GPU TECHNOLOGY CONFERENCE

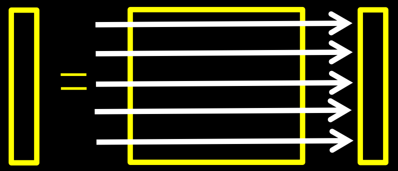
On the Parallel Solution of Sparse Triangular Linear Systems

M. Naumov*
San Jose, CA | May 16, 2012

*NVIDIA

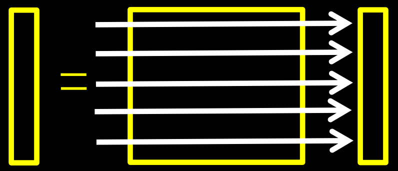
Why Is This Interesting?

- There exist different classes of parallel problems
  - ✓ Embarrassingly parallel
    - ✓ sparse matrix-vector multiplication
      (memory bound - address coalescing of memory accesses)

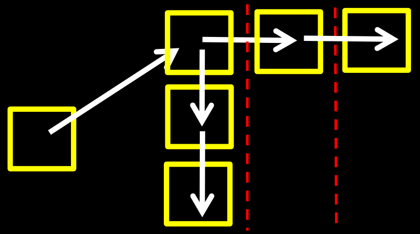


Why Is This Interesting?

- There exist different classes of parallel problems
  - ✓ Embarrassingly parallel
    - ✓ sparse matrix-vector multiplication
      (memory bound - address coalescing of memory accesses)

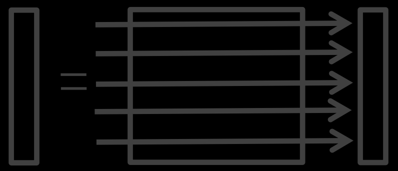


  - ✓ Dynamic Parallelism
    - ✓ graph traversals
      (parallelism bound - spawn work directly from the device)

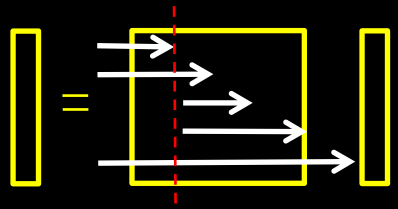


Why Is This Interesting?

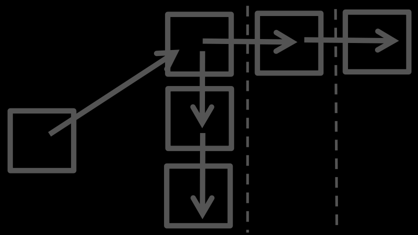
- There exist different classes of parallel problems
  - ✓ Embarrassingly parallel
    - ✓ sparse matrix-vector multiplication
      (memory bound - address coalescing of memory accesses)

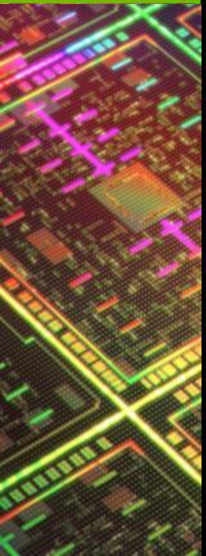


  - ✓ Static Parallelism
    - ✓ sparse triangular solve
      (parallelism bound - predict it ahead of time)



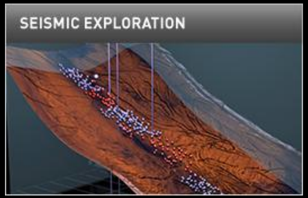
  - ✓ Dynamic Parallelism
    - ✓ graph traversals
      (parallelism bound - spawn work directly from the device)





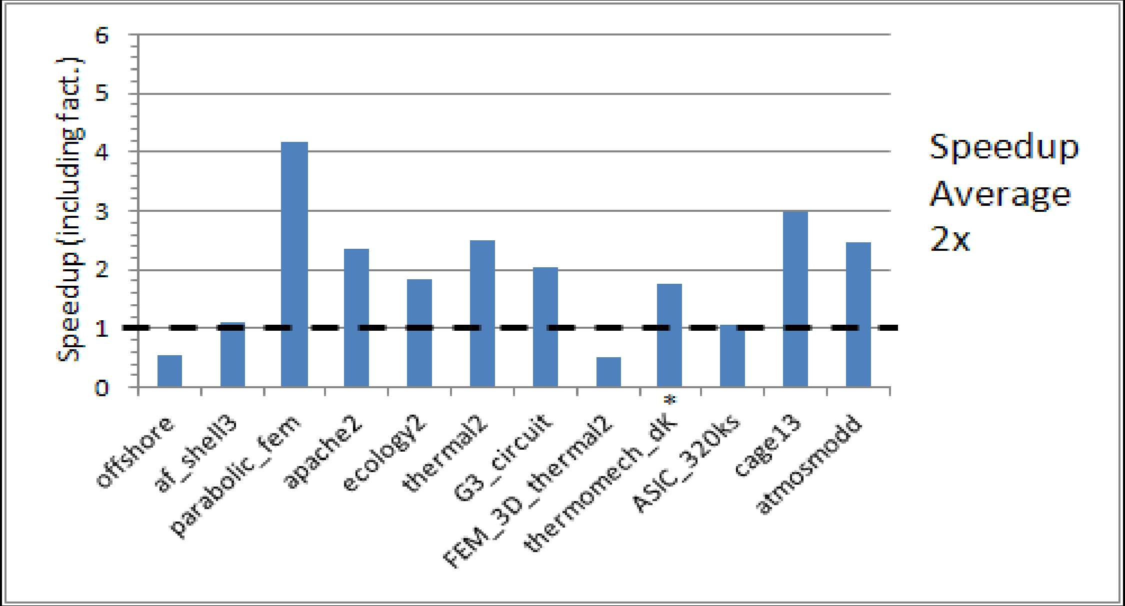
Applications

- Direct Methods
  - ✓ Solve resulting triangular factors (usually once)
- Preconditioned Iterative methods
  - ✓ Solve resulting triangular factors (multiple times)
- Incomplete-LU factorization (assuming 0 fill-in)
  - ✓ Has the same pattern of parallelism



