CS 267
Dense Linear Algebra:
Parallel Gaussian Elimination

James Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr16

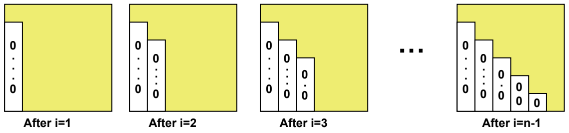
Outline

- Review Gaussian Elimination (GE) for solving Ax=b
- Optimizing GE for caches on sequential machines
  - using matrix-matrix multiplication (BLAS and LAPACK)
- Minimizing communication for sequential GE
  - Not LAPACK, but Recursive LU minimizes bandwidth (latency possible)
- Data layouts on parallel machines
- Parallel Gaussian Elimination (ScaLAPACK)
- Minimizing communication for parallel GE
  - Not ScaLAPACK (yet), but “Comm-Avoiding LU” (CALU)
  - Same idea for minimizing bandwidth and latency in sequential case
- Summarize rest of dense linear algebra
- Dynamically scheduled LU for Multicore
- LU for Heterogeneous computers (CPU + GPU)

Gaussian Elimination (GE) for solving Ax=b

- Add multiples of each row to later rows to make A upper triangular
- Solve resulting triangular system Ux = c by substitution

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
   … for each row j below row i
   for j = i+1 to n
       … add a multiple of row i to row j
       tmp = A(j,i);
       for k = i to n
           A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```



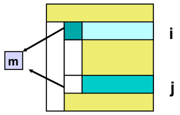
Refine GE Algorithm (1)

- Initial Version

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
   … for each row j below row i
   for j = i+1 to n
       … add a multiple of row i to row j
       tmp = A(j,i);
       for k = i to n
           A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```

- Remove computation of constant tmp/A(i,i) from inner loop.

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i to n
            A(j,k) = A(j,k) - m * A(i,k)
```



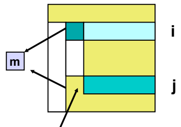
Refine GE Algorithm (2)

- Last version

```
for i = 1 to n-1
   for j = i+1 to n
      m = A(j,i)/A(i,i)
      for k = i to n
         A(j,k) = A(j,k) - m * A(i,k)
```

- Don't compute what we already know: zeros below diagonal in column i

```
for i = 1 to n-1
   for j = i+1 to n
      m = A(j,i)/A(i,i)
      for k = i+1 to n
         A(j,k) = A(j,k) - m * A(i,k)
```



Do not compute zeros

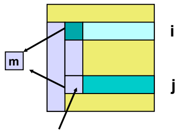
Refine GE Algorithm (3)

- Last version

```
for i = 1 to n-1
   for j = i+1 to n
      m = A(j,i)/A(i,i)
      for k = i+1 to n
         A(j,k) = A(j,k) - m * A(i,k)
```

- Store multipliers m below diagonal in zeroed entries for later use

```
for i = 1 to n-1
   for j = i+1 to n
      A(j,i) = A(j,i)/A(i,i)
      for k = i+1 to n
         A(j,k) = A(j,k) - A(j,i) * A(i,k)
```



Store m here

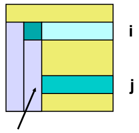
Refine GE Algorithm (4)

- Last version

```
for i = 1 to n-1
   for j = i+1 to n
      A(j,i) = A(j,i)/A(i,i)
      for k = i+1 to n
         A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- Split Loop

```
for i = 1 to n-1
   for j = i+1 to n
      A(j,i) = A(j,i)/A(i,i)
   for j = i+1 to n
      for k = i+1 to n
         A(j,k) = A(j,k) - A(j,i) * A(i,k)
```



Store all m's here before updating rest of matrix

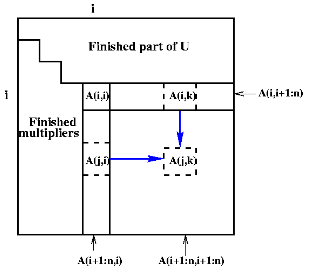
Refine GE Algorithm (5)

- Last version

```
for i = 1 to n-1
   for j = i+1 to n
      A(j,i) = A(j,i)/A(i,i)
   for j = i+1 to n
      for k = i+1 to n
         A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- Express using matrix operations (BLAS)

Work at step i of Gaussian Elimination

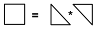


```
for i = 1 to n-1
   A(i+1:n,i) = A(i+1:n,i) * ( 1 / A(i,i) )
      … BLAS 1 (scale a vector)
   A(i+1:n,i+1:n) = A(i+1:n , i+1:n )
         - A(i+1:n , i) * A(i , i+1:n)
      … BLAS 2 (rank-1 update)
```

What GE really computes

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)     … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)   …  BLAS 2 (rank-1 update)
```

- Call the strictly lower triangular matrix of multipliers M, and let L = I+M
- Call the upper triangle of the final matrix U
- *Lemma (LU Factorization):* If the above algorithm terminates (does not divide by zero) then A = L*U

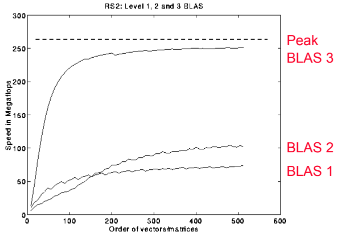


- Solving A*x=b using GE
  - Factorize A = L*U using GE (cost = 2/3 n^3 flops)
  - Solve L*y = b for y, using substitution (cost = n^2 flops)
  - Solve U*x = y for x, using substitution (cost = n^2 flops)
- Thus A*x = (L*U)*x = L*(U*x) = L*y = b as desired

Problems with basic GE algorithm

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)         … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  … BLAS 2 (rank-1 update)
          - A(i+1:n , i) * A(i , i+1:n)
```

- What if some A(i,i) is zero? Or very small?
  - Result may not exist, or be "unstable", so need to pivot
- Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures…)



Pivoting in Gaussian Elimination

- A = $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ fails completely because can't divide by A(1,1)=0
- But solving Ax=b should be easy!
- When diagonal A(i,i) is tiny (not just zero), algorithm may terminate but get completely wrong answer
  - Numerical instability
  - Roundoff error is cause
- Cure: Pivot (swap rows of A) so A(i,i) large

Gaussian Elimination with Partial Pivoting (GEPP)

- Partial Pivoting: swap rows so that A(i,i) is largest in column

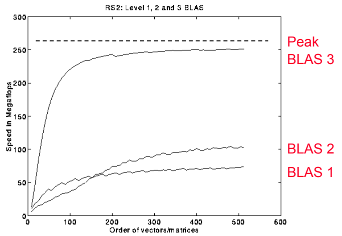
```
for i = 1 to n-1
    find and record k where |A(k,i)| = max{i ≤ j ≤ n} |A(j,i)|
        … i.e. largest entry in rest of column i
    if |A(k,i)| = 0
        exit with a warning that A is singular, or nearly so
    elseif  k ≠ i
        swap rows i and k of A
    end if
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)       … each |quotient| ≤ 1
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

- *Lemma*: This algorithm computes A = P*L*U, where P is a permutation matrix.
- This algorithm is numerically stable in practice
- For details see LAPACK code at http://www.netlib.org/lapack/single/sgetf2.f
- Standard approach – but communication costs?

