Gary W. Howell

BLAS-3 Sparse U B V Decomposition

512 Farmington Woods Dr
Cary
NC 27511

gary$_h$owell@ncsu.edu

BLAS-3 Sparse $UB_{K+1}V$ Factorization

A matrix A can be reduced to upper triangular banded form by BLAS-3 block Housholder transformations. Deferring matrix updates, the algorithm accesses A only to extract blocks and to perform multiplications AX, AY^T, where if X and Y have K columns than the bandwidth of upper triangular B_{K+1} is $K+1$. When A is sparse, block Householder eliminations provide a BLAS-3 method to contruct a $UB_{K+1}V$ approximation, with U and V orthogonal, U $(m \times l)$, V $(n \times l)$, $B_{K+!}$ $l \times l$, with the size of l constrained by available RAM.

The decomposition is stable, is efficient in terms of cache utilization, and should scale well in distributed parallel computation.

The $UB_{K+1}V$ approximate (truncated) decomposition can be used to provide some matrix singular values, to solve linear systems and least squares problems, and to provide an approximate inverse preconditioner. Multi-grid applications may be parallel iterations with the full matrix, as a preconditioner, or in solution of a coarsened problem. Each of these applications is discussed.

A primary advantage of the block reduction to a banded upper triangular form is that the underlying sparse matrix is accessed only for multiplications by blocks of matrices (sparse matrix dense matrix multiplications). Serial SPMD multiplications run at a significant fraction of peak speed on cache based processors, and should also run well in parallel. Stability of block Householder transformations aids in scalability.

Numerically, the truncated $UB_{K+1}V$ decomposition is seen to be particularly efficient in approximating low rank matrices or low rank matrices added to a matrix with a known factorization.

Some unresolved questions are how to best prepermute A, how to best pad A (thick start), and considered as a Krylov method the best restart strategies (thick restart?, repermutation of A?).

The remainder of the abstract discusses why multiplication of AX is likely to be faster than computing Ax, assuming A sparse, x a dense vector X a dense matrix

with relatively few columns. Assume A is too large to fit in cache memory. A is typically stored so that it can be pulled in a stream from RAM. If x fits in cache, then as each element of A appears in the CPU it can be matched by the appropriate element of x. If indexing operations do not take too long the main cost is then the fetch of A from RAM. Since the fetch of A is streamed, elements of A arrive more or less at the peak speed of the data bus.

On Intel Xeons, sparse Ax can attain up to about ten per cent of the advertised peak flop rate. When x becomes too large to fit in cache, and if x is accessed randomly, then the computation is dominated by cache misses and slows dramatically. In some experiments with Intel Xeons, Ax compted at less than one per cent of peak processor speed. When multiplying by X with K columns as opposed to x, each access of an element of A allows K multiplications. Storage of X should be arranged so that when a given element of A is accessed, the next required elements of X are accessed. In Fortran, X for AX is stored as X^T so that each row of X is sequentially stored as a column of X^T.

In experiments summarized here, we also blocked A (column blocks) to further minimize cache misses. The blocked SPMD AX can execute at a flop rate several orders of magnitude faster than non-blocked Ax. The marked speedup of BLAS 3 over BLAS 2 motivates the algorithm development described in the presentation.