High Performance Bidiagonalization by BLAS 2.5 Operators

Gary Howell, Charles Fulton
Karen Marmol
James Demmel, UC Berkeley

Mathematical Sciences
Florida Institute of Technology
Melbourne, FL 32901
e-mail howell@zach.fit.edu

Numerical Linear Algebra and Scientific Computation

The main driving force for speedier computations is of course video games. A good PC is competitive with the fastest serial machine curently made.

Faster computing is available by linking PC-s together. Florida Tech is in the process of acquiring a Beowulf processor consisting of 48 processors (essentially PCS) linked together with a doubled up 100 MBit switching network and a RAID common storage.

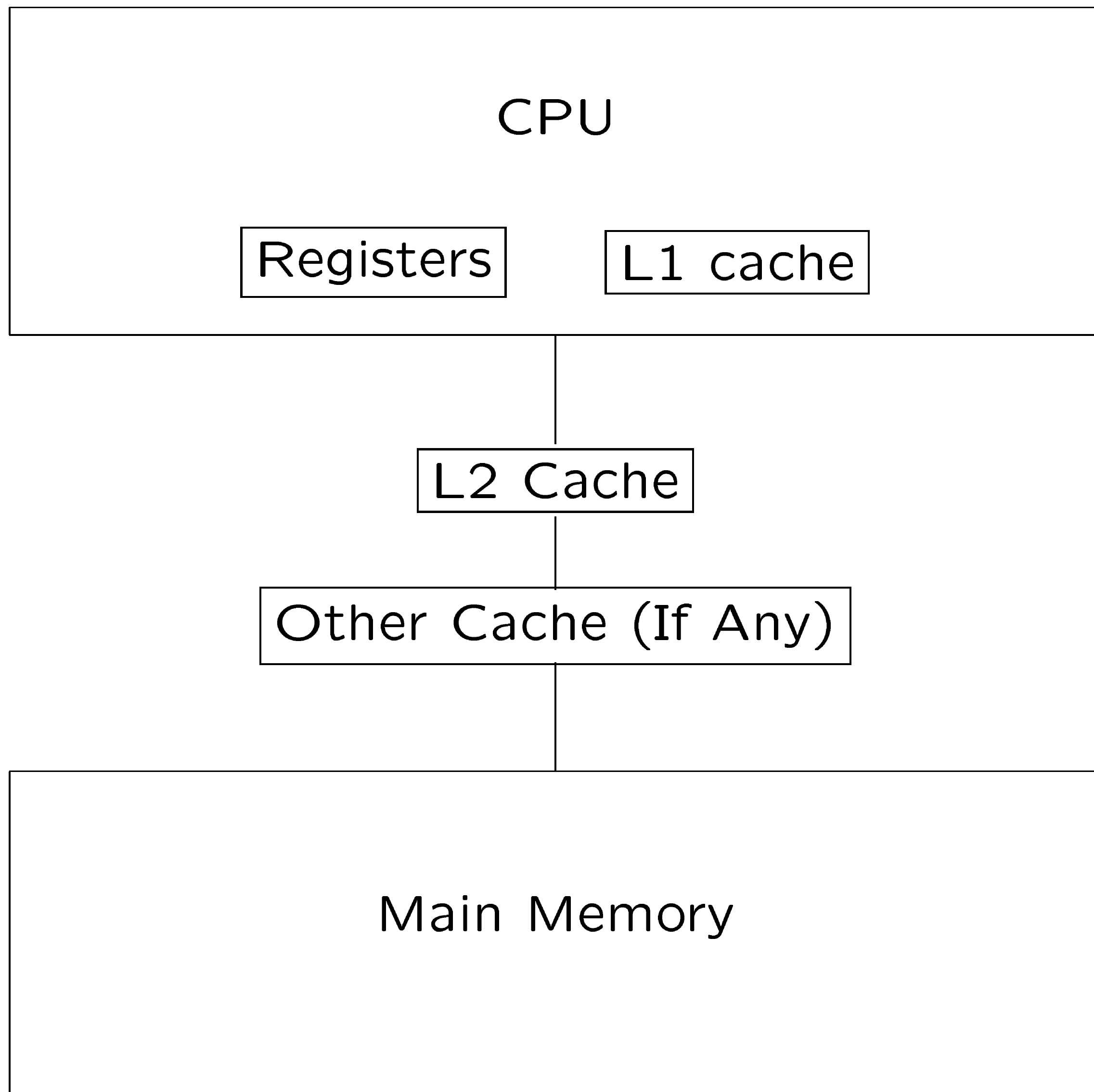
Programming for high performance is a bit tricky, both in serial and parallel. It is easy to