Problems with basic GE algorithm

- What if some A(i,i) is zero? Or very small?
  - Result may not exist, or be "unstable", so need to pivot
- Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures…)

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)          … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  … BLAS 2 (rank-1 update)
              - A(i+1:n , i) * A(i , i+1:n)
```

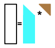


Converting BLAS2 to BLAS3 in GEPP

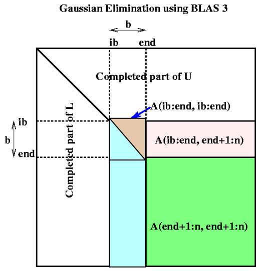
- Blocking
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP
- BIG IDEA: Delayed Updates
  - Save updates to "trailing matrix" from several consecutive BLAS2 (rank-1) updates
  - Apply many updates simultaneously in one BLAS3 (matmul) operation
- Same idea works for much of dense linear algebra
  - Not eigenvalue problems or SVD – need more ideas
- First Approach: Need to choose a block size b
  - Algorithm will save and apply b updates
  - b should be small enough so that active submatrix consisting of b columns of A fits in cache
  - b should be large enough to make BLAS3 (matmul) fast

Blocked GEPP (www.netlib.org/lapack/single/sgetrf.f)

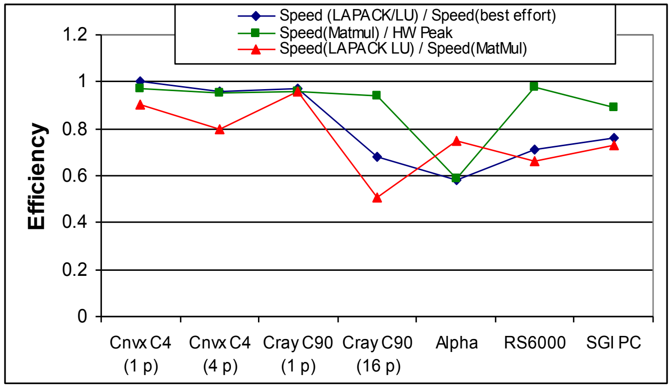
```
for   ib = 1 to n-1 step b      … Process matrix b columns at a time
    end = ib + b-1                   … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P' * L' * U'
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
    A(ib:end , end+1:n) = LL^{-1} * A(ib:end , end+1:n)        … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
              - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                            … apply delayed updates with single matrix-multiply
                            … with inner dimension b
```



(For a correctness proof, see on-line notes from CS267 / 1996.)



Efficiency of Blocked GEPP (all parallelism "hidden" inside the BLAS)



Communication Lower Bound for GE

- Matrix Multiplication can be “reduced to” GE
- Not a good way to do matmul but it shows that GE needs at least as much communication as matmul
- Does blocked GEPP minimize communication?

$$\begin{bmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{bmatrix} = \begin{bmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{bmatrix} \cdot \begin{bmatrix} I & 0 & -B \\ & I & A\cdot B \\ & & I \end{bmatrix}$$

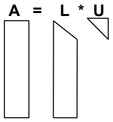
Does LAPACK’s GEPP Minimize Communication?

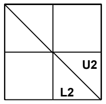
```
for   ib = 1 to n-1 step b      … Process matrix b columns at a time
    end = ib + b-1                … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P’ * L’ * U’
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
    A(ib:end , end+1:n) = LL^-1 * A(ib:end , end+1:n)        … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
              - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                          … apply delayed updates with single matrix-multiply
                          … with inner dimension b
```

- Case 1: $n \geq M$ - huge matrix – attains lower bound
  - $b = M^{1/2}$ optimal, dominated by matmul
- Case 2: $n \leq M^{1/2}$ - small matrix – attains lower bound
  - Whole matrix fits in fast memory, any algorithm attains lower bound
- Case 3: $M^{1/2} < n < M$ - medium size matrix – not optimal
  - Can’t choose b to simultaneously optimize matmul and BLAS2 GEPP of n x b submatrix
  - Worst case: Exceed lower bound by factor $M^{1/6}$ when $n = M^{2/3}$
- Detailed counting on backup slides

Alternative cache-oblivious GE formulation (1/2)

- Toledo (1997)
  - Describe without pivoting for simplicity
  - “Do left half of matrix, then right half”





```
function [L,U] = RLU (A)   … assume A is m by n
    if (n=1)   L = A/A(1,1),  U = A(1,1)
    else
        [L1,U1] = RLU( A(1:m , 1:n/2)) … do left half of A
            … let L11 denote top n/2 rows of L1
        A( 1:n/2 , n/2+1 : n ) = L11^-1 * A( 1:n/2 , n/2+1 : n )
            … update top n/2 rows of right half of A
        A( n/2+1: m, n/2+1:n ) = A( n/2+1: m, n/2+1:n )
            - A( n/2+1: m, 1:n/2 ) * A( 1:n/2 , n/2+1 : n )
            … update rest of right half of A
        [L2,U2] = RLU( A(n/2+1:m , n/2+1:n) ) … do right half of A
        return [ L1,[0;L2] ] and [U1, [ A(.,.) ; U2 ] ]
```

Alternative cache-oblivious GE formulation (2/2)

```
function [L,U] = RLU (A)   … assume A is m by n
    if (n=1)   L = A/A(1,1),  U = A(1,1)
    else
        [L1,U1] = RLU( A(1:m , 1:n/2)) … do left half of A
            … let L11 denote top n/2 rows of L1
        A( 1:n/2 , n/2+1 : n ) = L11^-1 * A( 1:n/2 , n/2+1 : n )
            … update top n/2 rows of right half of A
        A( n/2+1: m, n/2+1:n ) = A( n/2+1: m, n/2+1:n )
            - A( n/2+1: m, 1:n/2 ) * A( 1:n/2 , n/2+1 : n )
            … update rest of right half of A
        [L2,U2] = RLU( A(n/2+1:m , n/2+1:n) ) … do right half of A
        return [ L1,[0;L2] ] and [U1, [ A(.,.) ; U2 ] ]
```

- $W(m,n) = W(m,n/2) + O(\max(m\cdot n, m\cdot n^2/M^{1/2})) + W(m-n/2,n/2)$

$\leq 2 \cdot W(m,n/2) + O(\max(m\cdot n, m\cdot n^2/M^{1/2}))$

$= O(m\cdot n^2/M^{1/2} + m\cdot n\cdot \log M)$

$= O(m\cdot n^2/M^{1/2})$ if $M^{1/2}\cdot \log M = O(n)$

Still doesn’t minimize latency, but fixable CLASS PROJECT

Explicitly Parallelizing Gaussian Elimination

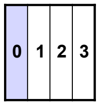
- Parallelization steps
  - Decomposition: identify enough parallel work, but not too much
  - Assignment: load balance work among threads
  - Orchestrate: communication and synchronization
  - Mapping: which processors execute which threads (locality)
- Decomposition
  - In BLAS 2 algorithm nearly each flop in inner loop can be done in parallel, so with n^2 processors, need 3n parallel steps, O(n log n) with pivoting

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)          … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )   … BLAS 2 (rank-1 update)
            - A(i+1:n , i) * A(i , i+1:n)
```

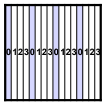
  - This is too fine-grained, prefer calls to local matmuls instead
  - Need to use parallel matrix multiplication
- Assignment and Mapping
  - Which processors are responsible for which submatrices?

Different Data Layouts for Parallel GE

Bad load balance: P0 idle after first n/4 steps



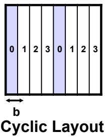
1) 1D Column Blocked Layout



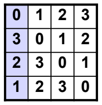
Load balanced, but can’t easily use BLAS3

2) 1D Column Cyclic Layout

Can trade load balance and BLAS3 performance by choosing b, but factorization of block column is a bottleneck



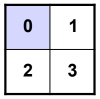
3) 1D Column Block Cyclic Layout



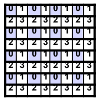
Complicated addressing, May not want full parallelism In each column, row

4) Block Skewed Layout

Bad load balance: P0 idle after first n/2 steps



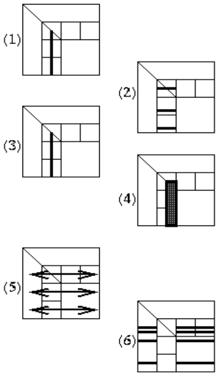
5) 2D Row and Column Blocked Layout



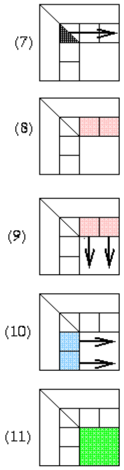
The winner!

