Blocking Techniques in Numerical Software

Wilfried N. Gansterer[1],
Dieter F. Kvasnicka[2], and Christoph W. Ueberhuber[1]

[1] Institute for Applied and Numerical Mathematics,
University of Technology, Vienna
ganst@aurora.tuwien.ac.at, christof@uranus.tuwien.ac.at
[2] Institute for Physical and Theoretical Chemistry,
University of Technology, Vienna
dieter@titania.tuwien.ac.at

Abstract. When developing high performance algorithms blocking is a standard procedure to increase the locality of reference. Conflicting factors which influence the choice of blocking parameters are described in this paper. These factors include cache size, load balancing, memory overhead, algorithmic issues, and others. Optimal block sizes can be determined with respect to each of these factors. The resulting block sizes are independent of each other and can be implemented in several levels of blocking within a program. A tridiagonalization algorithm serves as an example to illustrate various blocking techniques.

1 Introduction

In modern computer systems the gap between raw computing power and memory throughput is increasing tremendously. This leads to the effect that for many algorithms the processors are not running at full speed since they have to wait for data from memory.

One solution to this problem is to insert one or several levels of fast cache memory between main memory and the processor. These caches hold a subset of the data stored in main memory. If the processor accesses a data item from memory and this data item is resident in cache the memory access time is much shorter. The *locality of reference* is high if a high fraction of the memory accesses can be served from the high levels of the memory hierarchy, which consist (in top down order) of registers, one to three levels of caches, local memory, distributed memory, and secondary storage.

Many algorithms can be restructured to reuse data by rearranging the order of the operations performed. To exploit the higher speed of cache memory linear algebra algorithms are recast as block algorithms which operate on submatrices (blocks) of the original matrix. Partitioning a matrix into disjoint submatrices is referred to as *blocking* (see Section 2).

This work was supported by the Austrian Science Fund (*Österreichischer Fonds zur Förderung der wissenschaftlichen Forschung*).

Section 3 summarizes conflicting aspects which sometimes favor large blocks and under other circumstances favor small blocks. Section 5 describes a tridiagonalization algorithm, which serves as an example for the various blocking techniques discussed in this paper. Subsequent sections go into detail with these blocking methods.

2 Blocked Memory Access

The key to high-performance computing is to reuse data which have been transferred to a faster level of the memory hierarchy as often as possible before they are replaced by new data and restored at a slower level (see Lam et al. [29], Schreiber, Dongarra [32], Laderman et al. [28], Pan [31], Carr, Lehoucq [8]).

In order to transform a matrix algorithm in such a way that the highest levels of the memory hierarchy are exploited to a high degree, the algorithm is reformulated as a block algorithm which operates on submatrices (blocks) of the original matrix. (The concept of blocking as well as hierarchical blocking of matrix operations is explained in detail in Ueberhuber [34].)

On the one hand, the submatrices have to be chosen small enough such that for each operation of the block algorithm, all blocks needed as arguments can be stored in the respective level of the memory hierarchy. On the other hand, the blocks should be large enough so that the number of operations executed on elements of those blocks is as large as possible.

Other factors that influence the optimal block sizes are cache line sizes and page sizes. If a cache line is already transferred to a high level of the memory hierarchy, it should be used completely before fetching the next cache line. As a result of the complexity of all these factors, the optimal blocking strategy can often only be determined by way of experiment (see, for example, Anderson et al. [1]).

3 Block Size

At first sight the size of submatrices to be worked on should be chosen so that they fit into a certain level of the memory hierarchy. To exploit more than one memory level, several blocking levels, i. e., *hierarchical blocking*, can be applied (see Ueberhuber [34], Haunschmid, Kvasnicka [21]). However, as will be shown in this paper, blocking techniques are also influenced by aspects which were not mentioned in Section 2.

Large Block Sizes. The main reasons for choosing large block sizes are (i) the exploitation of all cache levels (Section 4), (ii) the reduction of the number of messages in data parallel implementations (Section 10), and (iii) the reduction of algorithmic complexity (obtained, for instance, using the Strassen algorithm or particular complex matrix multiplication algorithms; Sections 8 and 9).