GPU TECHNOLOGY CONFERENCE

Preconditioned Iterative Methods



- The performance depends on the sparsity pattern of the matrix
- We will come back to this result at the end of the presentation

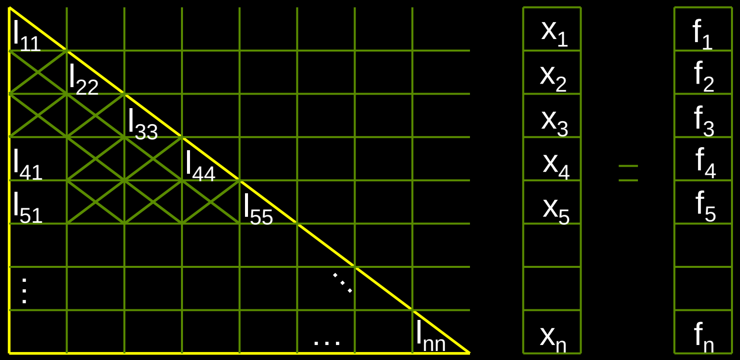
*: indicates that for this particular matrix the method did not converge to required tolerance.

NVIDIA C2050, ECC on
MKL 10.2.3 , Core™ i7 CPU 950 @ 3.07GHz

Sparse Triangular Solve

Introduction

- Problem description (sparse lower triangular solve)



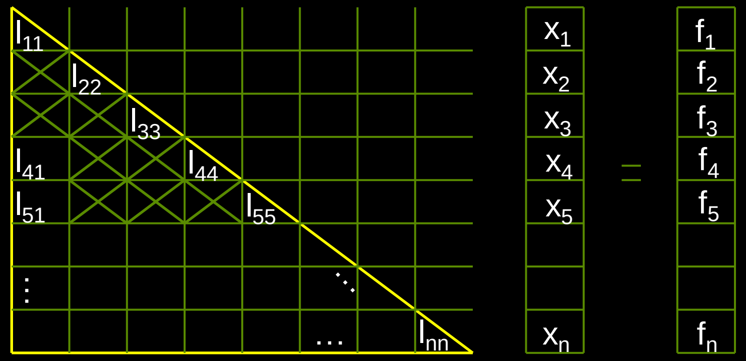
$x_1 = f_1 / l_{11}$

$x_2 = (f_2 – l_{21}x_1) / l_{22}$

$x_3 = (f_3 – l_{31}x_1 – l_{32}x_2) / l_{33}$

Introduction

- Problem description (sparse lower triangular solve)

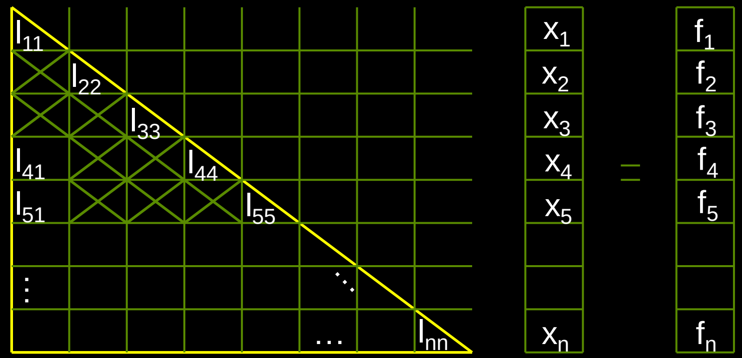


x_1

$$x_1 = f_1 / l_{11}$$
$$x_2 = (f_2 - l_{21}x_1) / l_{22}$$
$$x_3 = (f_3 - l_{31}x_1 - l_{32}x_2) / l_{33}$$
$$x_4 = (f_4 - l_{41}x_1 - l_{42}x_2 - l_{43}x_3) / l_{44}$$
$$x_5 = (f_5 - l_{51}x_1 - l_{52}x_2 - l_{53}x_3 - l_{54}x_4) / l_{55}$$