6) 2D Row and Column Block Cyclic Layout

Distributed Gaussian Elimination with a 2D Block Cyclic Layout



```
for ib = 1 to n−1 step b
    end = min( ib+b−1, n )
    for i = ib to end
    (1)   find pivot row k, column broadcast
    (2)   swap rows k and i in block column, broadcast row k
    (3)   A( i+1:n , i ) = A( i+1:n , i ) / A( i , i )
    (4)   A( i+1:n , i+1:end) −= A( i+1:n , i ) * A( i , i+1:end )
    end for
(5) broadcast all swap information right and left
(6) apply all rows swaps to other columns
```



(7) Broadcast LL right

(8) A(ib:end , end+1:n) = LL \ A(ib:end , end+1:n)

(9) Broadcast A(ib:end , end+1:n) down

(10) Broadcast A(end+1:n , ib:end) right

(11) Eliminate A(end+1:n , end+1:n)

Matrix multiply of green = green - blue * pink

Review of Parallel MatMul

- **Want Large Problem Size Per Processor**

PDGEMM = PBLAS matrix multiply

Observations:

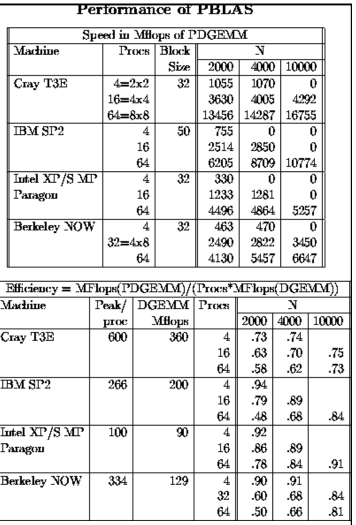
- **For fixed N, as P increasesn Mflops increases, but less than 100% efficiency**
- **For fixed P, as N increases, Mflops (efficiency) rises**

DGEMM = BLAS routine for matrix multiply

Maximum speed for PDGEMM = # Procs * speed of DGEMM

Observations:

- **Efficiency always at least 48%**
- **For fixed N, as P increases, efficiency drops**
- **For fixed P, as N increases, efficiency increases**



Performance of PBLAS

Speed in Mflops of PDGEMM

Machine	Procs	Block	N		
		Size	2000	4000	10000
Cray T3E	4=2x2	32	1055	1070	0
	16=4x4		3630	4005	4292
	64=8x8		13456	14287	16755
IBM SP2	4	50	755	0	0
	16		2514	2850	0
	64		6205	8709	10774
Intel XP/S MP Paragon	4	32	330	0	0
	16		1233	1281	0
	64		4496	4864	5257
Berkeley NOW	4	32	463	470	0
	32=4x8		2490	2822	3450
	64		4130	5457	6647

Efficiency = MFlops(PDGEMM)/(Procs*MFlops(DGEMM))

Machine	Peak/	DGEMM	Procs	N		
	proc	Mflops		2000	4000	10000
Cray T3E	600	360	4	.73	.74	
			16	.63	.70	.75
			64	.58	.62	.73
IBM SP2	266	200	4	.94		
			16	.79	.89	
			64	.48	.68	.84
Intel XP/S MP Paragon	100	90	4	.92		
			16	.86	.89	
			64	.78	.84	.91
Berkeley NOW	334	129	4	.90	.91	
			32	.60	.68	.84
			64	.50	.66	.81

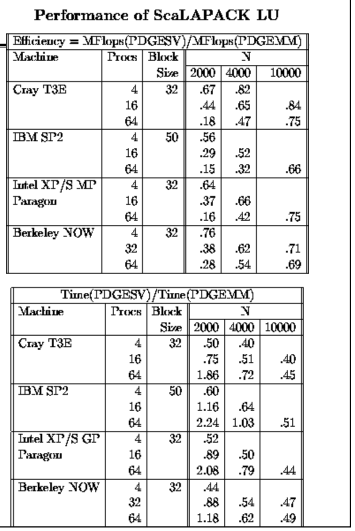
PDGESV = ScaLAPACK Parallel LU

Since it can run no faster than its inner loop (PDGEMM), we measure:

Efficiency = Speed(PDGESV)/Speed(PDGEMM)

Observations:

- **Efficiency well above 50% for large enough problems**
- **For fixed N, as P increases, efficiency decreases (just as for PDGEMM)**
- **For fixed P, as N increases efficiency increases (just as for PDGEMM)**
- **From bottom table, cost of solving**
  - **Ax=b about half of matrix multiply for large enough matrices.**
  - **From the flop counts we would expect it to be $(2*n^3)/(2/3*n^3) = 3$ times faster, but communication makes it a little slower.**



Performance of ScaLAPACK LU

Efficiency = MFlops(PDGESV)/MFlops(PDGEMM)

Machine	Procs	Block	N		
		Size	2000	4000	10000
Cray T3E	4	32	.67	.82	
	16		.44	.65	.84
	64		.18	.47	.75
IBM SP2	4	50	.56		
	16		.29	.52	
	64		.15	.32	.66
Intel XP/S MP Paragon	4	32	.64		
	16		.37	.66	
	64		.16	.42	.75
Berkeley NOW	4	32	.76		
	32		.38	.62	.71
	64		.28	.54	.69

Time(PDGESV)/Time(PDGEMM)

Machine	Procs	Block	N		
		Size	2000	4000	10000
Cray T3E	4	32	.50	.40	
	16		.75	.51	.40
	64		1.86	.72	.45
IBM SP2	4	50	.60		
	16		1.16	.64	
	64		2.24	1.03	.51
Intel XP/S GP Paragon	4	32	.52		
	16		.89	.50	
	64		2.08	.79	.44
Berkeley NOW	4	32	.44		
	32		.88	.54	.47
	64		1.18	.62	.49

Does ScaLAPACK Minimize Communication?

- **Lower Bound: $O(n^2 / P^{1/2})$ words sent in $O(P^{1/2})$ mess.**
  - **Attained by Cannon and SUMMA (nearly) for matmul**
- **ScaLAPACK:**
  - **$O(n^2 \log P / P^{1/2})$ words sent – close enough**
  - **O(n log P) messages – too large**
  - **Why so many? One reduction costs O(log P) per column to find maximum pivot, times n = #columns**
- **Need to abandon partial pivoting to reduce #messages**
  - **Suppose we have n x n matrix on $P^{1/2}$ x $P^{1/2}$ processor grid**
  - **Goal: For each panel of b columns spread over $P^{1/2}$ procs, identify b "good" pivot rows in one reduction**
    - **Call this factorization TSLU = "Tall Skinny LU"**
  - **Several natural bad (numerically unstable) ways explored, but good way exists**
    - **SC08, "Communication Avoiding GE", D., Grigori, Xiang**

Choosing Rows by "Tournament Pivoting"

$$W^{nxb} = \left(\begin{array}{c} W_1 \\ \hline W_2 \\ \hline W_3 \\ \hline W_4 \end{array}\right) = \left(\begin{array}{c} P_1 \cdot L_1 \cdot U_1 \\ \hline P_2 \cdot L_2 \cdot U_2 \\ \hline P_3 \cdot L_3 \cdot U_3 \\ \hline P_4 \cdot L_4 \cdot U_4 \end{array}\right)$$

Choose b pivot rows of W_1, call them W_1'
Choose b pivot rows of W_2, call them W_2'
Choose b pivot rows of W_3, call them W_3'
Choose b pivot rows of W_4, call them W_4'

$$\left(\begin{array}{c} W_1' \\ W_2' \\ \hline W_3' \\ W_4' \end{array}\right) = \left(\begin{array}{c} P_{12} \cdot L_{12} \cdot U_{12} \\ \hline P_{34} \cdot L_{34} \cdot U_{34} \end{array}\right)$$

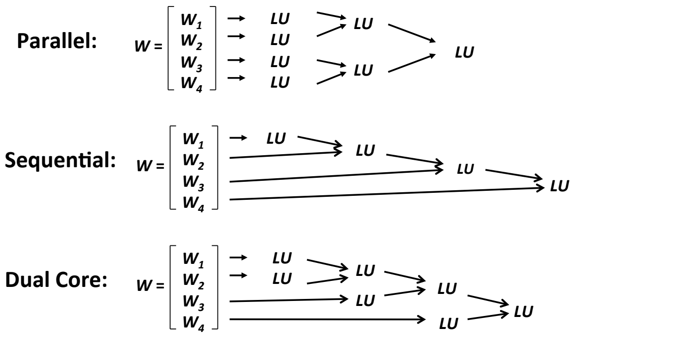
Choose b pivot rows, call them W_{12}'
Choose b pivot rows, call them W_{34}'

$$\left(\begin{array}{c} W_{12}' \\ W_{34}' \end{array}\right) = P_{1234} \cdot L_{1234} \cdot U_{1234}$$