Small Block Sizes. Reasons for choosing small block sizes are (i) the reduction of the number of Level 2 (matrix-vector) operations in relation to Level 3 (matrix-matrix) operations (Section 6), (ii) load balancing purposes in data parallel implementations (Section 10), (iii) algorithmic requirements, which can sometimes produce significant overhead when using large block sizes (Section 7), (iv) storage requirements (Section 11), and (v) memory throughput for complex floating-point numbers (Section 9).

All these aspects are investigated in more detail in the sections indicated above.

4 Basic Linear Algebra Subprograms

To account for the great variety of architectures available and in order to combine efficiency with portability, a set of matrix routines has been defined: the Basic Linear Algebra Subprograms (BLAS; see Lawson et al. [30], Dongarra et al. [11, 10]), which are optimized for each new computer system by hardware or compiler vendors. The determination of the optimal parameters may also be done automatically (see Bilmes et al. [2, 3], Whaley, Dongarra [36], Kagstrom et al. [25]). These hardware-optimized BLAS routines usually run at the highest level of floating-point performance achieveable on a certain computer.

Table 1 illustrates the reason for the advantage of Level 3 (matrix-matrix) BLAS by a comparison of the ratios of floating-point operations to data movement for three closely related operations from the Level 1, 2, and 3 BLAS (see Dongarra et al. [12]).

Table 1: Ratio of floating-point operations to data movement.

BLAS	Routine	Memory Accesses	Flops	Flops per Memory Access
Level 1	`daxpy`	$3n$	$2n$	$2/3$
Level 2	`dgemv`	n^2	$2n^2$	2
Level 3	`dgemm`	$4n^2$	$2n^3$	$n/2$

Numerical algorithms should exploit Level 3 BLAS as much as possible in order to achieve satisfactory floating-point performance on a wide variety of computers. This is precisely what can be achieved with block algorithms, where Level 3 BLAS routines are applied to submatrices. These routines yield higher performance for larger matrices (see Ueberhuber [34]).

5 Tridiagonalization

One particular algorithm whose performance is strongly determined by an appropriate way of blocking is the tridiagonalization algorithm. This algorithm

is used in eigensolvers to transform a symmetric input matrix into a similar tridiagonal matrix (see Golub, Van Loan [19] and Anderson et al. [1]).

Unblocked transformation of a symmetric $n \times n$ matrix A into a tridiagonal matrix $T = Q^T AQ$ requires the product $Q = H_{n-2} H_{n-3} \ldots H_1$ of Householder reflectors $H_i = I - \tau u_i u_i^T$. The vector u_i is chosen to eliminate the ith column of A except for the diagonal and subdiagonal elements.

The matrix Q is not explicitly accumulated, but each Householder reflector is applied to the matrix A as soon as it becomes available: $A^{(i)} = H_i^T A^{(i-1)} H_i$, with $A^{(0)} \equiv A$ and $A^{(n-2)} \equiv T$. The computation of T can be achieved by a series of rank-two updates to the matrix A (see Gansterer et al. [16]).

6 Blocked Tridiagonalization in One Sweep

LAPACK provides a blocked *one-sweep* tridiagonalization algorithm (see Anderson et al. [1] and Dongarra et al. [13, 14]). Like in most linear algebra routines, this blocked tridiagonalization routine computes a matrix panel from a block of columns and then applies this panel to the rest of the matrix in a rank-$2k$ update. The computation of this panel cannot be implemented using efficient Level 3 (matrix-matrix) operations, but has to be done vector-wise, based on Level 2 (matrix-vector) operations (see Dongarra et al. [13, 14]).

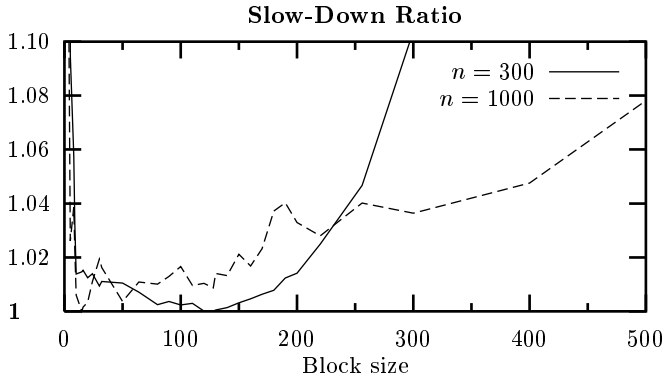

Fig. 1: Slow-down effect of suboptimal choice of the block size in the blocked one-sweep tridiagonalization. The minimum (optimum) execution time is scaled to 1 for each matrix size ($n = 300$ and $n = 1000$).

