A Block Algorithm for Orthogonalization in Elliptic Norms.

Conference Paper *in* Lecture Notes in Computer Science · January 1992

DOI: 10.1007/3-540-55895-0_434 · Source: DBLP

CITATIONS
4

READS
37

1 author:

Stephen Thomas
National Renewable Energy Laboratory
71 PUBLICATIONS **1,367** CITATIONS

SEE PROFILE

A Block Algorithm for Orthogonalization in Elliptic Norms

S. J. Thomas

Département d'informatique et de recherche opérationnelle
Université de Montréal.

Abstract

The need for block formulations of numerical algorithms in order to achieve high performance rates on pipelined vector supercomputers with hierarchical memory systems is discussed. Methods for analysing the performance of standard kernel linear algebra subroutines, such as the BLAS (levels 1, 2 and 3), are reviewed. We extend the L_2 block Gram–Schmidt algorithm of Jalby and Philippe to orthogonalization in *elliptic* norms induced by the inner product $u^T M u$, where $M = B^T B$ is a positive definite matrix. A performance analysis of this block algorithm is presented along with experimental results obtained on the Cray–2 supercomputer.

1. Introduction. A constraint on achieving fast instruction execution rates on the current generation of parallel vector supercomputers is limited memory bandwidth. It is not uncommon for each processor of such a machine to contain multiple floating point vector pipelines capable of operating in parallel and of producing one result per clock cycle. Moreover, arithmetic operations often can be overlapped by 'chaining' compound add/multiply units. In theory, this can result in impressive peak instruction execution rates. However, it is usually difficult to deliver arithmetic operands to the vector pipelines at a sufficient rate. To significantly increase memory transfer rates, current generation vector supercomputers employ a hierarchical memory architecture. Typically a high–speed cache (local memory, vector register files etc.) is placed between the very fast CPU registers and the large slow main memory of a machine with the result that the initial memory access time for a cache memory can be several times faster than for main memory. Sophisticated numerical algorithms exploit a hierarchical memory organization to achieve the highest possible memory transfer rates. Algorithms must not only employ vectorization and concurrency on such machines, but they must also possess a certain amount of data locality. The result is that the designer of numerical algorithms is faced with the additional task of managing the movement of data between different levels in the memory hierarchy. Data locality can be improved by employing 'block' variants of traditional numerical linear algebra algorithms. Typically, this is achieved by formulating an algorithm with respect to BLAS3 matrix–matrix operations such as matrix multiplication. Basic matrix–matrix primitives such as these are now available as standard kernel subroutines for parallel vector supercomputers. The basic linear algebra subroutines (BLAS levels 1, 2, and 3) are part of LAPACK, which is a package of numerical linear algebra codes for high–performance scientific computing [1]. Gallivan *et al* [2], [3] have analysed the performance of such block algorithms on machines containing high–speed cache memories.

The class of machines studied consists of a multiprocessor MIMD parallel machine, with n_p processors. In particular, all processors may access a small fast cache of size CS words and a large, but much slower, global main memory. It is assumed that any processor can read or write to any location in the cache or main memory. To analyse the performance of block algorithms on this type of architecture, the authors of [3] propose that the effects of data locality, concurrency and vectorization can be studied by applying a 'decoupling' strategy. The total computation time T of an algorithm is broken down into two components: the arithmetic time T_a and the data loading time T_l. If it is assumed

