

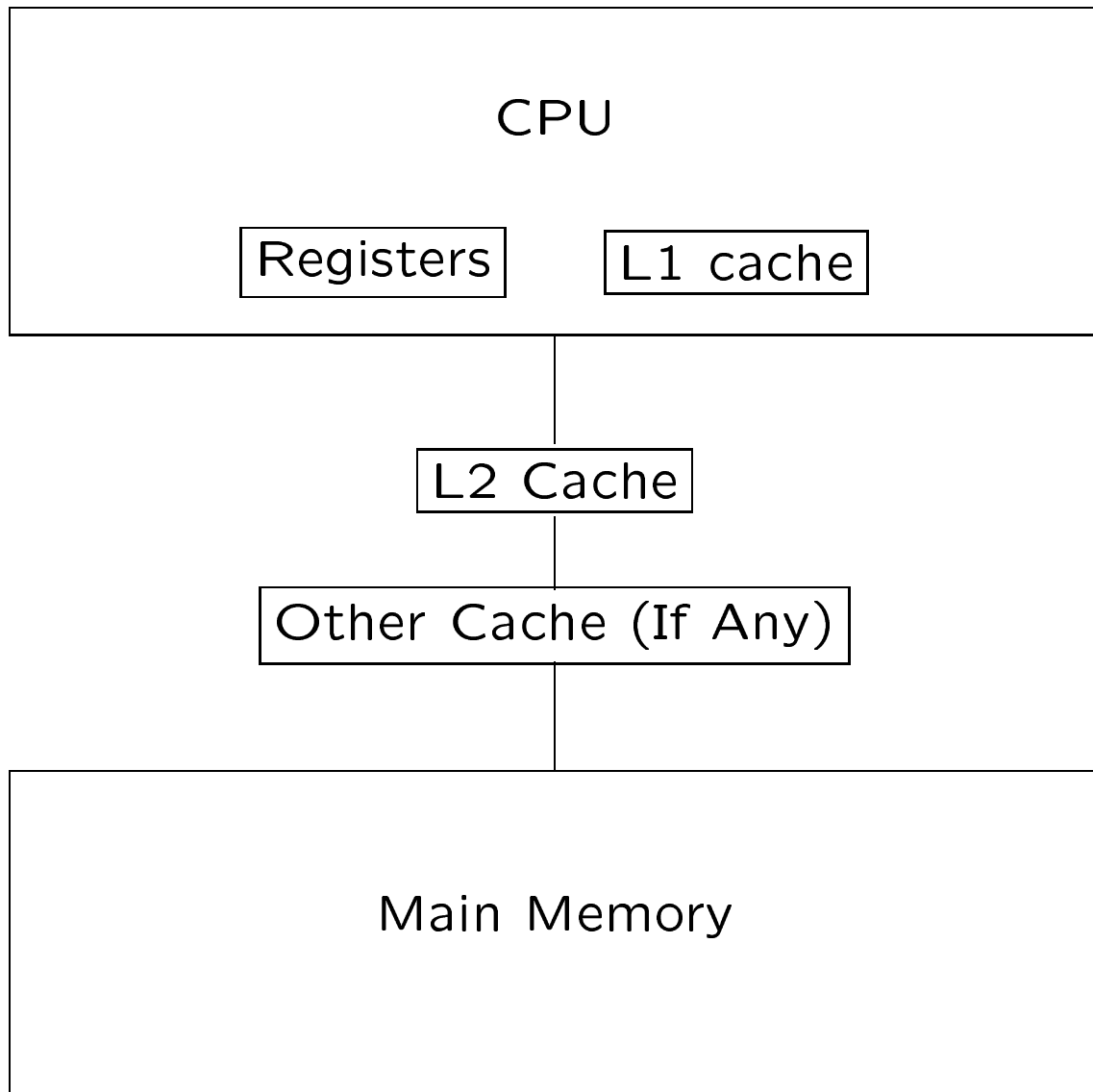
# **Cache Efficient Bidiagonalization by BLAS 2.5 Operators**