The optimum blocking parameter is heavily influenced by the amount of work which has to be done using Level 2 (matrix-vector) operations. The influence of the blocking parameter is illustrated in Fig. 1, which shows the slow-down effect for a suboptimal choice of the block size in the LAPACK routine `LAPACK/dsytrd` on one MIPS R8000 processor of an SGI Power Challenge. Note, the optimum block size decreases with increasing matrix size n, so that the current panel fits

into the largest cache. This is, however, a minor effect: for a 300×300 matrix block sizes from 6 to 300 lead to a performance variation which is within a 10 % range near the optimum floating-point performance. Execution times of this algorithm (and of the two-sweep algorithm presented in Chapter 7) are shown in Fig. 2.

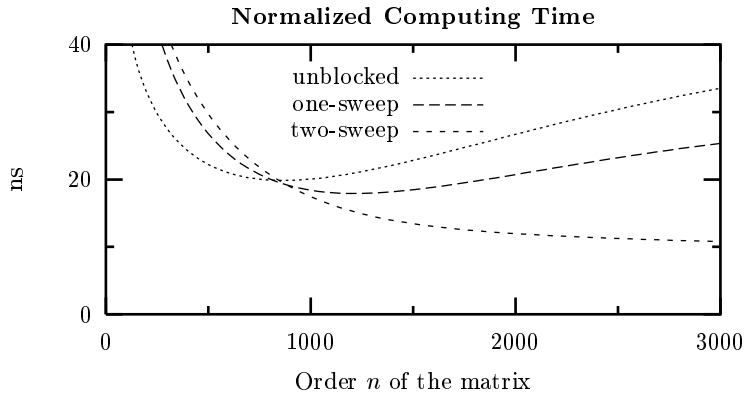

Fig. 2: Comparison of normalized execution times (T/n^3) of the unblocked, the one-sweep and the two-sweep tridiagonalization on one MIPS R8000 processor of an SGI Power Challenge.

The one-sweep tridiagonalization routine `LAPACK/dsytrd` is able to utilize Level 3 BLAS routines for about one half of the floating-point operations. The default blocking parameter for the tridiagonalization in LAPACK is set to 1, which was the best choice for many architectures at the time LAPACK was released. Nowadays, the blocking parameters can be adapted to any given architecture during the LAPACK installation process. However, most installations do not take advantage of this feature.

The Level 3 operations of the one-sweep algorithms occur as rank-$2k$ updates. The remaining half of the floating-point operations are performed as matrix-vector products, i. e., Level 2 operations. These matrix-vector products cannot be avoided in a one-sweep tridiagonalization (see Haunschmid, Kvasnicka [21] and Gansterer et al. [16]). Thus, one-sweep tridiagonalization of large matrices on cache-based computers may spend more than 80 % of the overall execution time in the 50 % of floating-point operations required by these matrix-vector products (see Haunschmid, Kvasnicka [21]).

To enable the utilization of Level 3 BLAS for all accesses to the full matrix, the algorithm has to be modified. A *two-sweep* algorithm has to be used (see Fig. 3, Gansterer, Kvasnicka [16], and Kvasnicka et al. [27]): A first sweep reduces the full matrix A to a band matrix B using Level 3 BLAS, and a second sweep reduces B to a tridiagonal matrix T. If the band width of B is chosen reasonably small,

most computational effort is spent in the first sweep, where fast Level 3 BLAS routines can be applied.