Choose b pivot rows

Go back to W and use these b pivot rows
(move them to top, do LU without pivoting)
Not the same pivots rows chosen as for GEPP
Need to show numerically stable (D., Grigori, Xiang, '11)

Minimizing Communication in TSLU



Parallel: $W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix}$ → LU, LU, LU, LU → LU, LU → LU

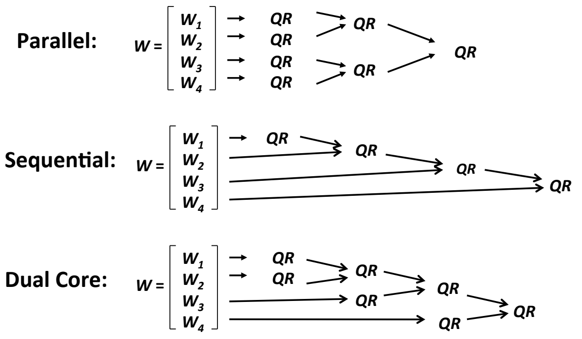
Sequential: $W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix}$ → LU → LU → LU → LU

Dual Core: $W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix}$ → LU, LU → LU → LU → LU; LU → LU

Multicore / Multisocket / Multirack / Multisite / Out-of-core: ?

Can Choose reduction tree dynamically

Same idea for QR of Tall-skinny matrix (TSQR)



Parallel: $W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix}$ → QR, QR, QR, QR → QR, QR → QR

Sequential: $W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix}$ → QR → QR → QR → QR

Dual Core: $W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \end{bmatrix}$ → QR, QR → QR → QR → QR; QR → QR

First step of SVD of Tall-Skinny matrix

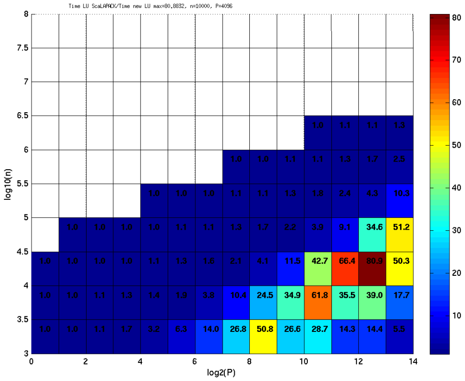
Performance vs ScaLAPACK LU

- **TSLU**
  - **IBM Power 5**
    - **Up to 4.37x faster (16 procs, 1M x 150)**
  - **Cray XT4**
    - **Up to 5.52x faster (8 procs, 1M x 150)**
- **CALU**
  - **IBM Power 5**
    - **Up to 2.29x faster (64 procs, 1000 x 1000)**
  - **Cray XT4**
    - **Up to 1.81x faster (64 procs, 1000 x 1000)**
- **See INRIA Tech Report 6523 (2008), paper at SC08**

TSQR Performance Results

- Parallel
  - Intel Clovertown
    - Up to **8x** speedup (8 core, dual socket, 10M x 10)
  - Pentium III cluster, Dolphin Interconnect, MPICH
    - Up to **6.7x** speedup (16 procs, 100K x 200)
  - BlueGene/L
    - Up to **4x** speedup (32 procs, 1M x 50)
  - Tesla C 2050 / Fermi
    - Up to **13x** (110,592 x 100)
  - Grid – **4x** on 4 cities vs 1 city (Dongarra, Langou et al)
  - Cloud – (Gleich and Benson) ~2 map-reduces
- Sequential
  - "Infinite speedup" for out-of-core on PowerPC laptop
    - As little as 2x slowdown vs (predicted) infinite DRAM
    - LAPACK with virtual memory never finished
- SVD costs about the same
- Joint work with Grigori, Hoemmen, Langou, Anderson, Ballard, Keutzer, others

CALU speedup prediction for a Petascale machine - up to 81x faster



Petascale machine with 8192 procs, each at 500 GFlops/s, a bandwidth of 4 GB/s.

$$\gamma = 2\cdot 10^{-12} s, \alpha = 10^{-5} s, \beta = 2\cdot 10^{-9} s/word.$$

Summary of dense *sequential* O(n^3) algorithms attaining communication lower bounds

- References are from Table 3.1 in "Communication lower bounds and optimal algorithms for numerical linear algebra", Ballard et al, 2014
  - #words moved = $\Omega(n^3/M^{1/2})$, #messages = $\Omega(n^3/M^{3/2})$
- Cache-oblivious, Ours, LAPACK, *Randomized*

Computation	2-Level Mem		Multiple Level	
BLAS-3				
Cholesky				
LU				
Sym Indef	[10]	[10]	[10]	[10]

Summary of dense *parallel* O(n^3/p) algorithms attaining communication lower bounds

- References are from Table 3.2 in "Communication lower bounds and optimal algorithms for numerical linear algebra", Ballard et al, 2014
- Assume nxn matrices on p procs, minimum memory er proc: M = O(n^2/p)
  - #words moved = $\Omega(n^2/p^{1/2})$, #messages = Ω(
- Ours, ScaLAPACK, *Randomized*

Computation	Minimizes #	Minimizes # Messages
BLAS3		
Cholesky		
LU		
Symmetric Inde ite		
QR		
Eig(A=A^T) and SVD		
Eig(A)	[illegible]	[illegible]

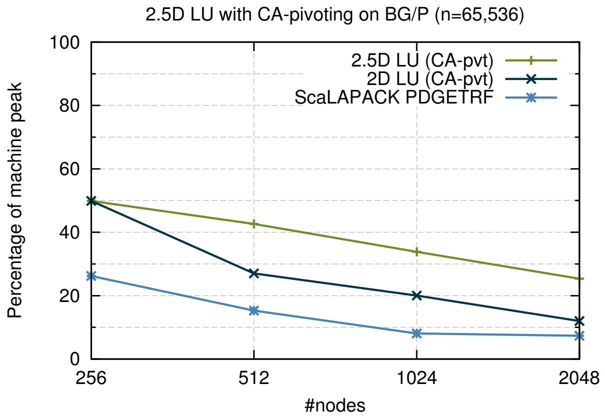
CLASS PROJECTS

Can we do even better?

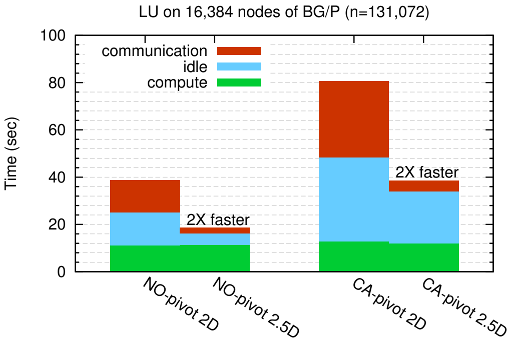
- Assume nxn matrices on p processors
- **Use c copies of data:** M = O(cn^2 / p) per processor
- Increasing M reduces lower bounds:
  - #words_moved = $\Omega((n^3/ P) / M^{1/2}) = \Omega(n^2 / (c^{1/2} P^{1/2}))$
  - #messages = $\Omega((n^3/ P) / M^{3/2}$ $\Omega(P^{1/2} / c^{3/2})$
- Attainable for Matmul
- *Not* attainable for LU, Choleskv
- Thm: #words_moved * #me $\Omega(n^2)$
  - Lowering #words b *must increase* #messages by same factor
  - Cor: Perfe caling impossible for LU, Cholesky,QR
- Both lower attainable for Cholesky, LU, QR (via Cho ky QR):
  - #words_moved = $\Omega(n^2 / (c^{1/2} P^{1/2}))$
  - #messages = $\Omega(c^{1/2} P^{1/2})$

CLASS PROJECTS

LU Speedups from Tournament Pivoting and 2.5D



2.5D vs 2D LU With and Without Pivoting



Dense Linear Algebra on Recent Architectures

- Multicore
  - How do we schedule all parallel tasks to minimize idle time?
- GPUs
  - Heterogeneous computer: consists of functional units (CPU and GPU) that are good at different tasks
  - How do we divide the work between the GPU and CPU to take maximal advantage of both?
  - Challenging now, will get more so as platforms become more heterogeneous