that the cache is infinitely large, then T_a represents the execution time of an algorithm when there is no memory hierarchy present. Thus, T_a varies according to the degree of vectorization and concurrency that is possible. One can account for the scheduling of arithmetic operations for SIMD execution across several pipelines or MIMD parallelism by letting $T_a = n_a \tau_a$, where n_a is the number of arithmetic operations (flops) and τ_a is the average time to complete a single operation. The data loading component T_l represents the additional time required to load data from the main memory into a cache of size CS. Given n_l the number of data transfers required and τ_l, the average load time, this component can be expressed as $T_l = n_l \tau_l$. Therefore, the total computation time can be given as the sum: $T = n_a \tau_a + n_l \tau_l$. However, it should be noted that the possible overlap of computation with data loading is not excluded by this model, and could result in a reduction of τ_l according to the cache prefetch policy employed. Despite the rather complicated factors which influence the parameters τ_a and τ_l, it is still possible to study the relationship between data locality and performance. The approach taken in [3] is to consider the 'relative' cost of data loading $T_l/T_a = \lambda\mu$, where the authors define a *cache-miss* ratio $\mu = n_l/n_a$ and a *cost* ratio $\lambda = \tau_l/\tau_a$. For machines that implement a specific cache prefetch policy, it is often possible to reduce λ so that the cache-miss ratio has less of an impact on performance. However, for most machines a lower bound on λ can be given and the problem reduces to minimizing the factor μ through algorithm design. If the time components are regarded as parameterized cost functions, then T_a and T_l can be minimized separately to optimize performance. A region in each parameter space is found that is close to the minimum of the cost function.

2. BLAS3 primitives. Since block formulations of algorithms employ BLAS3 primitives, our first task is to determine the performance characteristics of these matrix operations for a general architectural model. Thus, we apply the analysis techniques introduced in [3] to the most basic BLAS3 primitive; $C = C + AB$, where C, A and B are $n_1 \times n_3$, $n_1 \times n_2$, and $n_2 \times n_3$ matrices. If $n_2 = k \ll n_1, n_3$, then this operation is referred to as a rank-k update, but may also represent a dense matrix multiplication when A, B, and C are all large matrices. When $n_2 = 1$, for example, this primitive reduces to the BLAS2 rank-1 update. Both vectorization and concurrency are present in our model architecture, with data movement between two memory levels in the memory hierarchy. To optimize the performance of the above BLAS3 primitive on such a machine, consider a partitioning of the matrices C, A, and B into submatrices of dimension $m_1 \times m_3$, $m_1 \times m_2$ and $m_2 \times m_3$, where $n_1 = k_1 m_1$, $n_2 = k_2 m_2$, and $n_3 = k_3 m_3$. The 'individual' block operations $C_{ij} = C_{ij} + A_{ik} B_{kj}$ can be performed in parallel across n_p processors. Moreover, these operations can be pipelined for SIMD execution via multiple vector pipelines. Optimal block size selection for m_1, m_2 and m_3 can be determined by minimizing T_a, the arithmetic time component, and T_l, the data loading component associated with these operations. Consider the kernel algorithm for one such block operation:

```
do r = 1, m3
   do t = 1, m2
      do s = 1, m1
         c_sr ← c_sr + a_st b_tr
end
```

where we denote matrix elements by variables which are lowercase letters. The outermost loop iterations can be performed in parallel. The two inner loops represent an m_2-adic

operation. The latter can be implemented by a *multiply–add* vector instruction, which happens to be the best combination of operations for most vector supercomputers. In particular, fast execution rates for this operation can be achieved on register–to–register machines such as the CRI Cray–2. Values of m_1, m_2 and m_3 that minimize T_a are generally machine dependent. For register–to–register machines, m_1 should be a multiple of the vector register length (64 words for the Cray–2). The value of m_2 should be large enough to achieve a significant fraction of the peak speed of the multiply–add. Finally, m_3 should be a multiple of the number of processors, n_p (four for the Cray–2).

