Tensor computations – in particular tensor contraction (TC) – are important kernels in many scientific computing applications (SCAs). Due to the fundamental similarity of TC to matrix multiplication (MM) and to the availability of optimized implementations such as the BLAS, tensor operations have traditionally been implemented in terms of BLAS operations, incurring both a performance and a storage overhead. Instead, we implement TC using the much more flexible BLIS framework, which allows for reshaping of the tensor to be fused with internal partitioning and packing operations, requiring no explicit reshaping operations or additional workspace. This implementation, TBLIS, achieves performance approaching that of MM, and in some cases considerably higher than that of traditional TC. Our implementation also supports multithreading using an approach identical to that used for MM in BLIS, with similar performance characteristics. The complexity of managing tensor-to-matrix transformations is also handled automatically in our approach, greatly simplifying its use in SCAs.

References
BLIS: github.com/flame/blis, shpg.ices.utexas.edu/publications.htm

Leveraging the BLIS Framework
BLIS extends the high-performance Goto approach to matrix multiplication, such as used in GotoBLAS and OpenBLAS, by adding two partitioning loops in plain C, and only requiring a small microkernel to be highly optimized in assembly.

TBLIS uses this five-loop framework to perform tensor contraction by logically treating the tensor as a matrix during partitioning, and only requiring extra tensor-aware steps (yellow and green above) during packing of the input tensors and updating the output tensor. The high-performance microkernel and cache blocking parameters are used unchanged, and overall overhead is small.

Threading is simple to implement, and can be added hierarchically to four out of the five loops, leading to high scalability.

Tensor Contraction
Tensor contraction:
- Transpose-transpose-DGEMM-transpose (TTDT), which reshapes (transposes) the tensors into matrices.
- Loop-over-GEMM (Log), which uses explicit loops over a small GEMM operation.

A high-performance “native” algorithm is ideal:
- Large memory overhead.
- Transpose does not parallelize well.
- Difficult to optimize out transposes.

“TTDT”
“Log”
“Native” (TBLIS)

Tensor as Matrices: The Block-Scatter Matrix
Tensor $T$ Above $6 \times 3 \times 2 \times 3 \times 4$

Matrix $M_{(c|d)b}(ac)$ $18 \times 24$

Column “$cd$” dimension has stride of 12 (6x3x18) most of the time. Blocking lets us use this constant stride when possible, and a scattered algorithm otherwise.

Row “$ae$” dimension similarity is stride-1 most of the time. (i.e. M is “row-major”)

$c[i]$ stores offset for each position in rows or columns. $r[i]$ stores stride for each block or zero for irregular blocks.

BLIS partitions the input matrices into successively smaller and smaller blocks, eventually reaching the small, fixed-size microkernel, such as 4x4 (pictured above), 8x4, 6x8, etc. These same “register blocksize”s are used during packing.

TBLIS uses this fact to encode tensor blocks as regular matrix blocks whenever possible, so that no overhead is incurred. Irregular blocks which do not have a constant row or column strides are handled specially by storing the offset for each row and column individually.

The tensor dimensions can also be reordered to assure unit-stride access in at least two of the tensors (see next panel for the effects of non-unit-stride access).

Results and Conclusions
Random “square” tensor contractions with TTDT and TBLIS, compared to matrix multiplication. Even for “square” cases where overhead is minimized, TTDT incurs as much as a 50% penalty, while TBLIS is consistent at 5-10%. Since TBLIS uses the BLIS microkernels and blocking parameters, performance improvements in BLIS and porting to new architectures automatically carries over to TBLIS.

Speedup of TBLIS over TTDT for the GETT benchmark (arXiv:1607.00145), which spans memory-bound contractions (left) to compute-bound contractions (right). TTDT is efficient for compute-bound problems because transposition is only O(n^2), but TBLIS speedup increases sharply for more memory-bound problems. The left-most contractions have non-unit-stride access during packing, which hinders TBLIS. This overhead can be overcome by using a 3D packing kernel (future work).