- Program serial machines so they run at 1 % of potential performance.

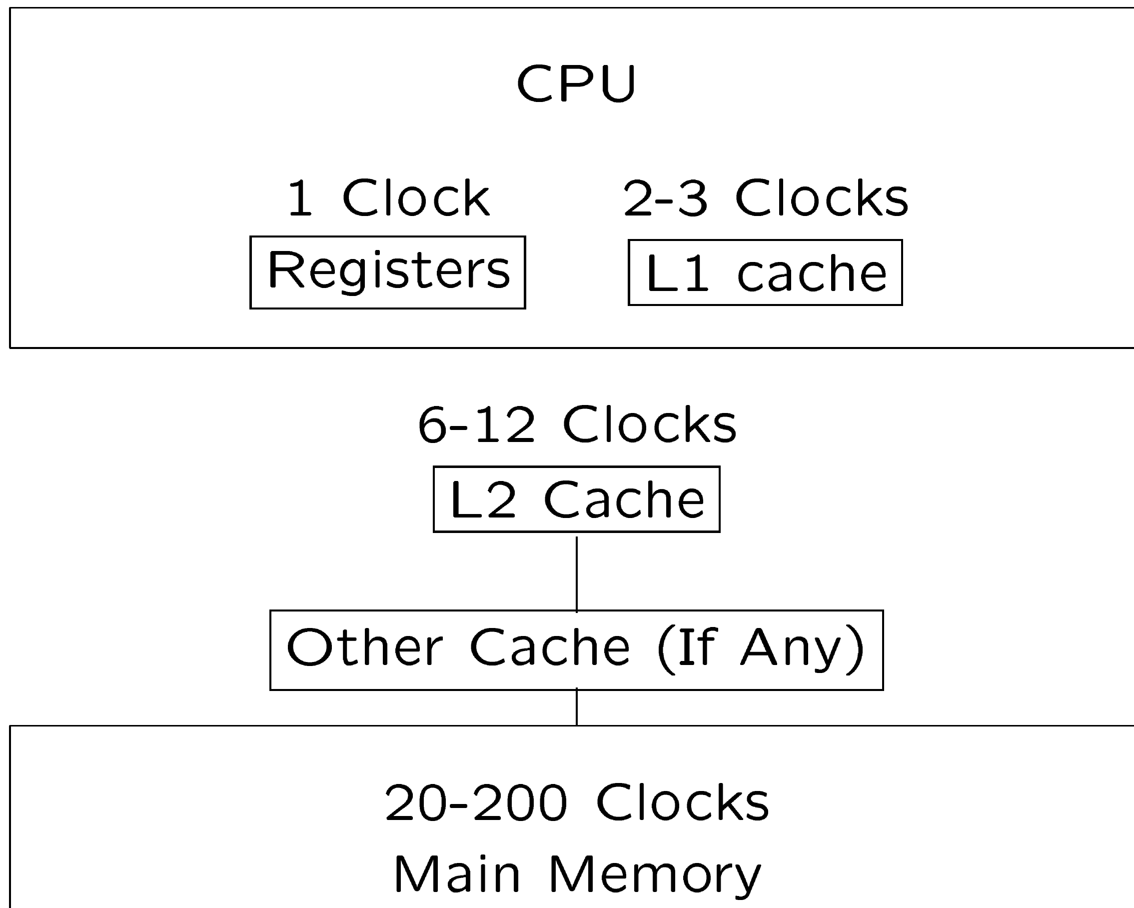
- Make the program run even slower in parallel.