It is assumed, when executing a block matrix–multiply, that the submatrices A_{ik} are loaded from main memory into the cache only once for each execution of the j–loop. Furthermore, we assume that C_{ij} and B_{kj} are repeatedly loaded into the cache. If L denotes the total number of element loads from main memory into the cache, then it follows that: $L = n_1 n_2 + n_1 n_2 n_3 \, \rho(m_1, m_2)$, where $\rho(m_1, m_2) = m_1^{-1} + m_2^{-1}$. Thus, minimization of the load time is equivalent to minimizing $\rho(m_1, m_2)$ subject to the constraints imposed by a cache of fixed size CS. If it is assumed that a block A_{ik} remains in the cache then $m_1 m_2$ cache memory locations are required. When m_2 exceeds the number of vector registers available for the multiply–add computation, an additional $m_2 p$ cache elements will be needed in order to store columns of B_{kj}. Therefore, minimization of the load time T_l for the BLAS3 primitive is equivalent to the minimization of $\rho(m_1, m_2)$ subject to $m_2(m_1 + p) \leq CS$, $1 \leq m_1 \leq n_1$ and $1 \leq m_2 \leq n_2$, where the value of m_3 is arbitrary. The constraints described above represent a rectangle and a hyperbola in the (m_1, m_2) parameter plane. It is shown in [3] that a solution to this optimization problem lies in one of four different regions (solution regimes) of the bounding rectangle. Since the number of memory element loads into cache is given by $L \equiv n_l$ and since the number of arithmetic operations (flops) required for the BLAS3 primitive is $n_a = 2n_1 n_2 n_3$, the resulting cache–miss ratio $\mu = n_l / n_a$ is: $\mu = 1/(2m_1) + 1/(2m_2) + 1/(2n_3)$. We recall that T_a is minimized for large positive values of m_1 and m_2, thus insuring efficient use of vector pipelines. Except for cases where the matrix dimensions n_1, n_2 and n_3 are small, the data load time T_l is minimized for the positive values of m_1 and m_2 specified in the abovementioned regimes. Therefore, the decoupling strategy proposed by Gallivan and his co–workers is an effective means of determining optimal blocksize values, since m_1 and m_2 can be chosen to simultaneously minimize T_a and T_l.

3. Orthogonalization in elliptic norms. Let B be a *known* full–rank matrix of dimension $m \times p$, with $m \geq p$, and let M be the positive definite $p \times p$ matrix defined by $M = B^T B$. For a $p \times 1$ vector u, an *elliptic* norm is induced by the inner product $u^T M u$: $\|u\|_M = \|Bu\|_2 = (u^T B^T B u)^{1/2} = (u^T M u)^{1/2}$. The vector u can be considered as representing, in an oblique coordinate system, the vector whose representation is the $m \times 1$ vector Bu in an orthonormal system. It is for this reason that the matrix B is sometimes referred to as an oblique transformation. Now consider a $p \times n$ matrix S, with $p \geq n$, and suppose that we wish to orthogonalize the columns of S with respect to such an elliptic norm. (The M–orthogonalization of a set of vectors is a basic step in methods for the solution of large symmetric systems of linear equations by iterative techniques and for the generalized eigenvalue problem $Ax = \lambda M x$.) To formulate a modified Gram–Schmidt algorithm in an elliptic norm one can simply take inner products with respect to the matrix M in the traditional L_2 method. However, slightly more efficient variants of this algorithm are given in [6]. For example, if the Cholesky factor B is known instead of M, then an algorithm which avoids matrix–matrix multiplication to form M can be obtained

by replacing M with B^TB and separating the individual matrix–vector products.

Algorithm PGSM (Product Gram–Schmidt in the M–norm)

for $j = 1$ **to** n **do**
  (1) $v_j := Bs_j^{(j)}$
  (2) $\rho_{jj} := \|v_j\|_2$
  (3) $g_j := s_j^{(j)}\rho_{jj}^{-1}$
  (4) $q_j := v_j\rho_{jj}^{-1}$
  (5) $t_j := B^Tq_j$
  for $k = j+1$ **to** n **do**
    (6) $\rho_{jk} := t_j^Ts_k^{(j)}$
    (7) $s_k^{(j+1)} := s_k^{(j)} - \rho_{jk}g_j$
  end
end

The complexity of the above algorithm is $4mpn + 2pn^2$. We note that the same basic operations found in the L_2 modified Gram–Schmidt algorithm are employed. In particular, step (6) above is an inner product (sdot) and step (7) is the vector triad (saxpy). However, steps (1) and (5) are BLAS2 matrix–vector products (sgemv) and these two operations account for the additional $4mpn$ flops. To generalize the results of Jalby and Philippe [5] to orthogonalization in elliptic norms, we remark that at step (7) of PGSM,

$$s_k^{(j+1)} = s_k^{(j)} - g_jg_j^TMs_k^{(j)} = (I - g_jg_j^TM)s_k^{(j)},$$

and thus the elementary operators $I - q_jq_j^T$ of the traditional Gram–Schmidt algorithm are simply transformed to $I - g_jg_j^TM$ when an elliptic norm is employed. Now suppose that the original $p \times n$ matrix S is partitioned into d blocks: $S = [S_1, S_2, \ldots, S_d]$, where a block S_j is of size n_j, $n_1 + n_2 + \cdots + n_d = n$. In order to generalize the algorithm to block form, we can define the block–operator:

