

Automatically tuned libraries for native-dimension tensor transpose and contraction algorithms

Jeff Hammond

June 1, 2011

Abstract

1 Introduction

Many numerical algorithms, particularly those of quantum many-body theory, rely heavily upon procedures called tensor contractions. Tensor contractions (TC) are the multi-dimensional generalization of matrix multiplication (MM). Whereas in MM, one has only a single internal (contracted) index and the two external indices of the matrix,

$$C_j^i = \sum_k A_k^i B_j^k, \quad (1)$$

a tensor contraction may have an arbitrary number of both. One simple example from quantum chemistry is

$$R_{i,j}^{a,b} = \sum_{c,d} V_{c,d}^{a,b} T_{i,j}^{c,d}. \quad (2)$$

While Eqn. 2 is isomorphic to MM upon fusion of the three pairs of indices, other similar contractions with permuted indices are not. An example of a TC that cannot be performed with MM alone is

$$R_{i,j}^{a,b} = \sum_{k,c} \tilde{V}_{c,j}^{k,b} T_{i,k}^{a,c}, \quad (3)$$

presuming that we are utilizing the straightforward layout of these objects in memory. That the objects R and T in Eqns. 2 and 3 are the same, the best one can do is to choose a layout which is optimal for the most expensive TC and use a sub-optimal ordering for the other. Alternatively, one can change the memory layout sub-optimally-ordered tensors via a transpose-like operation. A much more complex approach is to use Morton-ordering [1] (also known as Z -ordering), or more generally, hierarchical tiling [2], to improve the performance of all tensor contractions, but then it is not possible to use existing implementations of MM, such as BLAS.

The complexity introduced by the transposition of indices in TCs presents a significant challenge to programmers. If one hand-codes procedures which are not MM, then a significant performance loss is incurred, as MM kernels are perhaps the most optimized in all numerical computation. Alternatively, one can retain the use of fast MM kernels by realigning the memory layout such that operations like Eqn. 3 can be performed with MM.

The transformation of Eqn. 3 to a form which is consistent with matrix multiplication is as follows:

$$T_{i,k}^{a,c} \rightarrow T1(i, k, a, c) \quad (4)$$

$$\tilde{V}_{c,j}^{k,b} \rightarrow V1(c, j, k, b) \quad (5)$$

$$R_{i,j}^{a,b} \rightarrow R1(i, j, a, b) \quad (6)$$

$$T2(i, a, k, c) = T1(i, k, a, c) \quad (7)$$

$$V2((k, b), (c, j)) = V1((c, j), (k, b)) \quad (8)$$

$$V3(k, c, b, j) = V2(k, b, c, j) \quad (9)$$

$$R2((i, a), (b, j)) = \text{SUM}[(k, c)] V3((k, c), (b, j)) * T2((i, a), (k, c)) \quad (10)$$

$$R1(i, j, a, b) = R2(i, a, b, j) \quad (11)$$

where the matrix dimensions used in the matrix transpose and multiplication calls are denoted with parentheses. Row-major ordering (the last index is stride-1) is presumed throughout. Equations 7, 9 and 11 correspond to tensor transpose (TT) operations. Since matrix transpose is not a standard BLAS operation, Eqn. 10 will be treated as another TT. However, when a vendor BLAS library (i.e. IBM's ESSL) contains a matrix transpose procedure, it should be used instead if the performance justifies it.

For a rank- n tensor, there are $n!$ possible permutations of the indices, and writing fast code for each of these procedures by hand is impractical for

$n > 4$. The automatic generation of code for these procedures is the subject of this chapter.

2 Background

The target application for this project was the coupled-cluster codes within TCE module of the quantum chemistry package NWChem [3]. Because most of the code within the TCE module was written by a code generator, it employs a simple structure which is easily modified. The TCE module also has few, if any, manually optimized procedures and thus suffers in performance with respect to the best hand-written packages. In the particular case of TT, four subroutine calls, `tce_sortN` ($N=2,4,6,8$), were used to perform every associated array permutation. Nearly identical sort-accumulate calls (`tce_sortaccN`, $N=2,4,6,8$) have use the same code except with “+” instead of “=”. Replacing these procedure with faster ones would result in increased performance throughout the code.