Introduction

- Problem description (sparse lower triangular solve)



x_1

$x_1 = f_1 / l_{11}$

$x_2 = (f_2 - l_{21}x_1) / l_{22}$

$x_3 = (f_3 - l_{31}x_1 - l_{32}x_2) / l_{33}$

$x_4 = (f_4 - l_{41}x_1 - l_{42}x_2 - l_{43}x_3) / l_{44}$

$x_5 = (f_5 - l_{51}x_1 - l_{52}x_2 - l_{53}x_3 - l_{54}x_4) / l_{55}$

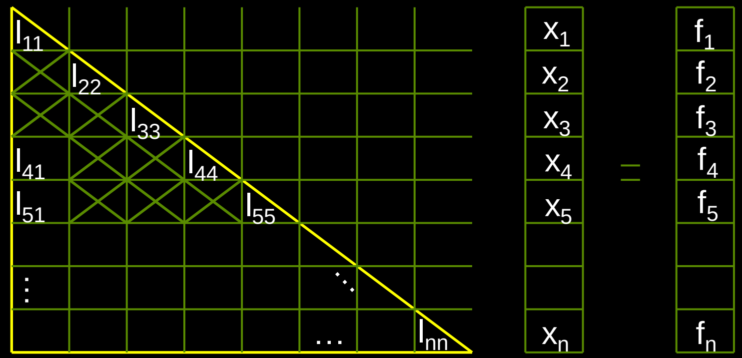
x_{n-1}

$\vdots$

$x_n = (f_n - l_{n1}x_1 - l_{n2}x_2 - l_{n3}x_3 \ldots - l_{n,n-1}x_{n-1}) / l_{nn}$

Introduction

- Available parallelism



$$x_1 = f_1 / l_{11}$$
$$x_2 = (f_2 - l_{21}x_1) / l_{22}$$
$$x_3 = (f_3 - l_{31}x_1 - l_{32}x_2) / l_{33}$$

level 1

$$x_4 = (f_4 - l_{41}x_1 - l_{42}x_2 - l_{43}x_3) / l_{44}$$
$$x_5 = (f_5 - l_{51}x_1 - l_{52}x_2 - l_{53}x_3 - l_{54}x_4) / l_{55}$$

level 2

$$\vdots$$
$$x_n = (f_n - l_{n1}x_1 - l_{n2}x_2 - l_{n3}x_3 \dots - l_{n,n-1}x_{n-1}) / l_{nn}$$

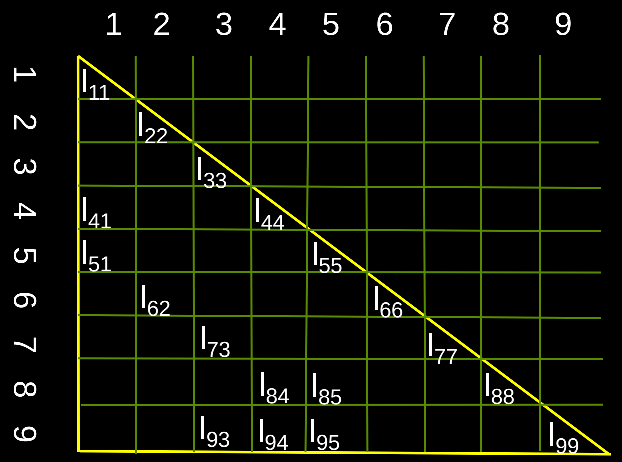
- Current step does not depend on all the previous steps
  - ✓ There is a dependency between levels
  - ✓ All rows within each level can be processed independently
  - ✓ Split the computation into an analysis and a solve phase

Analysis Phase

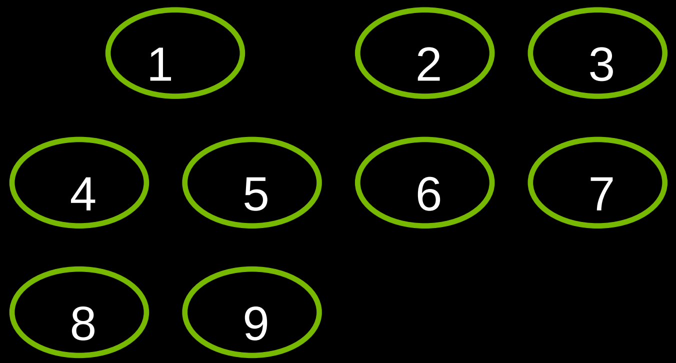
(shared between incomplete-LU/Cholesky and sparse triangular solve)

Analysis Phase: Example

matrix sparsity pattern



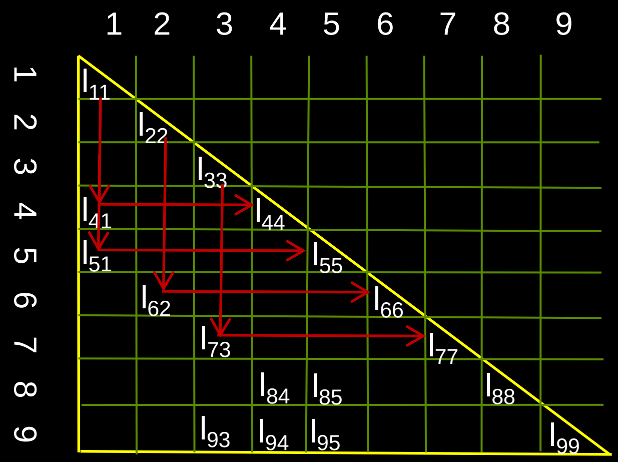
directed acyclic graph (DAG)



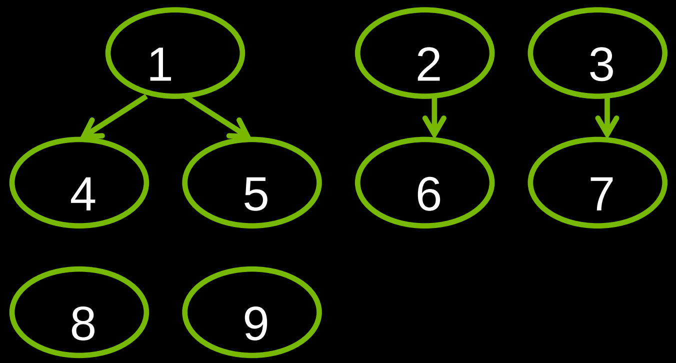
Level Ptr	1		
Level Index	1	2	3
Level/Depth	1		