Designing programs to run efficiently is harder and takes a good deal of thought.

Memory Hierarchy



Clock Ticks for Memory Access



$O(10^6)$ Clocks to Hard Drive

Accessing data on other processor RAM is slightly less expensive than accessing data on a processor's own hard drive.

Attaining speedup in parallel computation requires modeling algorithms in terms of computational and communication expense.

For example we can take parallel time T_p for p processors as

$$T_p = \frac{T_{serial}}{p} + \sum_i T_{communication_i}$$

where for a given communication

$$T_{communication} = T_{start} + (T_{perbyte}) * bytes$$

(Fulton, Shoaff, Howell, 1990) Similarly, effcient algorithm execution on one processor depends on data transfer. Typically, counting data transfer is more important than counting floating point operations.

Blocking for QR Decomposition

$$A = \left[\begin{array}{c|c|c|c|c|c} A_1 & A_2 & \ldots & A_i & \ldots & A_k \end{array} \right]$$

The first block of b columns is reduced to upper triangular by a block Householder transformation

$$A_1 = (I - U_1 S_1 U_1^T) A_1$$

where computing U_1, R_1, S_1 requires only A_1.

Denote the trailing blocks as A^1 and perform

$$A^1 = A^1 - U_1 [S_1 (U_1^T A^!)] \qquad (1)$$

Similarly, we reduce A_2 to upper triangular and form U_2, R_2, S_2 to update trailing blocks, etc.

Moral: For QR Decomposition MFLOP = O(CPU clock speed)

For the matrix matrix multiply _GEMM (BLAS 3) there are many floating point operations for each element of A fetched from and stored to main memory. So the limiting factor is CPU speed as oppposed to transfer of data to the CPU.

Careful implementations of QR decomposition run almost at CPU clock speed. For example, use LAPACK or NAG or IMSL and get the assembly language BLAS-3 from the manufacturer of your CPU.

Many (most) computations are constrained by rate of data transfer in the memory hierarchy.

Algorithms Where MFlop rate = CPU Rate

1. Dense matrix matrix multiply (BLAS-3 _GEMM)

2. QR decomposition (almost all flops are _GEMM)

3. LU decomposition (almost all flops are _GEMM)

4. Eigenvalues by matrix sign function. By using 100 n^3 flops almost all flops are _GEMM. In comparison 20 n^3 for ToHESS then QR or 8/3 n^3 for bounded multiplier BHESS-BR but most flops are not BLAS-3.

5. Claim: the average execution rate at supercomputing sites is 1% of peak performance.

BLAS-2 Matrix Vector Multiply Runs Slower Than BLAS-3

Consider Ax. For each element of A we perform one multiply and one add. If A is not resident in cache (e.g., A square of size 300 is too large for cache) we must also transfer A from main memory (or worse from the hard drive).

Most CPU time is spent waiting for A to arrive. The computation is constrained by

1. Latency of 20-200 CPU clock cycles.

2. **Throughput of the data bus.**

Naive algorithms often run at 1% of peak rate. Loop-unrolling, accessing the matrix in the right column or row order, using compiler optimizers etc. can get MFlop rate to about 20-30% of peak rate.

For example, in our course this semester, matrix vector multiplies in week one ran at 3 M-flops.

Since then we've had a race, who could get seedup faster, students who use parallel computing on networked SGU workstations, versus those who try to make the computation more efficient on a single workstation.

Currently, (on 200 MHz SGIs)

- The serial version runs at 120 Mflops (the inner loop has eight equations with 16 flops each) Speedup of 40.