Is it not unreasonable to question the utility of optimizing permutations at all. The permutation of a n -d array requires N^n floating-point operations (flops), where N is the rank of each dimension, whereas contracting an n -d array with an m -d array over k indices requires N^{m+n-k} flops. However, the number of memory operations (mops) required to permute a 4-d array or contract two 4-d arrays over 2 indices is $C \cdot N^4$ where C is 2 for permutation (1 read, 1 write) and 3 for contraction (2 reads, 1 write). On modern processors, mops are so expensive that some have said that flops can almost be ignored. MM achieves a large percentage of machine peak by obscuring memory latency through data reuse, which is possible because flops/mops is large. Since the performance of permutations is memory-bandwidth-limited, it is unreasonable to expect a large percentage of peak performance. At the same time, improper implementation of these procedures can be extraordinarily expensive. Unlike MM, the flow of data during permutation is necessarily not optimal since at least half of the mops will not be stride-1.

If we assume flops are free and that performance is determined by the number and type of mops occurring, then permutation, not MM will be the more expensive procedure of the two. Since theoretical analyses are rarely quantitative, the relative cost of the two procedures has been measured using profiling techniques. Both the GNU profiler gprof [4] and TAU [5] were used to profile the code to ensure correct measurements. TAU profiling results are

not reported as they do not differ significantly from those of gprof.

3 Results

All results are for a single water molecular at the equilibrium geometry. Calculations were performed without point-group symmetry using spherical angular functions. The tile size for the virtual orbitals (VO) was no greater than 32. For the cc-pVDZ, cc-pVTZ and cc-pVQZ basis sets, there were 2, 4 and 8 virtual orbital tiles with average dimension 19, 26.5 and 27.5, respectively. Using a larger tile size favors `dgemm` performance, while smaller favors `tce_sortN`.

3.1 Profiling of CCSD within NWChem

First it was established empirically that the tensor transpose operation is a significant portion of the wall time, as predicted by the aforementioned theoretical analysis. In Table 1, the results of profiling are given for computing the CCSD ground-state energy. When computing the CCSD energy, the coupled-cluster equations (described in previous chapters) are solved iteratively. The energy evaluation two orders less expensive than the iterative procedure and does not contribute significantly to the computational cost. The data given in Table 1 shows that MM and transpose both contribute significantly to wall time. However, the fraction of the wall time devoted to MM grows with the basis set, so it is not entirely clear that the optimality of the transpose will matter for larger systems. It should be noted that the MM implementation used was from NETLIB. A high-performance BLAS library such as GotoBLAS [6] or ATLAS [7] would greatly decrease the time spent on MM operations.

As should be clear from previous chapters, computing the ground-state energy is but one of many possible tasks for a coupled-cluster code. In Table 2, profiling information is given for the evaluation of all steps necessary to compute the hyperpolarizability using the method described in Chapter 8. The number of difficult transposes required for the solution of the $\Lambda^{(0)}$, $T^{(1)}$ and $\Lambda^{(1)}$ equations is significantly larger than required just for $T^{(0)}$, which is affirmed by the data. The relative amount of time spent in the transpose operations is approximately 50% greater than that spent in MM for the cc-pVQZ basis set, and while the overall trend in the basis set is the same as

Table 1: Profile (`gprof`) of the NWChem TCE module CCSD code for computing the ground-state energy.

| Basis | Matrix multiplication | | Tensor transpose | |
|---------|-----------------------|------------|--------------------------|------------|
| | dgemm | | tce_sort4 & tce_sortacc4 | |
| | Time (s) | % of Total | Time (s) | % of Total |
| cc-pVDZ | 0.40 | 27.59 | 1.49 | 29.65 |
| cc-pVTZ | 8.70 | 30.87 | 34.15 | 37.22 |
| cc-pVQZ | 154.46 | 38.54 | 108.47 | 27.07 |

Table 2: Profile (`gprof`) of the NWChem TCE module CCSD code for computing the hyperpolarizability.

| Basis | dgemm | | tce_sort4 & tce_sortacc4 | |
|---------|----------|------------|--------------------------|------------|
| | Time (s) | % of Total | Time (s) | % of Total |
| | cc-pVDZ | 5.78 | 28.18 | 11.08 |
| cc-pVTZ | 111.10 | 28.44 | 192.45 | 49.26 |
| cc-pVQZ | 1389.05 | 29.16 | 2137.17 | 44.87 |

Table 1, rate of which MM increases and transpose decrease is much less.