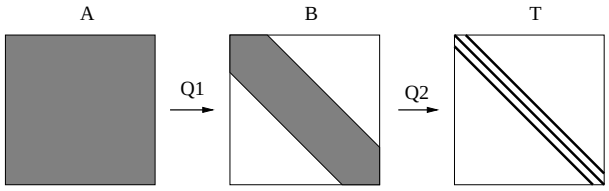

Fig. 3: Tridiagonalization in two sweeps.

7 Blocked Tridiagonalization in Two Sweeps

In the two-sweep tridiagonalization mentioned in Section 6, increasing the band width of B increases not only the number of operations which cannot be performed in Level 3 routines, but also increases the total number of operations.

Thus, optimum blocking enables the efficient use of Level 3 operations while the total number of operations is not increased excessively. The influence of the choice of the blocking parameter is shown in Fig. 4 for a two-sweep tridiagonalization using the routines `SBR/dsyrdb` and `LAPACK/dsbtrd` on one processor of an SGI Power Challenge.

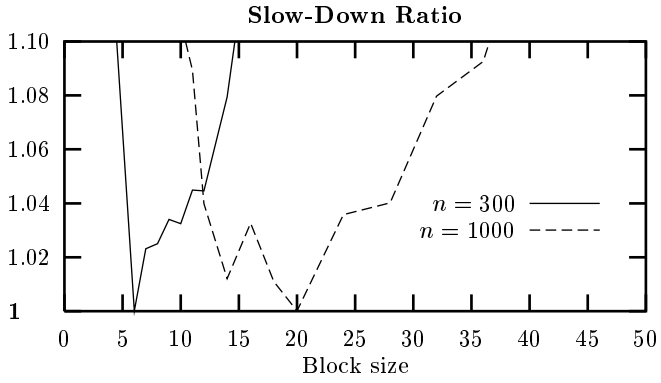

Fig. 4: Slow-down effect of suboptimal choice of blocking parameters in the blocked tridiagonalization. The minimum (optimum) execution time is scaled to 1 for each matrix size.

Note, the range of useful blocking parameters is narrower than for the one-sweep algorithm (see Fig. 1). The optimum block size increases with increasing n, because the overhead is of lower complexity than $O(n^3)$. Computing times

required by the three tridiagonalization algorithms discussed in this paper can be found in Fig. 2.

The SBR toolbox (see Bischof et al. [4–6]) is based on QR factorization to reduce a dense symmetric matrix to a band matrix. This algorithm (see Algorithm 1) reveals that, in contrast to the LAPACK blocking, no access to the rest of the matrix A has to be made to accumulate the update matrices. There are two accesses to the matrix A in the matrix update. Both of them are blocked with block size $\bar{p}$.

An implementation of a two-sweep reduction is included in the SBR toolbox (see Bischof et al. [4–6]). A complete algorithm for the symmetric eigenproblem is described in Gansterer, Kvasnicka [16, 17] and Kvasnicka et al. [27].

Algorithm 1

n is the order of the matrix A
$\bar{p}$ is the block size
$\bar{N} = (n-2)/\bar{p}$
$A^{(0)} = A$
do $k = 1, \bar{N}$
  $\bar{s} = (k-1)\bar{p} + 1$
  $\bar{r} = k\bar{p}$
  – QR factorization with $Q^{(k)} = I - W^{(k)}(Y^{(k)})^T$
  $(W^{(k)}, Y^{(k)}) = \mathrm{QR}(A^{(k-1)}(\bar{s} : \bar{r}))$
  – matrix update
  $A^{(k)} = (I - W^{(k)}(Y^{(k)})^T)^T A^{(k-1)}(I - W^{(k)}(Y^{(k)})^T)$
end do
$B = A^{(\bar{N})}$

8 Strassen's Algorithm

Using Strassen's algorithm for the multiplication of matrices of arbitrary order the complexity of this Level 3 operation is less than $28n^{\log_2 7}$ arithmetic operations (see Strassen [33]).

It is not easy to implement the Strassen algorithm efficiently. Thus, for a long time the Strassen algorithm was regarded to be of theoretical interest only. However, Douglas et al. [15] published efficient implementations for use on both single-processor and parallel computers. Some software libraries (e. g., IBM's ESSL) already comprise implementations of the Strassen algorithm.