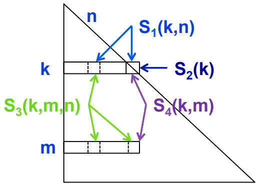
Multicore: Expressing Parallelism with a DAG

- DAG = Directed Acyclic Graph
  - S1 → S2 means statement S2 “depends on” statement S1
  - Can execute in parallel any Si without input dependencies
- For simplicity, consider Cholesky $A = LL^T$, not LU
  - N by N matrix, numbered from A(0,0) to A(N-1,N-1)
  - “Left looking” code: at step k, completely compute column k of L

```
for k = 0 to N-1
    for n = 0 to k-1
        A(k,k) = A(k,k) – A(k,n)*A(k,n)
    A(k,k) = sqrt(A(k,k))
    for m = k+1 to N-1
        for n = 0 to k-1
            A(m,k) = A(m,k) – A(m,n)*A(k,n)
        A(m,k) = A(m,k) / A(k,k)
```

Expressing Parallelism with a DAG - Cholesky

```
                for k = 0 to N-1
                   for n = 0 to k-1
$S_1(k,n)$               A(k,k) = A(k,k) – A(k,n)*A(k,n)
$S_2(k)$              A(k,k) = sqrt(A(k,k))
                   for m = k+1 to N-1
                        for n = 0 to k-1
$S_3(k,m,n)$                  A(m,k) = A(m,k) – A(m,n)*A(k,n)
$S_4(k,m)$               A(m,k) = A(m,k) · $A(k,k)^{-1}$
```



DAG has ≈$N^3/6$ vertices:

$S_1(k,n) \rightarrow S_2(k)$ for n=0:k-1

$S_3(k,m,n) \rightarrow S_4(k,m)$ for n=0:k-1

$S_2(k) \rightarrow S_4(k,m)$ for m=k+1:N

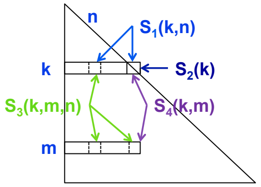
$S_4(k,m) \rightarrow S_3(k',m,k)$ for k' >k

$S_4(k,m) \rightarrow S_3(k,m',k)$ for m' >m

Expressing Parallelism with a DAG – Block Cholesky

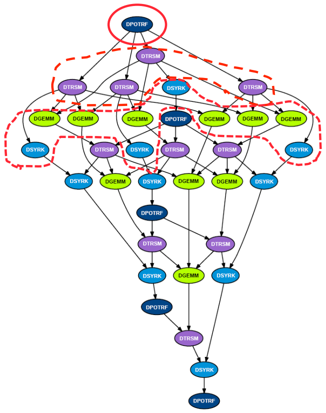
- Each A[i,j] is a b-by-b block

```
                            for k = 0 to N/b-1
                               for n = 0 to k-1
SYRK:     $S_1(k,n)$               A[k,k] = A[k,k] – A[k,n]*$A[k,n]^T$
POTRF:    $S_2(k)$              A[k,k] = unblocked_Cholesky(A[k,k])
                               for m = k+1 to N/b-1
                                    for n = 0 to k-1
GEMM:     $S_3(k,m,n)$                  A[m,k] = A[m,k] – A[m,n]*$A[k,n]^T$
TRSM:     $S_4(k,m)$               A[m,k] = A[m,k] · $A[k,k]^{-1}$
```



Same DAG, but only ≈$(N/b)^3/6$ vertices

Sample Cholesky DAG with #blocks in any row or column = N/b = 5



- Note implied order of summation from left to right
- Not necessary for correctness, but it does reflect what the sequential code does
- Can process DAG in any order respecting dependences

Slide courtesy of Jakub Kurzak, UTK

Scheduling options

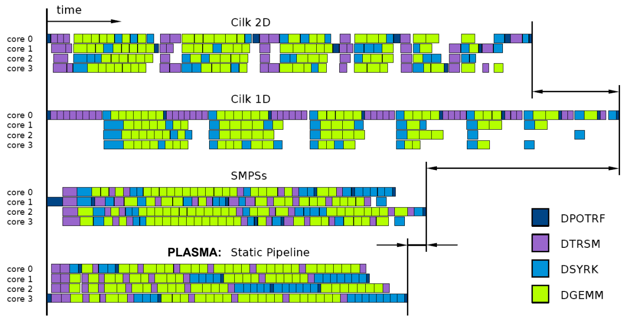
- *Static* (pre-assign tasks to processors) vs *Dynamic* (idle processors grab ready jobs from work-queue)
  - If dynamic, does scheduler take user hints/priorities?
- Respect locality (eg processor must have some task data in its cache) vs not
- Build and store entire DAG to schedule it (which may be very large, $(N/b)^3$), vs Build just the next few “levels” at a time (smaller, but less information for scheduler)
- Programmer builds DAG & schedule vs Depend on compiler or run-time system
  - Ease of programming, vs not exploiting user knowledge
  - If compiler, how conservative is detection of parallelism?
  - Generally useful, not just linear algebra

Schedulers tested

- Cilk
  - programmer-defined parallelism
  - spawn – creates independent tasks
  - sync – synchronizes a sub-branch of the tree
- SMPSs
  - dependency-defined parallelism
  - pragma-based annotation of tasks (directionality of the parameters)
- PLASMA (Static Pipeline)
  - programmer-defined (hard-coded)
  - apriori processing order
  - stalling on dependencies

Slide courtesy of Jakub Kurzak, UTK

Measured Results for Tiled Cholesky



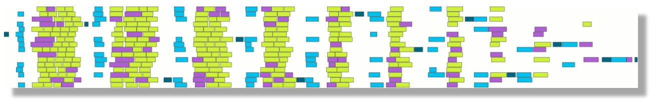
- Measured on Intel Tigerton 2.4 GHz
- Cilk 1D: one task is whole panel, but with “look ahead”
- Cilk 2D: tasks are blocks, scheduler steals work, little locality
- PLASMA works best

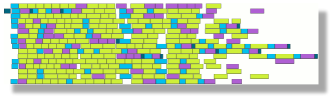
Slide courtesy of Jakub Kurzak, UTK

More Measured Results for Tiled Cholesky

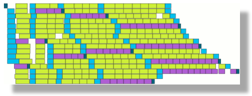
- Measured on Intel Tigerton 2.4 GHz

Cilk





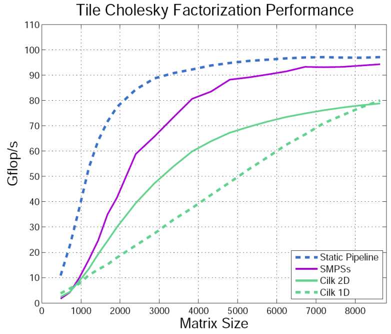
SMPSs



PLASMA (Static Pipeline)

Slide courtesy of Jakub Kurzak, UTK

Still More Measured Results for Tiled Cholesky



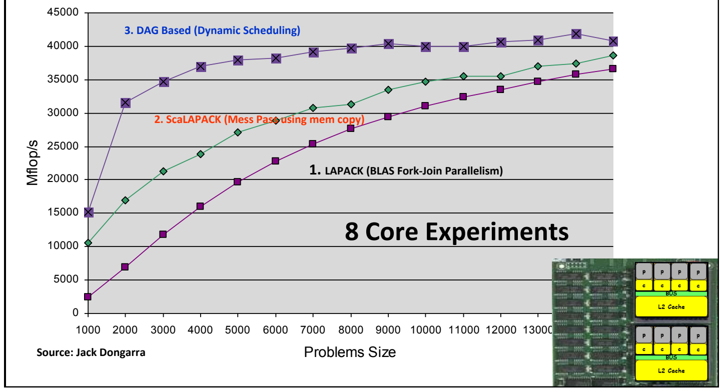
- PLASMA (static pipeline) – best
- SMPSs – somewhat worse
- Cilk 2D – inferior
- Cilk 1D – still worse

quad-socket, quad-core (16 cores total) Intel Tigerton 2.4 GHz

Slide courtesy of Jakub Kurzak, UTK

Intel's Clovertown Quad Core

3 Implementations of LU factorization
Quad core w/2 sockets per board, w/ 8 Threads



Source: Jack Dongarra

Scheduling on Multicore – Next Steps

- PLASMA 2.8.0 released 12/2015
  - Includes BLAS, Cholesky, QR, LU, LDL^T, eig, svd
  - icl.cs.utk.edu/plasma/
- Future of PLASMA
  - Continue adding functions
  - Add dynamic scheduling
    - QUARK dynamic scheduler released 12/2011
    - DAGs for eigenproblems are too complicated to do by hand
    - Plan to adopt OpenMP4.0 DAG scheduling features
  - Still assume homogeneity of available cores
    - What about GPUs, or mixtures of CPUs and GPUs?
  - MAGMA
    - icl.cs.utk.edu/magma