3.2 Autotuning transpose kernels

It was determined that the primary reason transpose operations are slow is that they access memory in a suboptimal way, that is, strided access rather than sequential (stride-1) access. While it is not possible to eliminate strided access, it is possible to minimize the cost of strided access by rearranging the loops such that the stride distance is minimal. If the stride distance is small enough that cache reuse occurs, a significant performance increase will result.

While it is possible to determine optimal loop ordering using mathematical analysis, a much cruder approach — exhaustive sampling — is sufficient in this case. In addition, sampling includes all possible hardware-specific factors which may not be available for integration into a performance mod-

eling used in the analytic approach. To determine the optimal loop-ordering for the 4-d transpose problem, a code-generator was developed which would produce source code for all possible implementations (24) for each of the 24 transposes, for a total of 576 cases. Source code was generated in both Fortran 77 and ANSI C since the former is known to be more amenable to compiler optimization, while the latter allows a more complete set of compiler pragmas and is the language of choice of people who would further hand-tune these kernels. A master program was instrumented to compile the source code into binary form using a variety of possible compiler flags to determine the effect of available optimization options. Some of the optimizations sampled for the Intel compilers were loop-unrolling, auto-vectorization and auto-parallelization; compiler pragmas were also explored as a means to explicitly control unrolling and vectorization. The master program built a self-contained binary for each possible transpose which, when executed, performed the timing and printed a complete table of results then identified the optimal loop-ordering. It also prints the compiler flags which were used to generate the code to prevent data rot.

Table 3 shows the best improvement obtained with the automatically-generated code as compared to the original implementation within NWChem by So Hirata. Four cases were considered: regular 4-d arrays of rank 20, 32 and 60 plus an irregular array. The speed-up for the rank 20 case is significantly better than the others because both the input and output array (1,250 KB each) fit into cache on the machine tested (Intel Core2Duo, 4 MB L3 cache). For larger dimensions, the arrays do not fit into cache. This clearly indicates that L3 cache-blocking will significantly improve the transpose performance, although finding the optimal code with that additional level of complexity becomes harder. Instead of performing an exhaustive search over just the space of loop-orderings or compiler options, an exhaustive search for the cache-blocking case involves exploring the tensor product space of blocking sizes and loop-orderings for each level of blocking. The dimensionality here is too large to consider by brute force, and a space-pruning algorithm must be employed to make the solution achievable in a reasonable amount of time.

In addition to the 4-d case, exhaustive search was used to find the best implementation of the subset of 6-d transpose-accumulate operations used in CCSD(T). Because of memory constraints imposed by the use triple-excitation amplitudes, dimensions of the arrays are much smaller. Due to the smaller stride length, cache-blocking is less important and the per-

Table 3: Best improvement relative to the original implementation by the ANSI C automatically-generated implementation of the transpose operations for a 4-d array. The Intel 10.1 compiler flags used were `-O3 -xT -march=core2 -mtune=core2 -funroll-loops -align`.