On most computers the break-even point $n_{\min}$, i. e., the minimum matrix order for which a Strassen implementation takes less time than the corresponding Level 3 BLAS routine, is between 32 and 256. For larger matrices a significant complexity and run-time reduction is achieved (see Fig. 5 and Ueberhuber [34]).

Thus, to exploit the advantages of efficient implementations of the Strassen algorithm in blocked tridiagonalization the block size used in matrix-matrix multiplications should be chosen as large as possible.

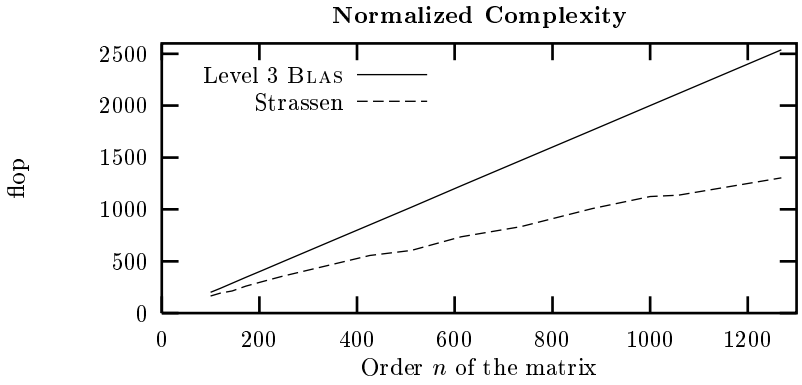

Fig. 5: Number *flop* of floating-point operations *per element* of the resulting matrix required by the Strassen algorithm and the Level 3 BLAS implementation based on standard matrix-matrix multiplication.

It has to be mentioned that the Strassen algorithm is less stable than the conventional method for multiplying two matrices. Similar to the modified error bounds for the multiplication of two complex matrices (cf. Section 9), the best bound known for the rounding error in Strassen's algorithm is a *normwise* and not an elementwise bound (Higham [23]). It turned out that in most practical cases the computed solution is of acceptable accuracy.

9 Multiplication of Complex Matrices

Blocked matrix multiplication algorithms are characterized by a high fraction of Level 3 operations. This can be utilized to reduce the overall complexity of the multiplication of complex matrices (see Winograd [37], Higham [24]). The multiply-add operation, $a = c \times d + e$, in its complex form

$$(a_r + a_i\mathrm{i}) = (c_r + c_i\mathrm{i}) \times (d_r + d_i\mathrm{i}) + (e_r + e_i\mathrm{i}),$$

can be written using four real multiplications and four additions

$$a_r = c_r \times d_r - c_i \times d_i + e_r, \qquad a_i = c_r \times d_i - c_i \times d_r + e_i$$

and in a form using only three real multiplications and seven additions

$$X_1 = c_r \times d_r, \qquad X_2 = c_i \times d_i, \qquad X_3 = (c_r + c_i) \times (d_r + d_i)$$

$$a_r = X_1 - X_2 + e_r, \qquad a_i = X_3 - X_1 - X_2 + e_i.$$

On modern computer architectures, the latter method has the major drawback of a larger number of add instructions. However, for blocked matrix algorithms, a new side of this method is revealed: A multiply-add operation on subblocks

requiring only three block-multiplies instead of four enables a great complexity reduction. A matrix multiply requires (roughly) n^3 multiply-add instructions. A matrix add requires n^2 add instructions[1]. So reducing the effort for the complex matrix-matrix multiply from $4n^3$ to $3n^3 + O(n^2)$ operations saves asymptotically 25 % of the required work.

Block Sizes. The aforementioned technique yields increasing complexity reduction with growing matrix size. For blocked algorithms matrix multiplications are applied to submatrices. So algorithms having large block sizes are to be preferred.

Nonetheless, if performance on a specific computer system is limited by memory throughput rather than processor performance, block sizes must not be too large. A multiply-add operation on complex data involves four multiplications and four additions on real data. The data transfer for complex data is only twice as much as for real data. So memory bound algorithms may have twice the floating-point performance when operating on complex data instead of real data.