$$\begin{aligned} I - G_jG_j^TM &= (I - g_{n_{j-1}+n_j}g_{n_{j-1}+n_j}^TM)\cdots(I - g_{n_{j-1}+1}g_{n_{j-1}+1}^TM) \\ &= (I - G_jT_j^T) \end{aligned}$$

where $T_j = MG_j$. Consequently, elliptic algorithms can be generalized to block form by programming the latter expressions as follows.

Algorithm BGSM (Block Gram–Schmidt in the M–norm)

for $j = 1$ **to** d **do**
  (1) $(G_j, T_j, R_{jj}) := PGSM(S_j^{(j)})$
  for $k = j+1$ **to** d **do**
    (2) $R_{jk} := T_j^TS_k^{(j)}$
    (3) $S_k^{(j+1)} := S_k^{(j)} - G_jR_{jk}$
  end
end

4. Performance Analysis of BGSM. Given the matrices $B(m \times p)$ and $S(p \times n)$, where $m \geq p \geq n$, we recall the performance analysis of the L_2 block algorithm BGS given in [3], [5]. It can be shown that, from the standpoint of block operations, Algorithms BGS and BGSM are basically identical. In fact, if we adopt a block partitioning with $n = d\omega$, then we see that Algorithm BGSM employs the matrix primitives:

1. Algorithm PGSM applied to the $p \times \omega$ block $S_j^{(j)}$.
2. A matrix multiplication of the form $C = AB$, A is $\omega \times p$ and B is $p \times (n - j\omega)$. $(R_{jk} = T_j^T S_k^{(j)},\ k = j+1, \ldots, d.)$
3. $C = C - AB$, C is $p \times \omega$, A is $p \times \omega$ and B is $\omega \times (n - j\omega)$. $(S_k^{(j+1)} := S_k^{(j)} - G_j R_{jk},\ k = j+1, \ldots, d.)$

It is possible to derive a load function $L(\omega) \equiv n_l$ for Algorithm BGSM by employing the analysis techniques described in sections 3 and 4. Despite the fact that $L(\omega)$ can be algebraically complex it is still possible to characterize the performance of the block algorithm by approximations to the cache–miss ratio over two separate intervals. An approximation to the cache–miss ratio valid over the interval $1 \leq \omega \leq CS/p$ is $\mu \approx 1/(2\omega) + \eta_1$, where η_1 is proportional to $1/n$. The minimum of this function occurs at $l = CS/p$, when $\mu = p/(2CS)$. For $CS/p \leq \omega \leq n$ it can be shown that the approximation,

$$\mu \approx \frac{1}{2\omega}\left(1 - \frac{\gamma}{n}\right) + \frac{\omega}{2}\left(\frac{1}{n} + \frac{1}{CS}\right) + \eta_2$$

where $\gamma = l^3 + 3l - 3$ and η_2 is proportional to $1/n$ holds. The above function is a hyperbola, which reaches a minimum at $\omega \approx \sqrt{CS}$ and increases thereafter from $\sqrt{CS}$ to n. If $\omega = n$, then $\mu \approx 1/2$ corresponding to the BLAS2 algorithm. Since algorithm BGS and BGSM employ the same operations, except for the additional (sgemv) operations and a normalization, we expect that the above cache–miss ratio approximations should be applicable.

We note, however, that the performance of the algorithm is characterized only partially by these approximate functions. If we assume that $m = p$, then the complexity of Algorithm BGSM is $4m^2n + 2mn^2$. Moreover, when $m \gg n$ the first term dominates. Thus, we propose that the performance of the block algorithm should more closely follow the performance of the (block) matrix–multiplies: $V_j = BS_j$, $T_j = B^T Q_j$, $j = 1, \ldots d$, when $\omega \approx CS/m$. The cache–miss ratio for the these block matrix multiplies is given by: $\mu_{min} = 1/\sqrt{CS} + n_p/(2CS) + 1/(2n_3)$ and when $n \gg \sqrt{CS}$ it follows that $\mu \approx 1/\sqrt{CS}$.