- The best parallel version runs at 30 Mflops (10 Mflops on each processor, but slowed by overhead. Speedup of 10.

Of course, a good matrix vector product will finally use the best serial code. A primary virtue of parallel computation is for problems too huge for serial computation. For problems with too much data for a single processor to hold in RAM, accessing data from many processors is more efficient than storing all the data on one hard drive.

Half the flops in reduction to small-band are BLAS-2.

Column by column reductions to Hessenberg, tridiagonal, or bidiagonal form involve two matrix vector multiplies per column elimination.

For orthogonal reduction to Hessenberg, the trailing matrix must be transferred from main memory to cache twice for each column elimination. Example, Pentium 133 (MHz CPU) with a 33 MHz 32 bit bus (32 bits per bus clock cycle) transfers at most 16.5 million double precision numbers per second.

	MFlops	Flops/read
Matrix Vector Mult	17.8	2
Orthogonal Reduction to Hessenberg	23.0	4
To Hess W/O Tuned BLAS	7.8	4

Naive Householder Bidiagonalization

$$A = UBV$$

is a URV decomposition that is also the $O(n^3)$ flop portion of determining singular values.

A column-row elimination with $\|u\| = \sqrt{2} = \|v\|$ is accomplished by

$$A \leftarrow (I - uu^T)A(I - vv^T)$$

naturally corresponding to the four BLAS-2 operations

$$\begin{aligned} z^T &\leftarrow u^T A, && \text{_GEMV} \\ w &\leftarrow Av, && \text{_GEMV} \\ A &\leftarrow A - uz^T, && \text{_GER} \\ A &\leftarrow A - wv^T, && \text{_GER} \end{aligned}$$

where _GER operations are rank one updates. But this requires four reads and two writes of A for eliminating one column-row pair.

Data Transfer of Bidiagonalization Algorithms

The current LAPACK algorithm defers updates (performed as a BLAS-3 matrix matrix multiply) but requires the two _GEMV calls.

Table Compares Floating Point Reads and Writes of the Trailing Matrix in One Row Column Elimination

BLAS level	2	2 & 3	2.5	2.5 & 3
READS WRITES	4 2	$2+1/b$ $1/b$	1 1	$1+1/b$ $1/b$
ALGORITHM	Naive	LAPACK	I	III

BLAS 2.5 operations reduce data transfer by combining several BLAS-2 calls into one operation.

BLAS 2.5 _GEMVER

Reductions to similar Hessenberg, banded Hessenberg, tridiagonal, or bidiagonal forms are the predominant execution time in determining principal or eigenvalues. The two new BLAS 2.5 _GEMVER and _GEMVT operators make the combined operation Ax and $y^T A$ more efficient by eliminating a read from each column row elimination.

_GEMVER performs

$$\begin{aligned} \widehat{A} &\leftarrow A + \alpha u_1 v_1^T + \beta u_2 v_2^T \\ \widehat{y}^T &\leftarrow y^T \widehat{A} \\ x &\leftarrow \widehat{A}\widehat{y} + z \end{aligned}$$

Reduction to small band or bidiagonal form by $_GEMVER$ does not require a BLAS-3 update.

BLAS 2 _**GEMV** performs

$$y \leftarrow Ax + z$$

BLAS 2.5 _**GEMVT** performs

$$\widehat{y}^T \leftarrow y^T A$$
$$\widehat{x} \leftarrow A\widehat{y} + z$$

_GEMVT has no update.

MFlops for Matrix Vector Multiplies

	BLAS 2	BLAS 2.5
	_GEMV	_GEMVT
Pentium 133	17.8	23.4
200Mhz SPARC	26.2	49.3
	Ax	$y^T A$ & Ax

On the SPARC, _GEMVT performs one extra matrix vector multiply in almost no additional time.