If the block size is too large to store all required data in the first level cache the speed-up factor of two for memory bound complex algorithms is lost and only a speed-up factor of 4/3, resulting from complexity reduction, can be obtained. The block size has to be decreased in order to approach the full potential speedup of $2 \times 4/3$.

It has to be mentioned that the multiplication technique of this section is numerically less stable than conventional complex matrix-matrix multiplication (see Higham [24]). Nevertheless, when *normwise* instead of componentwise rounding errors are considered the resulting bounds are comparable to that of conventional complex matrix-matrix multiplication.

10 Blocking for Parallel Computing

In parallel computing the choice of an appropriate data distribution heavily depends on the data flow features of the involved algorithms. For dense matrix computations, ScaLapack routines (see Blackford et al. [7]), PLapack routines (see van de Geijn [35]), and HPF programs (see High Performance Fortran Forum [22]) achieve their best floating-point performance if a two-dimensional block-cyclic data layout scheme is used.

The two main issues in choosing a data layout for dense matrix computations are (i) load balancing, i. e., splitting the work reasonably well among the

[1] Addition operations require only add instructions and no multiplies, but since the time required for one add is the same as the time needed for one multiply-add on most of the modern processors the respective numbers of instructions are directly comparable. A different situation occurs when the algorithm is memory bound. In this case the add as well as the multiply instructions are executed faster than the fused multiply-add instructions because they require one memory transfer less (only two reads and one write instead of three reads and one write).

processors, and (ii) extensive use of Level 3 BLAS for local computations on all processors, to avoid excessive movement of data to and from memory.

Another issue is that on parallel computers with significant latency, block sizes must not be too small, i. e., communication patterns should not be vector oriented but matrix oriented. This technique is called *message combining* (see Geist et al. [18], Gropp et al. [20]).

In general it can be said that in parallel computing large block sizes are more favorable than small ones, as long as the work load is reasonably balanced between processors.

11 Storage Efficiency

Symmetric and triangular matrices (as arising in Sections 5, 6, and 7) require special storage techniques to avoid significant waste of memory. A tradeoff between floating-point performance and storage efficiency influences the choice of the optimum blocking technique.

Triangular and symmetric matrices can be stored using only half the memory requirement of LAPACK routines using conventional storage and at the same time rely on Level 3 BLAS. Efficient storage schemes are discussed in Haunschmid, Kvasnicka [21], Kvasnicka [26] (see Fig. 6), and D'Azevedo, Dongarra [9]. The storage overhead of some of these schemes is given in Fig. 7.

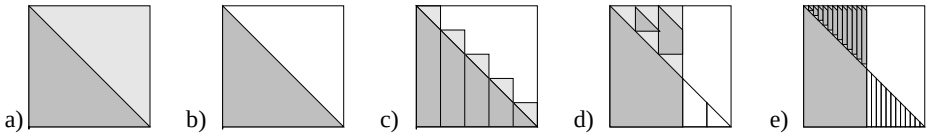

Fig. 6: Storage schemes for symmetric and triangular matrices: a) conventional (*unpacked*) storage, b) packed storage, c) block packed storage, d) and e) parallel packed storage scheme (d) $b = n/5$, e) $b = n/25$). The darker shaded areas indicate matrix elements whose values are stored and used; the lighter shaded parts indicate redundant storage.

Since the storage schemes c), d), and e) of Fig. 6 may lead to non-contiguous data layouts the respective algorithms become more complicated. In the algorithms, block sizes have to be used which are not larger than the block size of the underlying storage scheme.

12 Blocking Parameter Values

All blocking parameters have to be chosen in a problem and hardware dependent way. Typical values are

$$(32 \ldots 512) \geq b_{\text{parallel}} \geq b_{\text{storage}} \geq b_{\text{one-sweep}} \geq (32 \ldots 128) \geq b_{\text{two-sweep}} \geq (6 \ldots 16).$$