5. Experimental Results. Performance tests were carried out on the National Center for Supercomputer Applications (NCSA) Cray–2 supercomputer at the University of Illinois at Urbana–Champaign. Although the Cray–2 is a four processor vector supercomputer, only one processor was employed in our benchmarks. Each processor has a clock cycle time of 4.1ns, resulting in a theoretical peak rate of 243.9 Mflops for single floating point operations and 487.8 Mflops for compound add/multiply operations. The overall design of the Cray–2 is based upon a register–to–register architecture. Each processor has a $16k$ element, 64–bit word local memory. This local memory replaces the 64 element, 64–bit T (backup scalar) and 64 element, 24–bit Cray X–MP or 32–bit Cray Y–MP B (backup address) registers. There is one port to local memory for each processor, just as there is one port to common memory. The latency to local memory is about 4 clock

periods compared to about 20 clock periods for common memory. After this startup time, subsequent vector elements can move at one word per clock period for either memory.

For our tests of the BGS algorithm we employed a 1024×512 randomly generated matrix A. The performance of the algorithm is given in Figure 1, containing plots of Mflops per second versus blocksize ω. The peak speed of the Cray–2 and the Mflops rate of the BLAS2 MGS algorithm are represented by straight lines. In the case of Algorithm BGSM we generated a 1024×1024 matrix B and a 1024×512 matrix S. This results in $4m^2n = 2.15$ Gflops for the matrix multiply component and $2mn^2 = 0.536$ Gflops for the Gram–Schmidt component of the algorithm. Clearly, we expect the former to dominate the computation. Performance results are presented in Figure 1. For comparison purposes we have plotted the performance curve of the associated block matrix multiply in Figure 1. The results indicate that BGSM exhibits the composite behaviour of BGS and the matrix multiply. Finally, we ran a benchmark of the LAPACK block Householder algorithm on a 1024×1024 matrix.

Acknowledgements. The author would like to thank Prof. Ray Zahar for many helpful suggestions which greatly improved the manuscript. Dr. John Larson of the Center for Supercomputing Research and Development (CSRD) of the University of Illinois at Urbana–Champaign generously provided computing time on the NCSA Cray–2. Bill Harrod and Ed Anderson of Cray Research Inc. provided the author with pre–release versions of LAPACK routines and instructions on their use. Finally, it was very kind of Professors W. Jalby and B. Philippe to make their BGS code available on the NCSA Cray–2 and Cray Y–MP for benchmarking.

References

[1] K. GALLIVAN, W. JALBY, and U. MEIER, "The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory", *SIAM J. Sci. Stat. Comput.*, **8** (1987), pp. 1079–84.

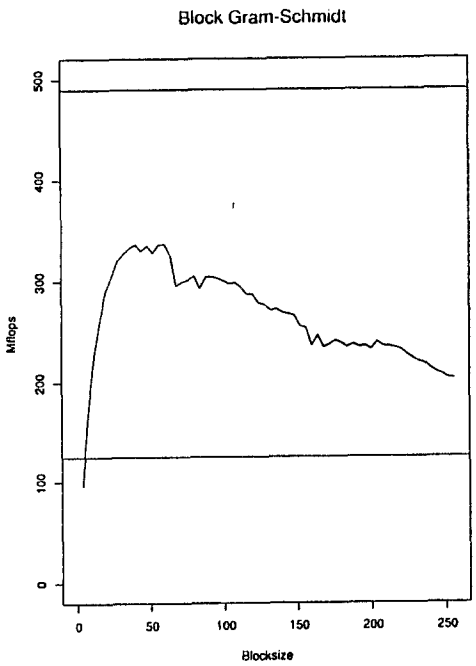
[2] K. GALLIVAN, R. J. PLEMMONS, and A. SAMEH, "Parallel algorithms for dense linear algebra computations", *SIAM Review*, **32** (1990), pp. 54–135.