3 New Bidiagonalization Algorithms

In associative arithmetic these would each result in the same bidiagonal matrix.

Alg I. Put almost all flops in the BLAS 2.5 operator _GEMVER. Efficient when data bus allows parallel reads and writes. Also when a matrix is so large that only a very few columns can fit in fast memory.

Alg II . A never update algorithm. All mat-vec-mults with the original sparse matrix. Desirable for low rank and/or sparse least squares.

Alg III. BLAS 2.5 _GEMVT with BLAS-3 matrix updates. Efficient when writes to main memory are more expensive than reads.

Tricks to Implement Bidiagonalization Using BLAS 2.5 Operators

A. Compute $u^T A$ and $A(A^T u)$ in one pass of data through cache.

B. Perform $(I - uu^T)A(I - vv^T)$ in one pass of data through cache.

C. Multiply by a matrix not yet updated.

D. Multiply by a Householder vector which eliminates a row (where the row is being computed in the same pass of the matrix through cache memory).

Trick A. Perform $u^T A$ and $A(A^T u)$ in one access of data

Let

$$A = \left(\, A_1 \,|\, A_2 \,|\, \ldots \,| \qquad |\, A_k \, \right)$$

If we perform

For $i = 1 : k$,

$$v_i^T = u^T A_i \,,$$
$$w = w + A_i v_i \,,$$

End

then only one column block of A is accessed at a time.

The second statement in the loop computes $Av = \sum_i A_i v_i$. This implementation allows the new BLAS operator GEMVT to be built on top of calls to the current BLAS operator GEMV.

Trick B. Performing $(I - uu^T)A(I - vv^T)$

Break up the computation into

$$\widehat{A} \leftarrow A - u_{old}z_{old}^T - w_{old}v_{old}^T$$

From knowing the updated matrix we can compute

$$u_{new}^T \widehat{A}$$

and

$$\widehat{A}v_{new}.$$

As in the last slide this can be performed in one loop through the matrix. In this case each succesive column of A is involved in three _AXPYs and a _DOT .

Trick B allows implementation of Algorithm I by _GEMVER.

Trick C. Multiplying by a Matrix as Yet Not Updated

If

$$\hat{A} = A - \sum_{i}^{k} w_i v_i^T$$

then

$$\begin{aligned} u^T \hat{A} &= u^T A - \sum_i (u^T w_i) v_i^T \\ &= u^T A - (u^T W) V^T \end{aligned}$$

becomes three matrix vector products.

Inexpensive so long as k is small.

Trick C is essential in Algorithms II and III.

Trick D. Multiplying by a Householder vector before we have it

The row to be eliminated is given as $\hat{y} \to y + u^T A$. We have seen we can perform $u^T A$ and $A\hat{y}$ in one call to _GEMVT. But actually we want to perform Av where

$$v = \frac{\hat{y} + \alpha e_1}{\kappa}.$$

Thus

$$Av = \frac{1}{\kappa}(A\hat{y} + \alpha A e_1)$$

with Ae_1 is simply a column of A.

So $A\hat{y}$ is merely a BLAS-1 _AXPY away from the desired product.

Current Status

So far

- We have Matlab script files implementing Algorithms I,II, and III.

- We have FORTRAN 77 implementations of Algorithms I, II, and III.

- _GEMVER and _GEMVT have been included in the new BLAS standards.

- There are FORTRAN 77 reference versions of _GEMVER and _GEMVT.

Preliminary optimization of _GEMVT is successful as the comparison of LAPACK and Algorithm III show.

Times for Bidiagonalization

MatrixSize	Time	Time
	Alg. III	LAPACK
250	.36	.5
300	.73	.90
350	1.14	1.52
400	1.75	2.23
450	2.43	3.22
500	3.33	4.41
550	4.44	5.94

Runs were on a Pentium II 266 MHz running under Linux using a tuned BLAS. Average rates of execution were 230 MHz for matrix matrix multiplies, 115 Mhz for Alg. III, 92 MHz for GEMVT, 80 MHz for LAPACK, 65 MHz for GEMV.