| Transpose | 20 ⁴ | 32 ⁴ | 60 ⁴ | irregular |
|-----------|-----------------|-----------------|-----------------|-----------|
| 1234 | 7.250 | 2.500 | 1.946 | 3.769 |
| 1243 | 7.667 | 2.345 | 2.257 | 2.733 |
| 1324 | 5.000 | 1.828 | 1.861 | 2.667 |
| 1342 | 5.250 | 2.379 | 2.173 | 2.923 |
| 1423 | 7.000 | 2.448 | 2.272 | 2.929 |
| 1432 | 5.250 | 2.000 | 2.372 | 3.154 |
| 2134 | 5.250 | 2.000 | 1.967 | 2.583 |
| 2143 | 8.334 | 2.586 | 2.108 | 2.786 |
| 2314 | 5.250 | 2.000 | 2.028 | 2.583 |
| 2341 | 5.000 | 2.267 | 2.179 | 3.000 |
| 2413 | 7.000 | 2.571 | 2.390 | 2.857 |
| 2431 | 5.000 | 1.889 | 2.756 | 3.385 |
| 3124 | 5.250 | 1.862 | 1.966 | 2.538 |
| 3142 | 5.000 | 3.233 | 2.216 | 3.000 |
| 3214 | 5.250 | 2.143 | 2.104 | 2.833 |
| 3241 | 7.000 | 1.971 | 2.219 | 2.571 |
| 3412 | 5.000 | 1.838 | 2.208 | 2.692 |
| 3421 | 6.334 | 1.976 | 2.281 | 2.429 |
| 4123 | 6.333 | 2.655 | 2.228 | 2.875 |
| 4132 | 4.750 | 1.944 | 2.202 | 3.000 |
| 4213 | 5.250 | 2.821 | 2.326 | 2.786 |
| 4231 | 5.250 | 2.195 | 2.299 | 3.077 |
| 4312 | 4.750 | 1.973 | 2.120 | 3.308 |
| 4321 | 4.500 | 1.767 | 2.163 | 3.077 |

irregular = $41 \times 17 \times 24 \times 39$

formance improvement realized just by finding the optimal loop-ordering (among $6! = 720$ possibilities) is quite good. The performance improvement realized in preliminary attempts ranges from a factor of 3 to a factor of 12. This is the subject of ongoing research.

4 Conclusions

Tensor operations, which compose the overwhelming majority of quantum chemistry codes, require optimal implementations to take advantage of high-performance computers. It was demonstrated that tensors transpose is a significant contribution to the wall time for coupled-cluster calculations and that a very simple approach decrease the time devoted to these operations by a factor of two. The successful approach employed here did not employ cache-blocking or many other possible optimization techniques which will further improve the performance

Ultimately, this project demonstrates that the artificial separation of transpose and MM in the implementation of tensor contractions is wholly inappropriate. The original motivation for it was to take advantage of vendor-optimized BLAS libraries, but developments in autotuning over the past 10 years clearly indicate that it is possible to generate tensor contraction kernels directly. The advantage is not only with respect to performance, but also in terms of mathematical elegance. The many-body formalism of coupled-cluster theory is multidimensional and flattening the data structures used in such codes into matrices just to use BLAS should not be tolerated.

References

- [1] G. M. Morton, "A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing," IBM technical report (1966).
- [2] L. Carter, J. Ferrante, S. Flynn Hummel, B. Alpern and K.S. Gatlin, UCSD Tech Report CS96-508, November 1996; L. Carter, J. Ferrante and S. Flynn Hummel, *Int. Par. Proc. Symp.*, April 1995; L. Carter, J. Ferrante and S. Flynn Hummel, *SIAM Conf. on Par. Proc. for Sci. Comp.*, February 1995.

- [3] E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, D. Wang, E. Apra, T. L. Windus, J. Hammond, J. Autschbach, P. Nichols, S. Hirata, M. T. Hackler, Y. Zhao, P.-D. Fan, R. J. Harrison, M. Dupuis, D. M. A. Smith, J. Nieplocha, V. Tipparraju, M. Krishnan, A. Vazquez-Mayagoitia, Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen, L. D. Crosby, E. Brown, G. Cisneros, G. I. Fann, H. Fruchtl, J. Garza, K. Hirao, R. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell, D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan, K. Dylla, D. Elwood, E. Glendening, M. Gutowski, A. Hess, J. Jaffe, B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X. Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M. Rosing, G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong and Z. Zhang. “NWChem, A Computational Chemistry Package for Parallel Computers, Version 5.1.1” (2007), Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA. A modified version.
- [4] gprof. Jay Fenlason and Richard Stallman. Copyright 1988, 92, 97, 98, 99, 2000, 2003 Free Software Foundation, Inc.
- [5] TAU: The TAU Parallel Performance System. S. Shende and A. D. Malony. *International Journal of High Performance Computing Applications*, Volume 20 Number 2 Summer 2006. Pages 287-331.
- [6] <http://www.tacc.utexas.edu/resources/software/>
- [7] <http://math-atlas.sourceforge.net/>