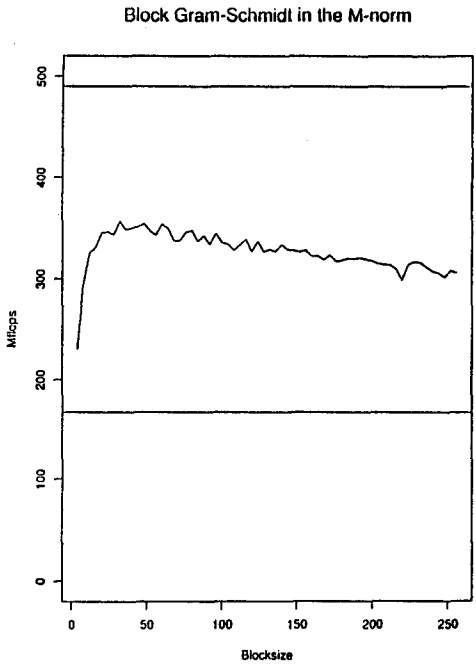
[3] K. GALLIVAN, W. JALBY, U. MEIER, and A. SAMEH, "The impact of hierarchical memory systems on linear algebra algorithm design", *Intl. J. Supercomputer Appl.*, **2** (1987), pp. 1079–1084.

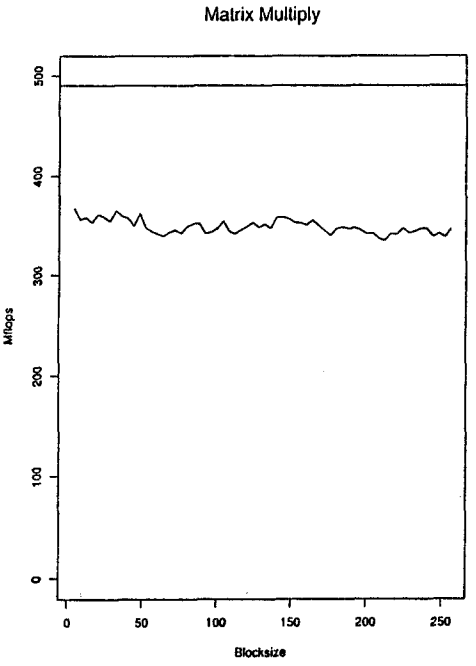
[4] W. JALBY and U. MEIER, "Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system", *ICPP Proceedings*, pp. 429–32, 1986.

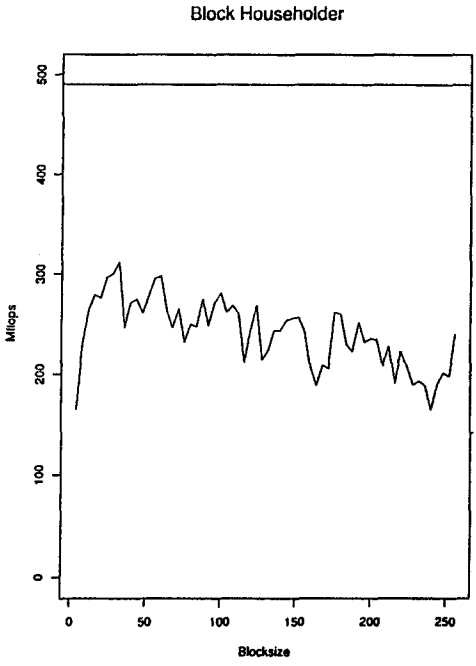
[5] W. JALBY and B. PHILIPPE, "Stability analysis and improvement of the block Gram–Schmidt algorithm", *SIAM J. Sci. Stat. Comput.*, **6** (1991), pp. 26–50.

[6] S. J. THOMAS and R. V. M. ZAHAR, "Efficient Orthogonalization in the M–norm", *Congressus Numerantium*, vol. 80, 1990.


Block Gram-Schmidt
Mflops
0
100
200
300
400
500
0
50
100
150
200
250
Blocksize


Block Gram-Schmidt in the M-norm
Mflops
0
100
200
300
400
500
0
50
100
150
200
250
Blocksize


Matrix Multiply
Mflops
0
100
200
300
400
500
0
50
100
150
200
250
Blocksize


Block Householder
Mflops
0
100
200
300
400
500
0
50
100
150
200
250
Blocksize