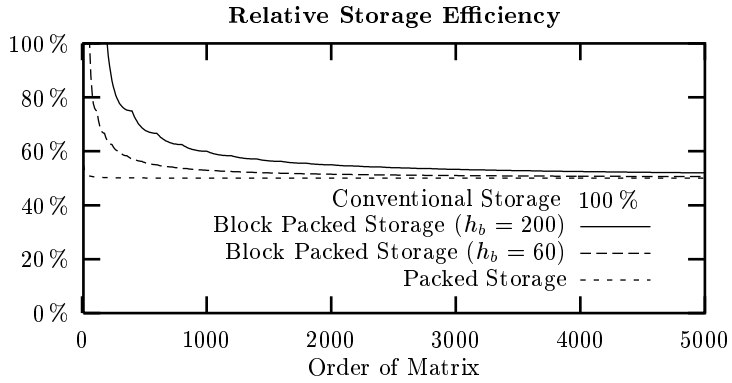

Fig. 7: Storage requirement of several storage formats. The ratio of required storage locations (words) to the total number of matrix elements (100 %) is plotted.

b_{parallel} denotes the block size for data distribution, b_{storage} the block size of the chosen storage scheme, $b_{\text{one-sweep}}$ the block size of the rank-$2k$ update of the one-sweep tridiagonalization (see Section 6), and $b_{\text{two-sweep}}$ the block size of the two-sweep tridiagonalization (see Section 7).

For local computations matrices only need to be blocked in one dimension. The other dimension can be as large as necessary, except for the complexity oriented blocking (see Section 9).

Level 3 BLAS routines are often called with rectangular matrices where one dimension is smaller by a factor of up to 100 than the other dimension. The smaller dimension should not be chosen below a certain hardware dependent level in order to avoid performance degradation.

Many block sizes will be chosen to be equal in practice. For instance, experience (made with SGI, DEC, HP, IBM, and SUN workstations) shows that values of

$$b_{\text{parallel}} = b_{\text{storage}} = b_{\text{lapack}} = 64 \qquad \text{and} \qquad b_{\text{two-sweep}} = 8$$

are a good choice for the tridiagonalization algorithms of Sections 6 and 7.

13 Conclusion

Choosing satisfactory blocking techniques and blocking parameters is not easy and straightforward. There are good reasons for choosing large blocks, for instance to better exploit the memory hierarchy. There are also good reasons for choosing small blocks, for instance, to reduce the number of floating-point operations which are not covered by Level 3 routines, or to reduce the total number of operations in some algorithm, or simply to save memory.

These conflicting reasons lead to totally different optimum block sizes, so that a portable *and* efficient algorithm requires several levels of blocking using different block sizes.

References

1. E. Anderson et al., LAPACK *Users' Guide, 2nd ed.*, SIAM Press, Philadelphia, 1995.
2. J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Proceedings of the International Conference on Supercomputing, ACM, Vienna, Austria, 1997, pp. 340–347.
3. J. Bilmes, K. Asanovic, J. Demmel, D. Lam, C.-W. Chin, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Technical report, LAPACK Working Note 111, 1996.
4. C. H. Bischof, B. Lang, X. Sun, *Parallel Tridiagonalization through Two-Step Band Reduction*, Proceedings of the Scalable High-Performance Computing Conference, IEEE, Washington D. C., 1994, pp. 23–27.
5. C. H. Bischof, B. Lang, X. Sun, *A Framework for Symmetric Band Reduction*, Technical report, Argonne Preprint ANL/MCS-P586-0496, 1996.
6. C. H. Bischof, B. Lang, X. Sun, *The SBR Toolbox - Software for Successive Band Reduction*, Technical report, Argonne Preprint ANL/MCS-P587-0496, 1996.
7. L. S. Blackford et al., SCALAPACK *Users' Guide*, SIAM Press, Philadelphia, 1997.
8. S. Carr, R. B. Lehoucq, *Compiler Blockability of Dense Matrix Factorizations*, ACM Trans. Math. Software 23 (1997), pp. 336–361.
9. E. F. D'Azevedo, J. J. Dongarra, *Packed Storage Extension for* SCALAPACK, Technical report, LAPACK Working Note 135, 1998.
10. J. J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, *A Set of Level 3* BLAS, ACM Trans. Math. Software 16 (1990), pp. 1–17.
11. J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, *An Extended Set of* BLAS, ACM Trans. Math. Software 14 (1988), pp. 18–32.
12. J. J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. van der Vorst, *Linear Algebra and Matrix Theory*, SIAM Press, Philadelphia, 1998.
13. J. J. Dongarra, S. J. Hammarling, D. C. Sorensen, *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations*, J. Comput. Appl. Math. 27 (1989), pp. 215–227.
14. J. J. Dongarra, S. J. Hammarling, D. C. Sorensen, *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations*, Technical report, LAPACK Working Note 2, 1987.
15. C. C. Douglas, M. Heroux, G. Slishman, R. M. Smith, GEMMW—*A Portable Level 3* BLAS *Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm*, J. Computational Physics 110 (1994), pp. 1–10.
16. W. N. Gansterer, D. F. Kvasnicka, *High Performance Computing in Material Sciences. The Standard Eigenproblem – Concepts*, Technical Report AURORA TR1998-18, Vienna University of Technology, 1998.
17. W. N. Gansterer, D. F. Kvasnicka, *High Performance Computing in Material Sciences. The Standard Eigenproblem – Experiments*, Technical Report AURORA TR1998-19, Vienna University of Technology, 1998.
18. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge London, 1994.
19. G. H. Golub, C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, 1996.
20. W. Gropp, E. Lusk, A. Skjelum, *Using MPI*, MIT Press, Cambridge London, 1994.

