

Fig. 4. $C_{abc} := A_{ija}B_{jbic}$: Performance prediction and measurements.

All gemm-based (see Algs. 3) and several of the gemv-based algorithms involve copy operations to ensure that each matrix has a contiguously stored dimension, as required by the BLAS interface. Once again, we consider a very challenging scenario where both contracted indices are of size $i = j = 8$ and the free indices $a = b = c$ vary together.

```
for c = 1:c                                          cj'-gemm (——)
    for j = 1:j
        B̃[:,:] := B[j,:,:,c]                                (copy)
        C[:,:,c] += A[:,j,:]^T B̃[:,:]^T                     (gemm)
```

```
for j = 1:j                                          jc'-gemm (——)
    for c = 1:c
        B̃[:,:] := B[j,:,:,c]                                (copy)
        C[:,:,c] += A[:,j,:]^T B̃[:,:]^T                     (gemm)
```

```
for c = 1:c                                          ci'-gemm (——)
    for i = 1:i
        Ã[:,:] := A[i,:,:]                                  (copy)
        C[:,:,c] += Ã[:,:]^T B[:,:,i,c]                     (gemm)
```

```
for i = 1:i                                          i'c-gemm (——)
    Ã[:,:] := A[i,:,:]                                      (copy)
    for c = 1:c
        C[:,:,c] += Ã[:,:]^T B[:,:,i,c]                     (gemm)
```

```
for b = 1:b                                          bj'-gemm (——)
    for j = 1:j
        B̃[:,:] := B[j,b,:,:]                                (copy)
        C[:,b,:] += A[:,j,:]^T B̃[:,:]                       (gemm)
```

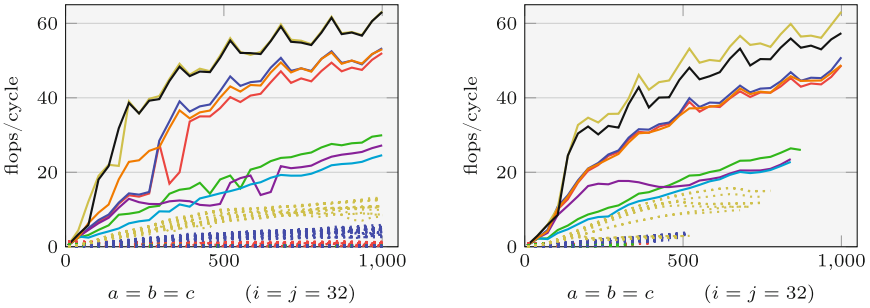
```
for j = 1:j                                          jb'-gemm (——)
    for b = 1:b
        B̃[:,:] := B[j,b,:,:]                                (copy)
        C[:,b,:] += A[:,j,:]^T B̃[:,:]                       (gemm)
```

```
for b = 1:b                                          bi'-gemm (——)
    for i = 1:i
        Ã[:,:] := A[i,:,:]                                  (copy)
        C[:,b,:] += Ã[:,:]^T B[:,b,i,:]                     (gemm)
```

```
for i = 1:i                                          i'b-gemm (——)
    Ã[:,:] := A[i,:,:]                                      (copy)
    for b = 1:b
        C[:,b,:] += Ã[:,:]^T B[:,b,i,:]                     (gemm)
```

Algs. 3: $C_{abc} := A_{ija}B_{jbic}$, gemm-based.

Starting with the predictions, in Fig. 4a, we present the expected performance in flops/cycle of the 176 algorithms, where BLAS-1 and BLAS-2 algorithms are grouped by kernel. Even with the copy operations, the gemm-based algorithms are the fastest. However, within these 8 algorithms, the performance differs by more than 20 %. Focusing on the gemm-algorithms, we compare with corresponding



(a) predictions (b) measurements

Fig. 5. $C_{abc} := A_{ija}B_{jbic}$: Performance prediction and measurements with 10 threads.

performance measurements[10] in Fig. 4b. The comparison shows that our predictions clearly separate the bulk of fast algorithms from the slightly less efficient ones.

Multithreading. The algorithms considered here can make use of shared memory parallelism by employing multithreaded BLAS kernels. To focus on the impact of parallelism, we increase the contracted tensor dimension sizes to $i = j = 32$ and use all 10 cores of the Ivy Bridge-EP CPU with OpenBLAS.

Performance predictions and measurements for this setup are presented in Fig. 5. Our predictions correctly separate the three groups of `gemm`-based implementations; moreover, algorithms $i'c$-`gemm` (——) and $i'b$-`gemm` (——) (see Algs. 3), which reach 60 flops/cycle,[11] are identified as the fastest. The slowest algorithm (jb'-`gemm` (——)) on the other hand merely reaches 20 flops/per cycle. This 3× difference in performance among `gemm`-based algorithms emphasizes the importance of selecting the right algorithm.

4.4 Efficiency Study

The ultimate goal of this work is to automatically and quickly select the fastest algorithm for a given tensor contraction. The experiments presented so far provide evidence that our automated approach successfully identifies the fastest algorithm(s). In this last study, we investigate the efficiency of our micro-benchmark-based approach. For this purpose, we once more consider the contraction $C_{abc} := A_{ai}B_{ibc}$, with $i = 8$ and varying $a = b = c$. Figure 6 displays the ratio of how much faster our micro-benchmark is compared to executing the corresponding algorithm. In general, our prediction proves to be several orders of magnitude faster than the algorithm itself. At $a = b = c = 1{,}000$, this relative improvement is smallest for the `gemm`-based algorithms (——) at $10^3\times$, since

[10] Slow tensor contraction algorithms were stopped before reaching the largest test-cases by limiting the total measurement time per algorithm to 15 minutes.