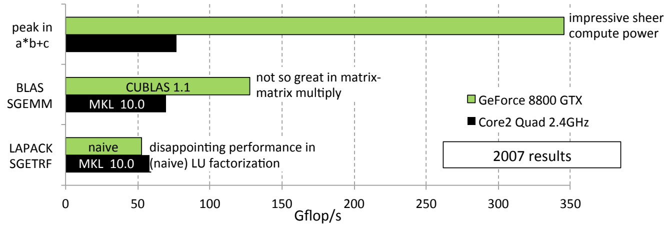
CLASS PROJECTS

Dense Linear Algebra on GPUs

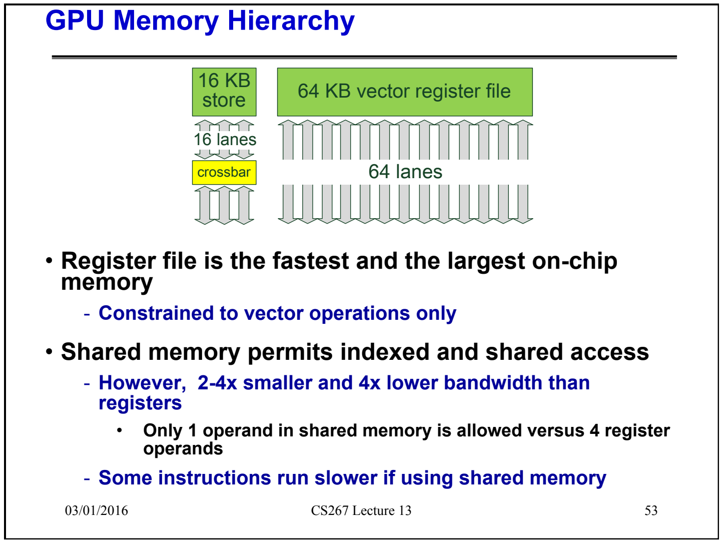
- Source: Vasily Volkov's SC08 paper
  - Best Student Paper Award (729 citations)
- New challenges
  - More complicated memory hierarchy
  - Not like "L1 inside L2 inside …",
    - Need to choose which memory to use carefully
    - Need to move data manually
  - GPU does some operations much faster than CPU, but not all
  - CPU and GPU fastest using different data layouts

Motivation

- NVIDIA released CUBLAS 1.0 in 2007, which is BLAS for GPUs
- This enables a straightforward port of LAPACK to GPU
  - Consider single precision only

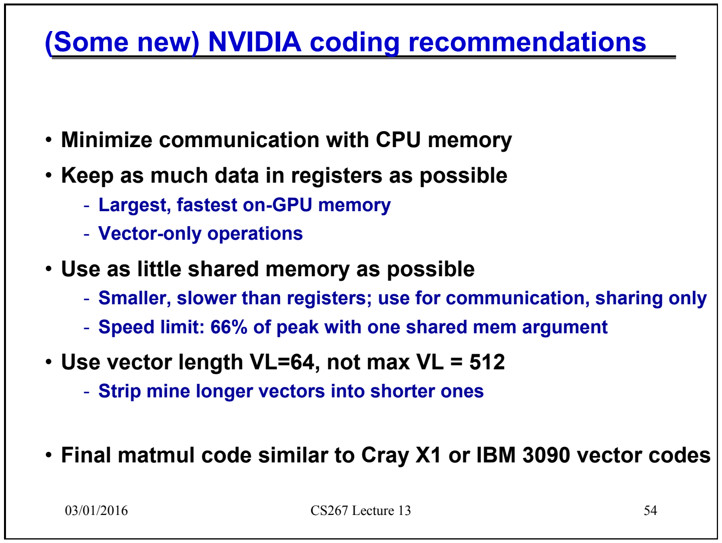


- Goal: understand bottlenecks in the dense linear algebra kernels
  - Requires detailed understanding of the GPU architecture
  - Result 1: New coding recommendations for high performance on GPUs
  - Result 2: New , fast variants of LU, QR, Cholesky, other routines



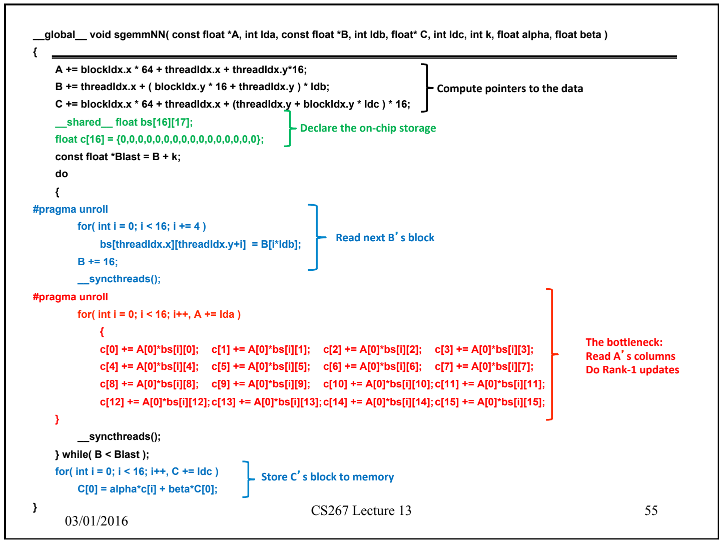
GPU Memory Hierarchy

- Register file is the fastest and the largest on-chip memory
  - Constrained to vector operations only
- Shared memory permits indexed and shared access
  - However, 2-4x smaller and 4x lower bandwidth than registers
    - Only 1 operand in shared memory is allowed versus 4 register operands
  - Some instructions run slower if using shared memory

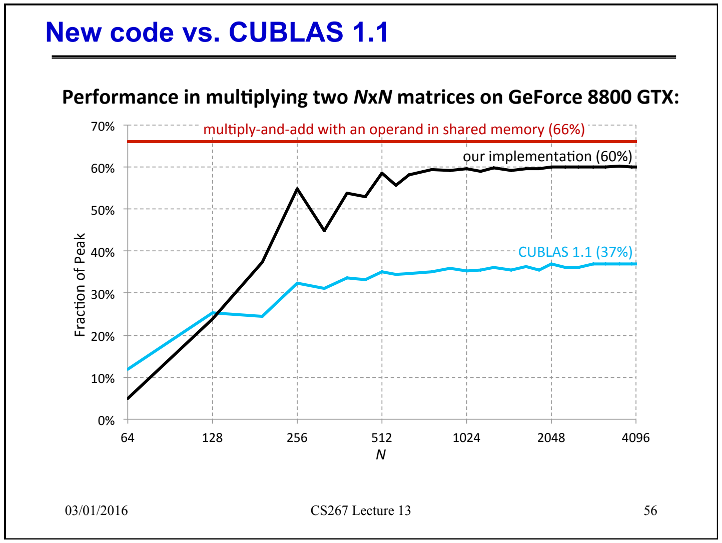


(Some new) NVIDIA coding recommendations

- Minimize communication with CPU memory
- Keep as much data in registers as possible
  - Largest, fastest on-GPU memory
  - Vector-only operations
- Use as little shared memory as possible
  - Smaller, slower than registers; use for communication, sharing only
  - Speed limit: 66% of peak with one shared mem argument
- Use vector length VL=64, not max VL = 512
  - Strip mine longer vectors into shorter ones
- Final matmul code similar to Cray X1 or IBM 3090 vector codes