21. E. Haunschmid, D. F. Kvasnicka, *High Performance Computing in Material Sciences. Maximizing Cache Utilization without Increasing Memory Requirements*, Technical Report AURORA TR1998-17, Vienna University of Technology, 1998.
22. High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, 1997.
23. N. J. Higham, *Exploiting Fast Matrix Multiplication within the Level 3* BLAS, ACM Trans. Math. Software 16 (1990), pp. 352–368.
24. N. J. Higham, *Stability of a Method for Multiplying Complex Matrices with Three Real Matrix Multiplications*, SIAM J. Matrix Anal. Appl. 13–3 (1992), pp. 681–687.
25. B. Kagstrom, P. Ling, C. Van Loan, *GEMM-Based Level 3* BLAS*: High-Performance Model Implementations and Performance Evaluation Benchmark*, ACM Trans. Math. Software 24 (1998).
26. D. F. Kvasnicka, *Parallel Packed Storage Scheme (P2S2) for Symmetric and Triangular Matrices*, Technical Report to appear, Vienna University of Technology, 1998.
27. D. F. Kvasnicka, W. N. Gansterer, C. W. Ueberhuber, *A Level 3 Algorithm for the Symmetric Eigenproblem*, Proceedings of the Third International Meeting on Vector and Parallel Processing (VECPAR'98), Vol. 1, 1998, pp. 267–275.
28. J. Laderman, V. Pan, X.-H. Sha, *On Practical Acceleration of Matrix Multiplication*, Linear Algebra Appl. 162-164 (1992), pp. 557–588.
29. M. S. Lam, E. E. Rothberg, M. E. Wolf, *The Cache Performance and Optimizations of Blocked Algorithms*, Computer Architecture News 21 (1993), pp. 63–74.
30. C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh, BLAS *for Fortran Usage*, ACM Trans. Math. Software 5 (1979), pp. 63–74.
31. V. Pan, *How Can We Speed Up Matrix Multiplication?*, SIAM Rev. 26 (1984), pp. 393–415.
32. R. Schreiber, J. J. Dongarra, *Automatic Blocking of Nested Loops*, Technical Report CS-90-108, University of Tennessee, 1990.
33. V. Strassen, *Gaussian Elimination Is not Optimal*, Numer. Math. 13 (1969), pp. 354–356.
34. C. W. Ueberhuber, *Numerical Computation*, Springer-Verlag, Heidelberg, 1997.
35. R. van de Geijn, *Using* PLAPACK*: Parallel Linear Algebra Package*, MIT Press, 1997.
36. R. C. Whaley, J. J. Dongarra, *Automatically Tuned Linear Algebra Software*, Technical report, LAPACK Working Note 131, 1997.
37. S. Winograd, *On Multiplication of* 2×2 *Matrices*, Linear Algebra Appl. 4 (1971), pp. 381–388.