Analysis Phase: Example

matrix sparsity pattern



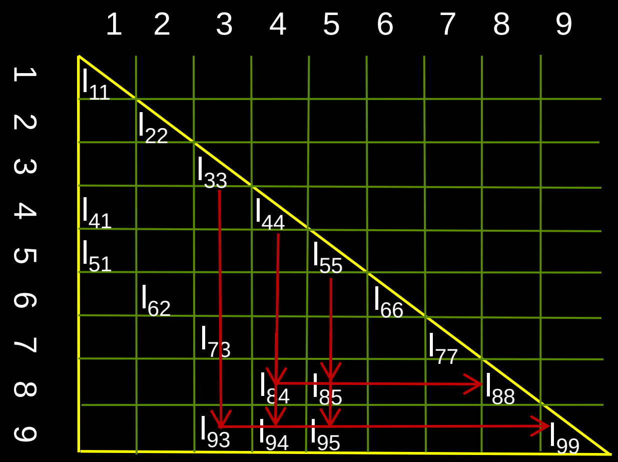
directed acyclic graph (DAG)



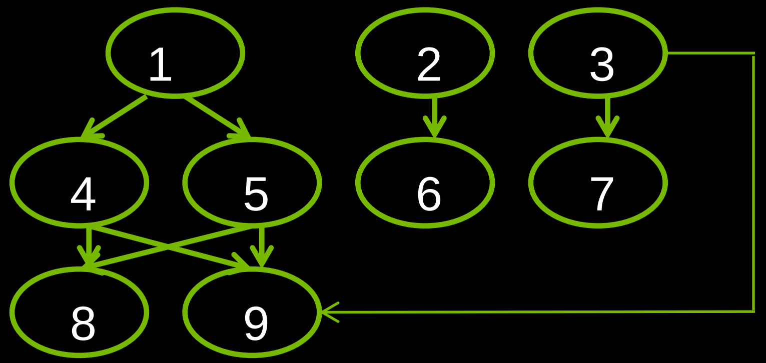
Level Ptr	1			4			
Level Index	1	2	3	4	5	6	7
Level/Depth	1			2			

Analysis Phase: Example

matrix sparsity pattern



directed acyclic graph (DAG)

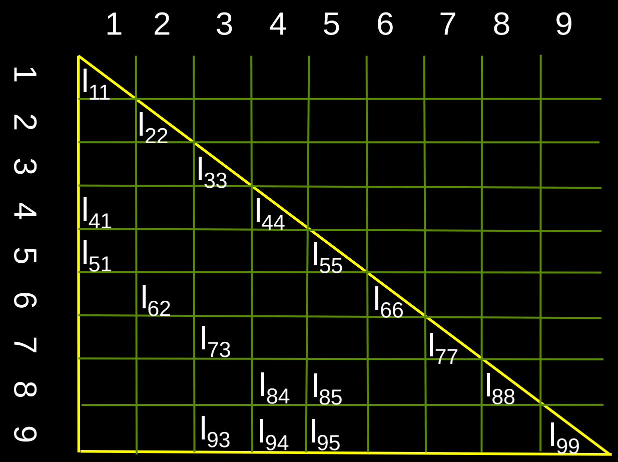


Level Ptr	1 4 8		
Level Index	1 2 3	4 5 6 7	8 9
Level/Depth	1	2	3

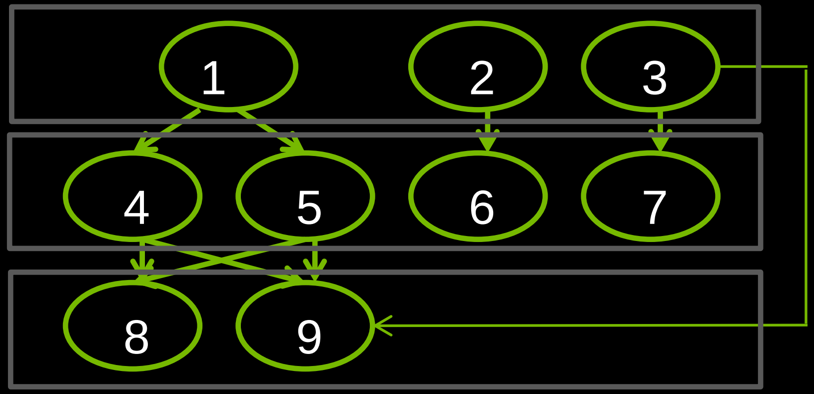
GPU TECHNOLOGY CONFERENCE

Analysis Phase: Example

matrix sparsity pattern



directed acyclic graph (DAG)