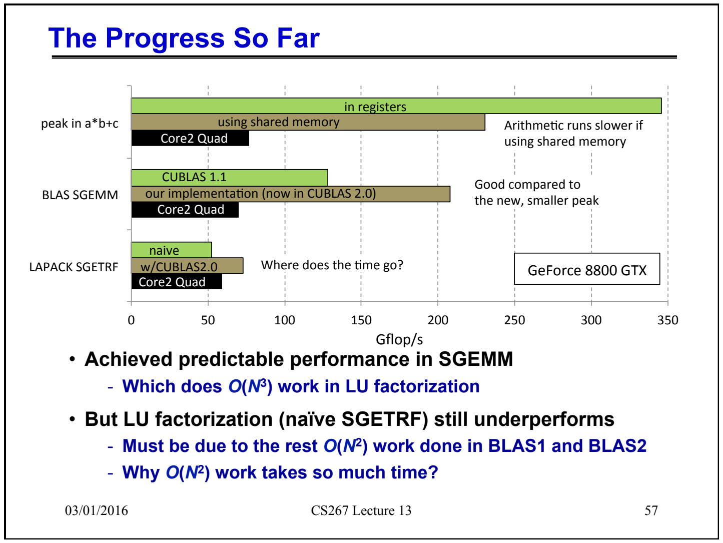


```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
     A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
     B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;             Compute pointers to the data
     C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;
     __shared__ float bs[16][17];
     float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};    Declare the on-chip storage
     const float *Blast = B + k;
     do
     {
#pragma unroll
          for( int i = 0; i < 16; i += 4 )
               bs[threadIdx.x][threadIdx.y+i]  = B[i*ldb];     Read next B' s block
          B += 16;
          __syncthreads();
#pragma unroll
          for( int i = 0; i < 16; i++, A += lda )
               {
               c[0] += A[0]*bs[i][0];    c[1] += A[0]*bs[i][1];    c[2] += A[0]*bs[i][2];    c[3] += A[0]*bs[i][3];
               c[4] += A[0]*bs[i][4];    c[5] += A[0]*bs[i][5];    c[6] += A[0]*bs[i][6];    c[7] += A[0]*bs[i][7];      The bottleneck:
               c[8] += A[0]*bs[i][8];    c[9] += A[0]*bs[i][9];    c[10] += A[0]*bs[i][10]; c[11] += A[0]*bs[i][11];   Read A' s columns
               c[12] += A[0]*bs[i][12]; c[13] += A[0]*bs[i][13]; c[14] += A[0]*bs[i][14]; c[15] += A[0]*bs[i][15];  Do Rank-1 updates
     }
          __syncthreads();
     } while( B < Blast );
     for( int i = 0; i < 16; i++, C += ldc )
          C[0] = alpha*c[i] + beta*C[0];          Store C' s block to memory
}
```



New code vs. CUBLAS 1.1

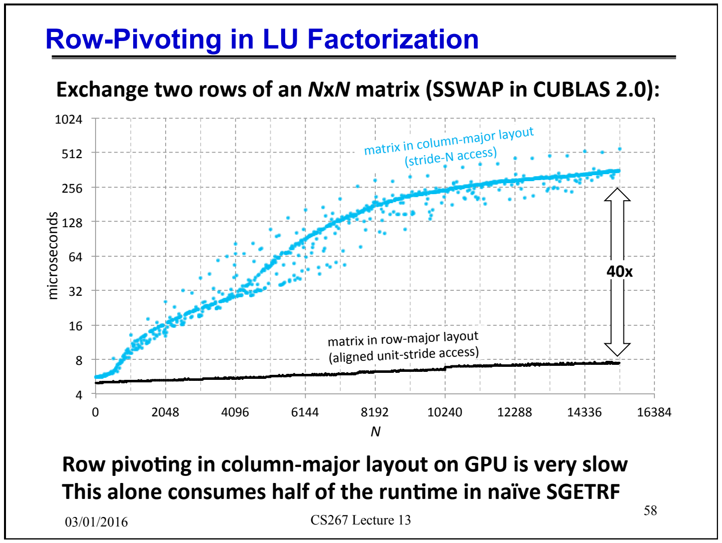
Performance in multiplying two *NxN* matrices on GeForce 8800 GTX:



The Progress So Far

- Achieved predictable performance in SGEMM
  - Which does $O(N^3)$ work in LU factorization
- But LU factorization (naïve SGETRF) still underperforms
  - Must be due to the rest $O(N^2)$ work done in BLAS1 and BLAS2
  - Why $O(N^2)$ work takes so much time?

03/01/2016 CS267 Lecture 13 57

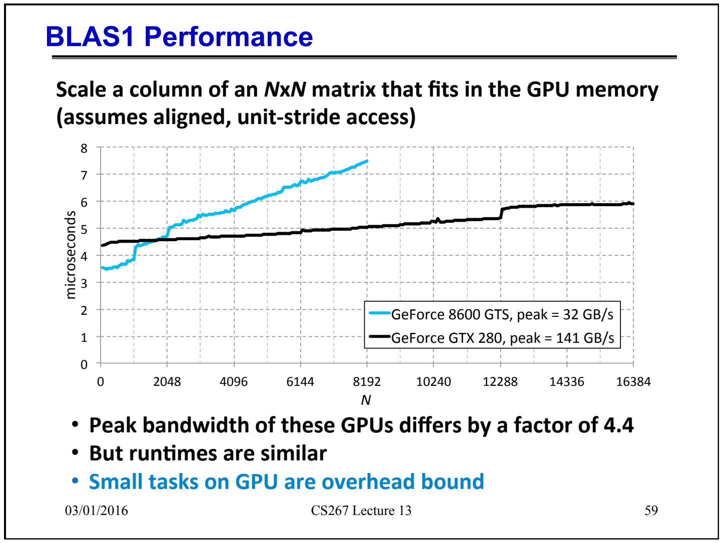


Row-Pivoting in LU Factorization

Exchange two rows of an *NxN* matrix (SSWAP in CUBLAS 2.0):

Row pivoting in column-major layout on GPU is very slow
This alone consumes half of the runtime in naïve SGETRF

03/01/2016 CS267 Lecture 13 58

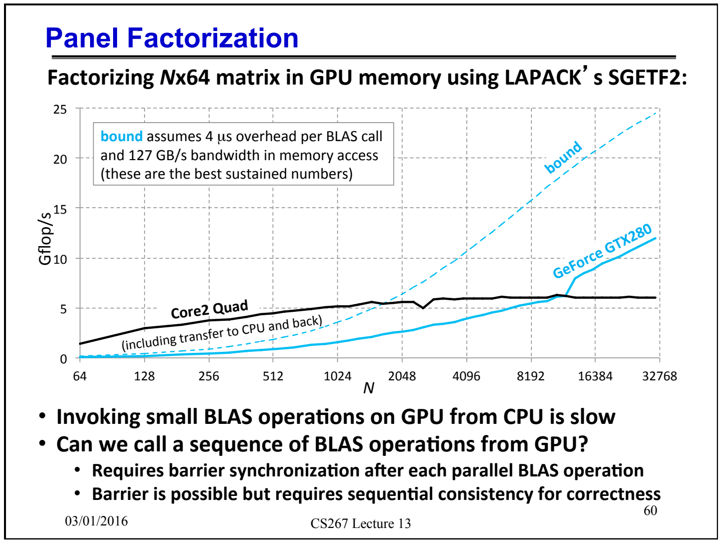


BLAS1 Performance

Scale a column of an *NxN* matrix that fits in the GPU memory (assumes aligned, unit-stride access)

- Peak bandwidth of these GPUs differs by a factor of 4.4
- But runtimes are similar
- Small tasks on GPU are overhead bound