**Gary Howell, Charles Fulton  
Karen Marmol**

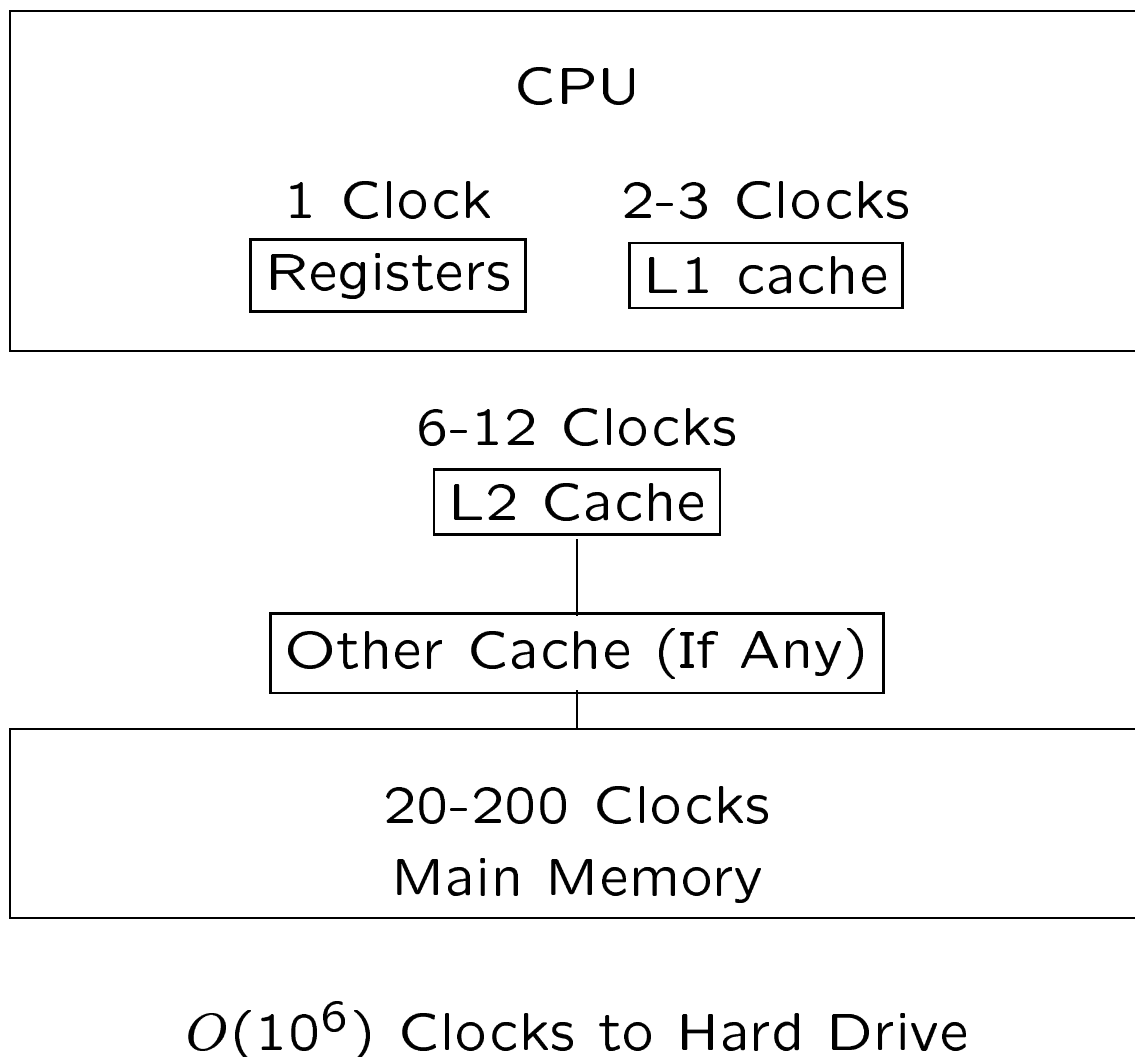
Mathematical Sciences  
Florida Institute of Technology  
Melbourne, FL 32901  
e-mail [howell@zach.fit.edu](mailto:howell@zach.fit.edu)

Thanks to James Demmel, UC Berkeley

# Memory Hierarchy



# Clock Ticks for Memory Access



We will assume that algorithm execution time is proportional to amount of data transferred from cache to main memory.

# 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  $20 n^3$  flops for ToHESS then QR or  $8/3 n^3$  flops for bounded multiplier BHES-  
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 10% - 20% of peak rate.