Comparison with Two-Stage Bidiagonalization

Two stage bidiagonalization (B. Grober and B. Lang, March 1998) requires $4mn^2 - 4/3n^3$ flops for a reduction to small band form, then a further $8n^2b$ flops for a reduction from bandwidth b to bidiagonal.

- The first stage is almost entirely BLAS-3 and therefore fast. For example, on a Pentium II 267 MHz it might run at 232 MHz, hence require 1.44 seconds for a 500 by 500 matrix, compared to 2.00 seconds for BLAS 2.5-BLAS 3.

- The second stage is in comparison negligible for large enough matrices. For $b = 20, n = 500$, this is about 12% of the flops. These are extra flops and tend to run slowly. If they run at 20Mflops, the total two stage time will be 3.44 seconds.

- For the case that the singular vectors are desired, two stage methods almost double the total number of flops compared to LAPACK or BLAS 2.5-BLAS 3 algorithms.

Conclusions

- BLAS 2.5 can halve the time to find
  - singular values (bidiagonalization)
  - eigenvalues of symmetric matrices (tridiagonalization)
  - eigenvalues of unsymmetric matrices (reduction to small-band form)

  In each case the reduced form allows determination of eigenvalues or singular values in $O(n^2)$ additional flops.

- The new bidiagonalization algorithms give a useful URV decomposition for sparse least squares problems and low rank least squares problems

e-mail **howell@zach.fit.edu**

Ongoing Projects

- Optimize BLAS 2.5
- Refine Code (reduce storage requirements, adapt for sparse case).
- Timings and Test Suite.
- Algorithm II for a numerically stable sparse least square algorithm
- Unsymmetric eigenvalue problem. Sparse and dense.

Latent Semantic Indexing

A term document matrix A for which the ij entry represents the frequency of term i in document j is a sparse matrix frequently used for document retrieval. One way to compress storage is to approximate

$$A = \sum_{i}^{n} \sigma_i u_i v_i^T$$

by its best rank k approximation

$$A_k = \sum_{i}^{k} \sigma_i u_i v_i^T$$

Conventional wisdom has been that stable Householder methods of producing A_k necessairly fill in the sparse matrix A, exploding storage. (Golub and Kahan 1965 in which both the Householder and Lanczos methods were first described, Berry, 1991, Golub and Van Loan 1995).

Deferring updates allows Householder transformations to be used without fill. As long as A_k requires less storage than sparse A, the (stable) Householder bidiagonalization method turns out to competitive with unstable Lanczos methods

- Storage. In either Lanczos or Householder methods, storage requirements are mainly for the original matrix A and the basis vectors stored in U and V. The Householder method requires keeping W and Z also.

- Floating Point Operations. The main floating point operation in either case are the same matrix vector multiplications.

- Communication and Potental Parallelism.

Blocking for Bidiagonalization

$$
A = \left[\begin{array}{c|c|c} B_k & \multicolumn{2}{c}{0} \\ \hline 0 & \hat{u} & \begin{array}{c} \hat{v}^T \\ A_{n-k-1} \end{array} \end{array}\right]
$$

$$
= \left[\begin{array}{c|c} B_k & 0 \\ \hline & \begin{array}{c} V^T \\ U \;\; | \; A_{n-k-b} \end{array} \end{array}\right]
$$

Block bordered algorithms produce U and V, deferring BLAS-3 updates to be performed on A_{n-k-b}.

Related work available by anonymous ftp at

cs.fit.edu

cd pub/howell

- BHESS (in revision for TOMS). *
- Bidiagonalization Algorithms (45 pages).
- Reports to BLAST committee
- Fortran and Matlab codes
- The BR Eigenvalue Algorithm (appeared in SIMAX in July 1999)

*howell@zach.fit.edu