Level Ptr	1 4 8 10		
Level Index	1 2 3	4 5 6 7	8 9
Level/Depth	1	2	3

Dependency DAG

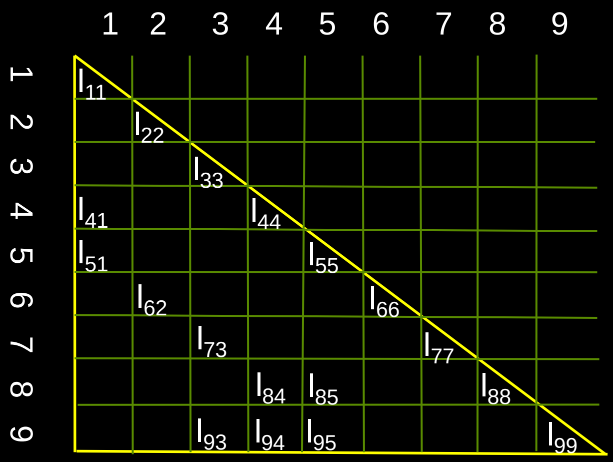
- How to construct the DAG
  - ✓ It is actually implicit in the sparsity pattern of matrix
  - ✓ Need easy access to elements stored in each column (CSC format)
- How to order nodes in the DAG
  - ✓ Topological Sort, using Depth-First-Search (DFS)
    - Hard to implement, parallel computational complexity* $O(\log^2 n)$
  - ✓ Level-by-level
    - Easy to implement, parallel computational complexity $O(n)$
- How to process the information in the DAG
  - ✓ Use separate solve/numerical factorization phase

*: if $O(n^{2.81} / \log n)$ processors are used.

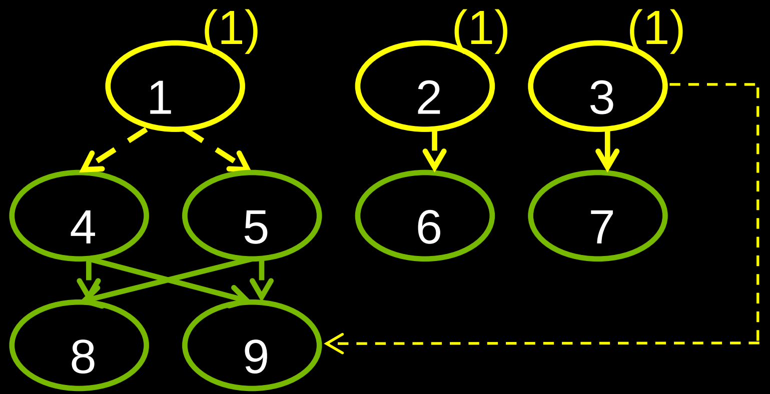
Level-by-level

- Do while there are no roots
  - ✓ Find roots
  - ✓ Remove the dependency from their children

matrix sparsity pattern



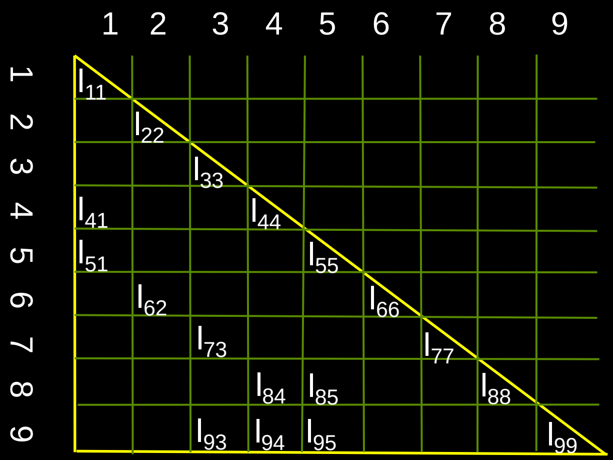
directed acyclic graph (DAG)



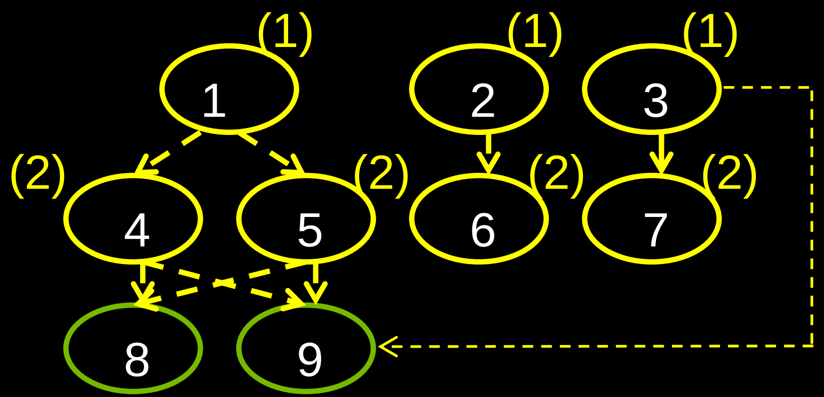
Level-by-level

- Do while there are no roots
  - ✓ Find roots
  - ✓ Remove the dependency from their children

matrix sparsity pattern



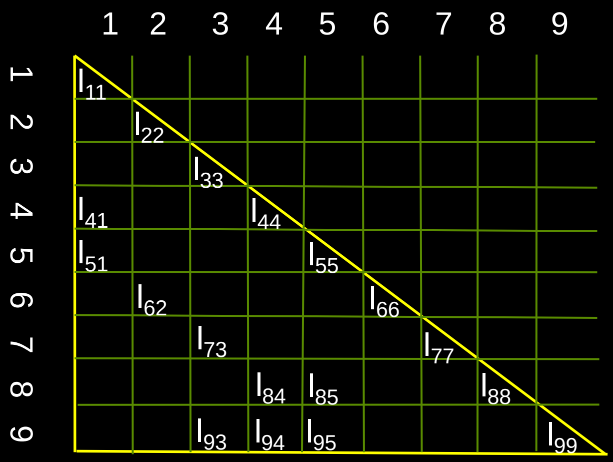
directed acyclic graph (DAG)