## 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<br>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

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

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

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

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

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

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

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

\_GEMVT has no update.

### MFlops for Matrix Vector Multiplies

|              | BLAS 2        | BLAS 2.5                 |
|--------------|---------------|--------------------------|
|              | _GEMV<br>$Ax$ | _GEMVT<br>$y^T A$ & $Ax$ |
| Pentium 133  | 17.8          | 23.4                     |
| 200Mhz SPARC | 26.2          | 49.3                     |
| Pentium 266  | 50            | 100 – 120                |

## 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

$$A = \left( A_1 \mid A_2 \mid \dots \mid A_n \right)$$

If we perform

For  $i = 1 : n$ ,

$$v_i = u^T A_i,$$

$$w = w + A_i v_i,$$

End

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

## Trick B. Performing

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

Break up the computation into

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

From knowing the updated matrix we can compute

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

and

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

As in the last slide this can be performed in one loop through the matrix. In this case each successive column block of  $A$  is involved in a rank two update and two `_GEMVs` .

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  $\hat{y} \rightarrow y + u^T A$ . 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

$$Av = \frac{1}{\kappa}(A\hat{y} + \alpha Ae_1)$$

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`.

Times in seconds for LAPACK vs. Algorithm III (BLAS 2.5-3.0) are shown below.

### **Times for Bidiagonalization**

| MatrixSize | Time Alg. III | Time LAPACK |
|------------|---------------|-------------|
| 250        | .26           | .5          |
| 300        | .45           | .90         |
| 350        | .74           | 1.52        |
| 400        | 1.05          | 2.23        |
| 450        | 1.52          | 3.22        |
| 500        | 2.00          | 4.41        |
| 550        | 2.68          | 5.94        |

Runs were on a 266 MHz Pentium II running under Linux using a tuned BLAS. Average rates of execution were 230 MFlops for matrix matrix multiplies, 160 Mflops for Alg. III, 120 Mflops for GEMVT, 80 Mflops for LAPACK, 50 Mflops for GEMV.

|                  |                      |
|------------------|----------------------|
| 266 MHz Pentium  | CPU Clock            |
| 230 Mflops       | BLAS – 3 DGEMM       |
| 160 Mflops       | BLAS 2.5 – 3 Alg III |
| 100 – 120 Mflops | BLAS 2.5 GEMVT       |
| 80 Mflops        | BLAS 2 – 3 LAPACK    |
| 50 Mflops        | BLAS – 2GEMV         |

## Comparison with Two-Stage Bidiagonalization

Two stage bidiagonalization (B. Grosser 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, i.e., savings of 25% to 30% .

- A second stage requires  $8n^2b$  flops where  $b$  is the bandwidth attained after the first stage. As  $n$  grows large the percentage of computations in the second stage goes to zero. In the large matrix limit, two stage reduction will be faster than Algorithm III.
- Typically, the flops in the second stage will be relatively slow. Assuming that the ratios of BLAS 2.5 and 3 are as in our numeric experiments, the following table projects the breakeven size matrix for several assumed MFLOPS rate of slow flops as per cent of BLAS 3 (assuming square matrices and  $b = 20$ ).

| Size $n \times n$ | 1000 | 2000 | 4000 |
|-------------------|------|------|------|
| Rel Exec Speed    |      |      |      |
| Stage 2 Flops     | 16%  | 8%   | 4%   |
| %Stage 2 Flops    | 12%  | 6%   | 3%   |

- If singular vectors are desired, two stage methods almost double the total number of bidiagonalization 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](mailto:howell@zach.fit.edu)

# Ongoing Projects

- Optimize BLAS 2.5
- Refine Code (reduce storage requirements, adapt for sparse case).
- Timings and Test Suite. Currently the timings look very close to the theory. Example, for a 500 by 500 matrix on a Pentium II 266 MHz with tuned BLAS
  - LAPACK 4.4 seconds, 78 MFlops,
  - Algorithm III, BLAS 2.5-3.0, 2.0 seconds, 163 Mflops.
- Algorithm II for a numerically stable sparse least square algorithm

- Unsymmetric eigenvalue problem. Sparse and dense.

## Blocking for Bidiagonalization

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

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**

- BHES (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