03/01/2016 CS267 Lecture 13 59

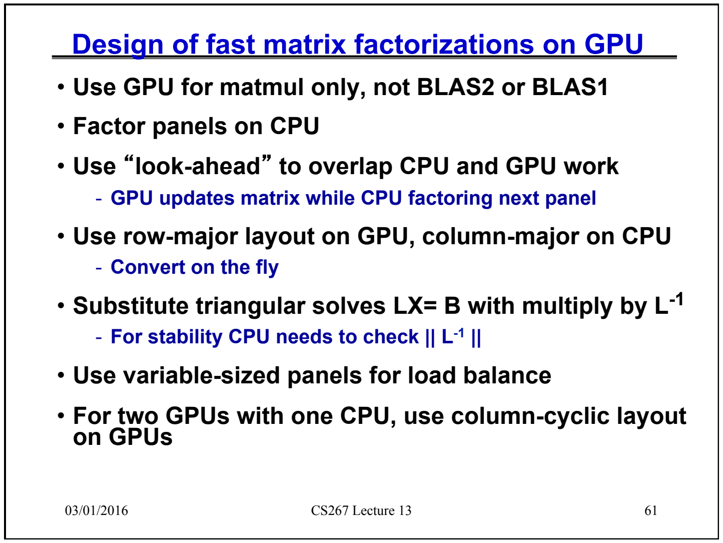


Panel Factorization

Factorizing *N*x64 matrix in GPU memory using LAPACK's SGETF2:

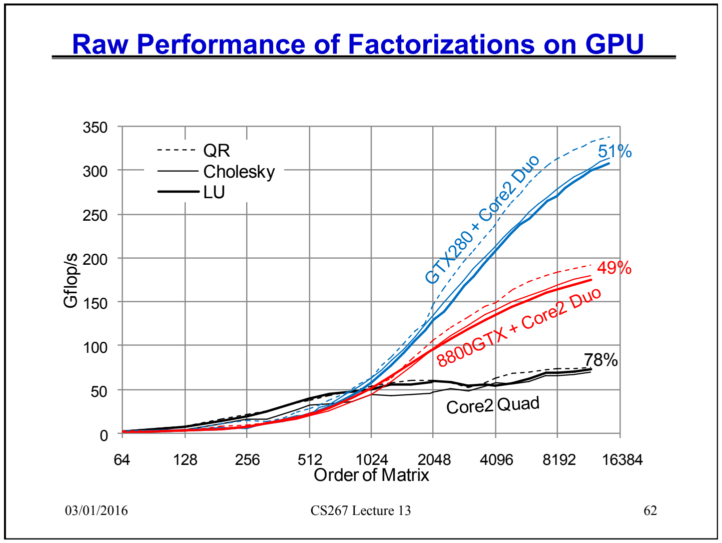
- Invoking small BLAS operations on GPU from CPU is slow
- Can we call a sequence of BLAS operations from GPU?
  - Requires barrier synchronization after each parallel BLAS operation
  - Barrier is possible but requires sequential consistency for correctness

03/01/2016 CS267 Lecture 13 60

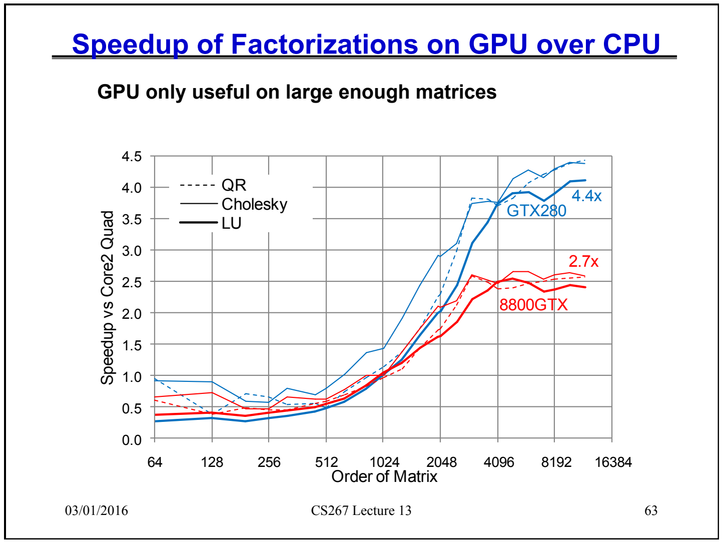


Design of fast matrix factorizations on GPU

- Use GPU for matmul only, not BLAS2 or BLAS1
- Factor panels on CPU
- Use “look-ahead” to overlap CPU and GPU work
  - GPU updates matrix while CPU factoring next panel
- Use row-major layout on GPU, column-major on CPU
  - Convert on the fly
- Substitute triangular solves LX= B with multiply by L^{-1}
  - For stability CPU needs to check $|| L^{-1} ||$
- Use variable-sized panels for load balance
- For two GPUs with one CPU, use column-cyclic layout on GPUs

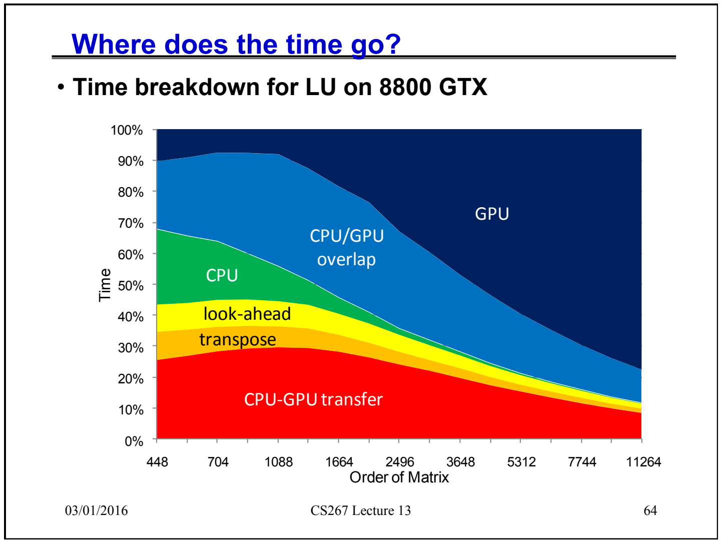


Raw Performance of Factorizations on GPU



Speedup of Factorizations on GPU over CPU

GPU only useful on large enough matrices

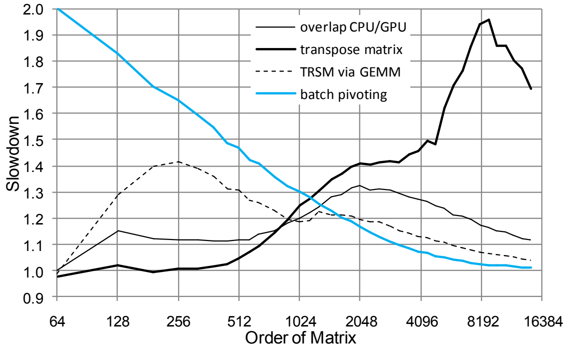


Where does the time go?

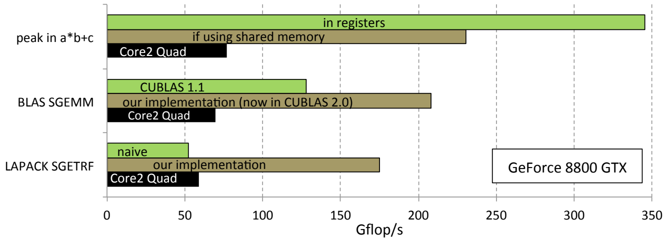
- Time breakdown for LU on 8800 GTX

Importance of various optimizations on GPU

- Slowdown when omitting one of the optimizations on GTX 280



Results for matmul, LU on NVIDIA



- What we' ve achieved:
  - Identified realistic peak speed of GPU architecture
  - Achieved a large fraction of this peak in matrix multiply
  - Achieved a large fraction of the matrix multiply rate in dense factorizations

Class Projects

- Pick one (of many) functions/algorithms
- Pick a target parallel platform
- Pick a "parallel programming framework"
  - LAPACK – all parallelism in BLAS
  - ScaLAPACK – distributed memory using MPI
  - PLASMA – DAG scheduling on multicore
    - Parallel Linear Algebra for Scalable Multi-core Architectures
    - http://icl.cs.utk.edu/plasma/
  - MAGMA – DAG scheduling for heterogeneous platforms
    - Matrix Algebra on GPU and Multicore Architectures
    - http://icl.cs.utk.edu/magma/
  - Spark, Elemental, …
- Design, implement, measure, model and/or compare performance
  - Can be missing entirely on target platform
  - May exist, but with a different programming framework