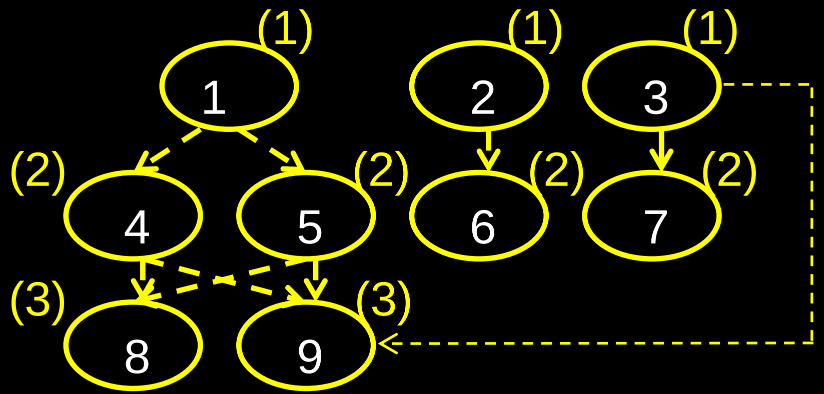
Level-by-level

- Do while there are no roots
  - ✓ Find roots
  - ✓ Remove the dependency from their children

matrix sparsity pattern



directed acyclic graph (DAG)

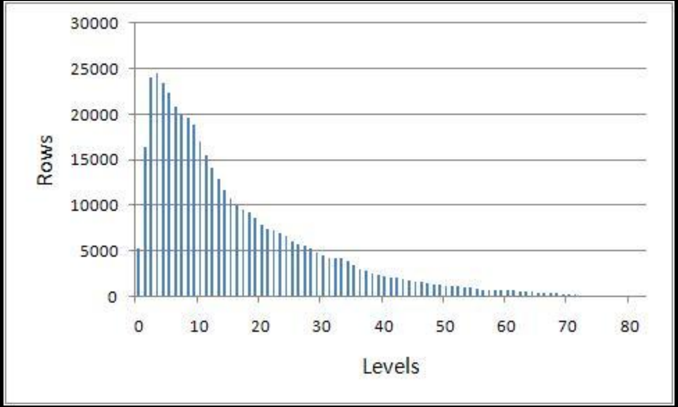


Level Ptr	1			4				8	10
Level Index	1	2	3	4	5	6	7	8	9
Level/Depth	1			2				3	

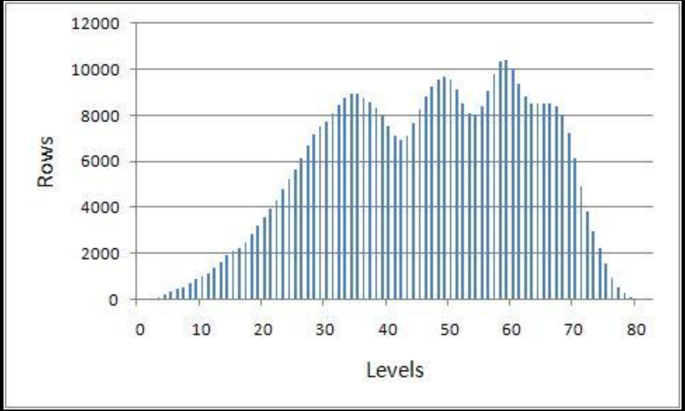
sort nodes based on depth

Distribution of Rows into Levels

matrix cage13 lower L



matrix cage13 upper U



Solve and Numerical Factorization Phase

(same framework, but different implementation)

Input

- Assume that analysis has been done, let
  - ✓ Li denote i-th level
  - ✓ Rj denote j-th row
- Then

	1	2	3	4	5	6	7	8	9
1	l_{11}								
2		l_{22}							
3			l_{33}						
4	l_{41}			l_{44}					
5	l_{51}				l_{55}				
6		l_{62}				l_{66}			
7			l_{73}				l_{77}		
8				l_{84}	l_{85}			l_{88}	
9			l_{93}	l_{94}	l_{95}				l_{99}

Distribution of rows into levels

L1: R1, R2, R3
L2: R4, R5, R6, R7
L3: R8, R9

In general, rows are not necessarily processed in order, so we might lose some coalescing.

Solve Phase

We need synchronization between levels

- ➢ Use a single-block kernel that loops through the levels (on GPU)
  - ✓ Advantage: lightweight synchronization

single thread block →

time ↓

K1

L1: R1, R2, R3

- __syncthreads()

L2: R4, R5, R6, R7

- __syncthreads()

L3: R8, R9

Solve Phase

We need synchronization between levels

- ➢ Use a single-block kernel that loops through the levels (on GPU)
  - ✓ Advantage: lightweight synchronization

single thread block →

time ↓

K1
L1: R1, R2, R3
- __syncthreads()
L2: R4, R5, R6, R7
- __syncthreads()
L3: R8, R9

- ➢ Use a multi-block kernels that are launched in a loop (on CPU)
  - ✓ Advantage: plenty of parallelism

multiple thread blocks →

time ↓

K1 L1: R1, R2, R3
- implicit synchronization
K2 L2: R4, R5, R6, R7
- implicit synchronization
K3 L3: R8, R9

Solve Phase

Can we combine both approaches?