[11] Using 10 cores, the theoretical peak performance is 80 flops/cycle.

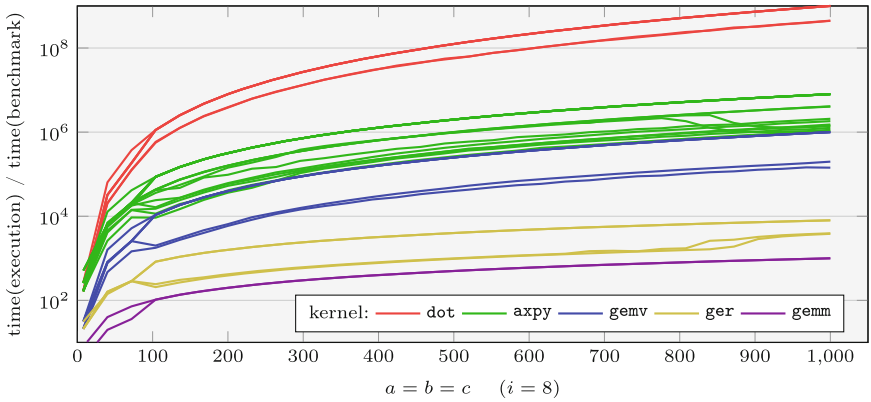


Fig. 6. $C_{abc} := A_{ai}B_{ibc}$: Prediction efficiency.

each `gemm` performs a significant portion of the computation; for the `ger`-based algorithms (——), it lies between $6 \cdot 10^3$ and $10^4\times$ and for the `gemv`-based algorithms (——) the gain is $5 \cdot 10^5$ to $10^6\times$; finally, the gain for both BLAS-1-based algorithms (——,——), where each BLAS-call only performs a tiny fraction of the contraction, our prediction is between 6 and 9 orders of magnitude faster than the execution.

5 Conclusion

In this paper, we focused on the performance prediction of BLAS-based algorithms for tensors contractions. First, based on previous work, we developed an algorithm and code generator that given the mathematical description of a tensor contraction, casts the computation in terms of five different BLAS kernels; since, in general, a tensor contraction may be decomposed in terms of matrix and vector products in many different ways, the generator often returns dozens of alternative algorithms.

Then, we tackled the problem of selecting the fastest algorithms without ever executing them. Instead of executing the full algorithms, our approach is based on timing the BLAS kernels in a small set of micro-benchmarks. These micro-benchmarks are run in a context that emulates that of the actual computation; thanks to careful treatment of cache-locality and a model of the cache prefetcher's behavior, our performance prediction tool is capable of identifying the best-performing algorithms in a tiny fraction of the time required to actually run and time all of them.

The quality of the predictions was showcased for a number of challenging scenarios, including contractions among tensors with small dimensions, contractions that can only be cast in terms of BLAS 1 and BLAS 2 kernels, and multi-threaded computations.

Acknowledgments. Financial support from the Deutsche Forschungsgemeinschaft (DFG) through Grant GSC 111 and the Deutsche Telekom Stiftung is gratefully acknowledged.

References

1. Kidder, L.E., Scheel, M.A., Teukolsky, S.A.: Extending the lifetime of 3d black hole computations with a new hyperbolic system of evolution equations. Phys. Rev. D **64**, 064017 (2001)
2. Lehner, L.: Numerical relativity: a review. Class. Quantum Gravity **18**(17), R25 (2001)
3. Helgaker, T., Jorgensen, P., Olsen, J.: Molecular Electronic-Structure Theory. Wiley, Chichester (2000)
4. Čížek, J.: On the correlation problem in atomic and molecular systems calculation of wavefunction components in ursell type expansion using quantum-field theoretical methods. J. Chem. Phys. **45**(11), 4256–4266 (1966)
5. Bartlett, R.J., Musiał, M.: Coupled-cluster theory in quantum chemistry. Rev. Mod. Phys. **79**, 291–352 (2007)
6. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Softw. **5**(3), 308–323 (1979)
7. Dongarra, J.J., Du Croz, J., Hammarling, S., Hanson, R.J.: An extended set of fortran basic linear algebra subprograms. ACM Trans. Math. Softw. **14**(1), 1–17 (1988)
8. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (1990)
9. Baumgartner, G., Auer, A., Bernholdt, D., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R., Hirata, S., Krishnamoorthy, S., Krishnan, S., Lam, C., Lu, Q., Nooijen, M., Pitzer, R., Ramanujam, J., Sadayappan, P., Sibiryakov, A.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proc. IEEE **93**(2), 276–292 (2005)
10. Lu, Q., Gao, X., Krishnamoorthy, S., Baumgartner, G., Ramanujam, J., Sadayappan, P.: Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions. J. Parallel Distrib. Comput. **72**(3), 338–352 (2012)
11. Di Napoli, E., Fabregat-Traver, D., Quintana-Orti, G., Bientinesi, P.: Towards an efficient use of the blas library for multilinear tensor contractions. Appl. Math. Comput. **235**, 454–468 (2014)
12. Iakymchuk, R., Bientinesi, P.: Modeling performance through memory-stalls. SIGMETRICS Perform. Eval. Rev. **40**(2), 86–91 (2012)
13. Iakymchuk, R., Bientinesi, P.: Execution-less performance modeling. In: Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High-Performance Computing Systems (PMBS 2011) held as part of the Supercomputing Conference (SC 2011), Seattle, USA, November 2011
14. Peise, E., Bientinesi, P.: Performance modeling for dense linear algebra. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. SCC 2012, pp. 406–416. IEEE Computer Society, Washington, DC, USA (2012)
15. OpenBLAS. http://xianyi.github.com/OpenBLAS