Solve Phase

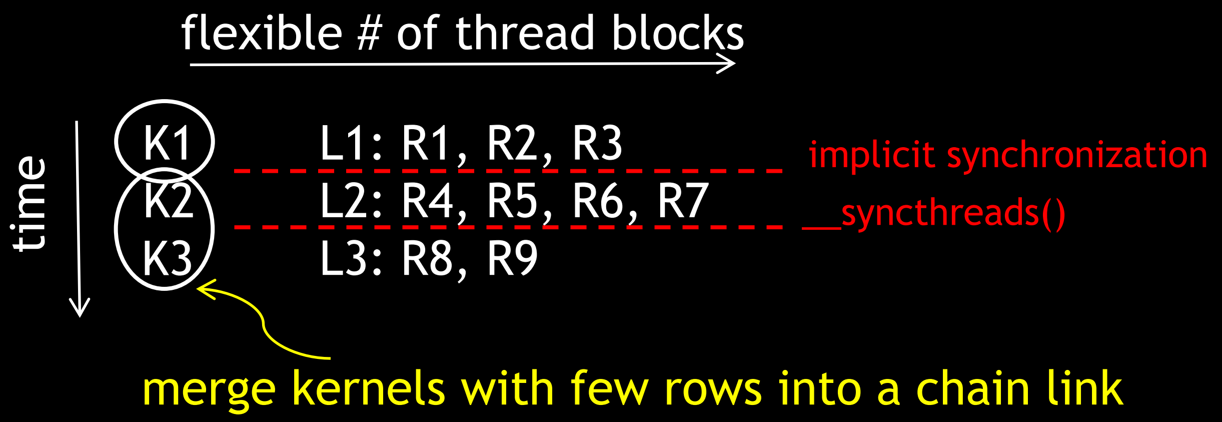
Can we combine both approaches?

- Merge consecutive single-block kernel launches into a chain link
  - ✓ Process a single chain link with a single kernel call
  - ✓ Process different chain links with a loop on the host
- Advantages
  - ✓ less kernel launches and less information on the host
  - ✓ lightweight synchronization with __syncthreads()

Solve Phase

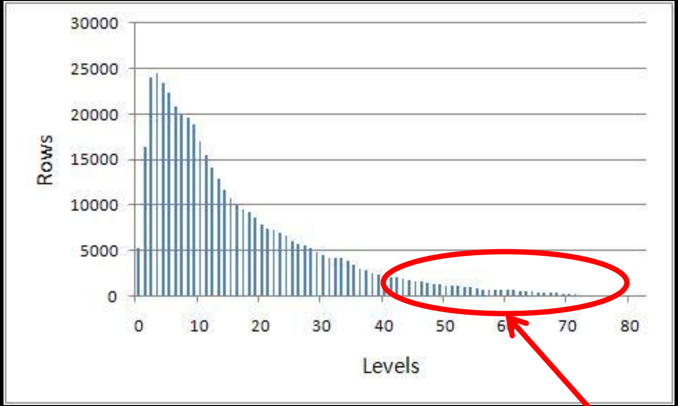
Can we combine both approaches?

- Merge consecutive single-block kernel launches into a chain link
  - ✓ Process a single chain link with a single kernel call
  - ✓ Process different chain links with a loop on the host
- Advantages
  - ✓ less kernel launches and less information on the host
  - ✓ lightweight synchronization with __syncthreads()

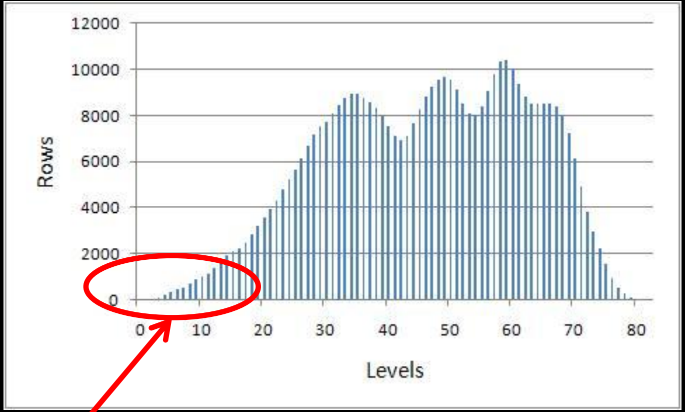


Distribution of Rows into Levels

matrix cage13 lower L



matrix cage13 upper U



(potentially) single kernel launch

Numerical Experiments

Conjugate Gradient

```
compute incomplete-LU/Cholesky M
for i=1:maxit
    Solve M z = r

    rho = r^T*z
    if (i = 1)
       p=z
    else
        beta = rho /rho^(i-1)
        p = z + p*beta
    end
    Compute q = A*p

    alpha = rho/(p^T*q)
    x = x + p*alpha
    r = r – q*alpha

    if (||r||_2 / ||r^(0)||_2  < tol) stop
end
```

csrsv_analysis – analysis phase

csrilu0/csric0 – numerical factorization phase

csrsv_solve – solve phase

csrmv – matrix-vector multiplication

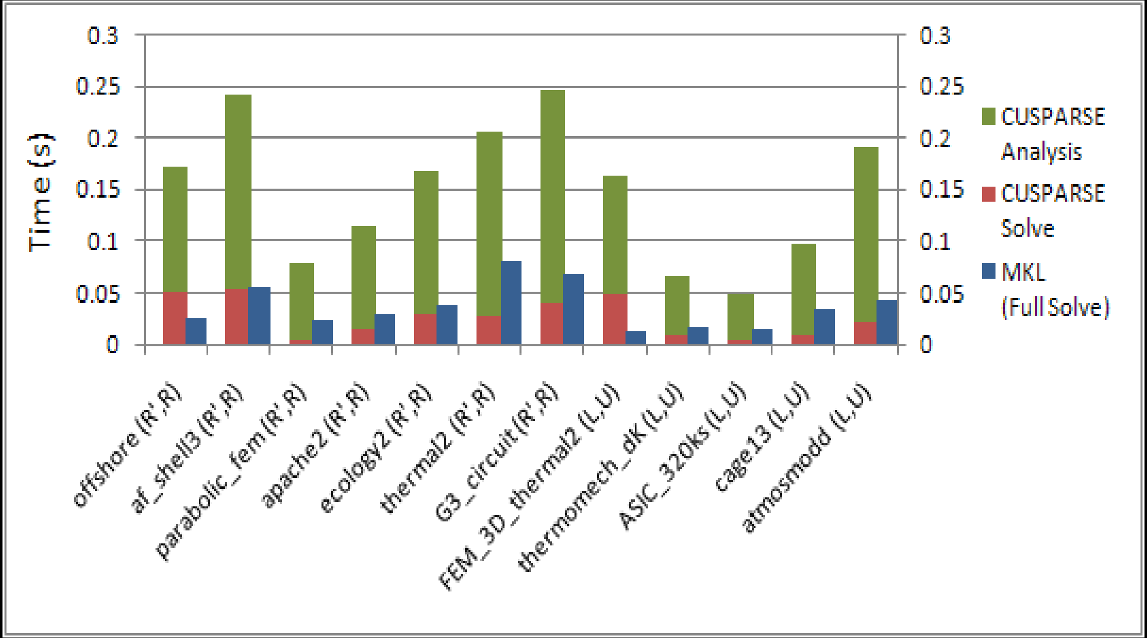
*: where superscript $^{(i)}$ indicates that the quantity is taken from the i-th iteration

Matrices

Matrix	n	nnz	s.p.d.	Application
offshore	259,789	4,242,673	yes	Geophysics
af_shell3	504,855	17,562,051	yes	Mechanics
parabolic_fem	525,825	3,674,625	yes	General
apache2	715,176	4,817,870	yes	Mechanics
ecology2	999,999	4,995,991	yes	Biology
thermal2	1,228,045	8,580,313	yes	Thermal Simulation
G3_Circuit	1,585,478	7,660,826	yes	Circuit Simulation
FEM_3D_thermal2	147,900	3,489,300	no	Mechanics
thermomech_dK	204,316	2,846,228	no	Mechanics
ASIC_320ks	321,671	1,316,085	no	Circuit Simulation
cage13	445,315	7,479,343	no	Biology
atmosmodd	1,270,432	8,814,880	no	Atmospheric Model.

- In our numerical experiments
  - ✓ Matrices are selected from The University of Florida Sparse Matrix Collection
  - ✓ Right-hand-side $f=A*e$, where $e=[1,\dots,1]^T$
  - ✓ Stopping criteria is based on
    maximum # of iterations 2000 and relative residual $||r||_2 / ||r_0||_2 < 10^{-7}$

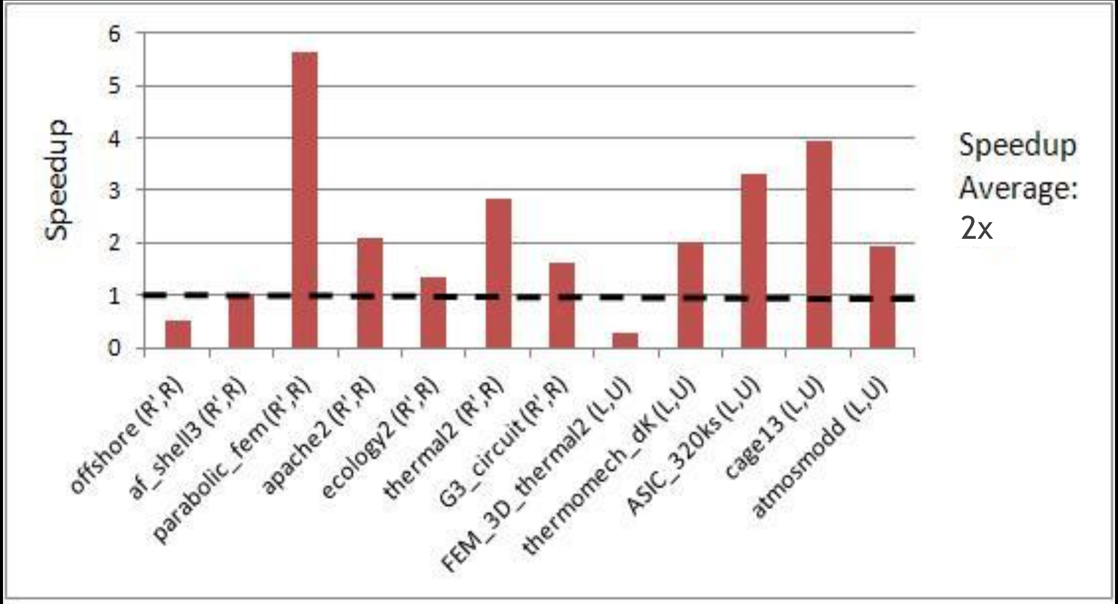
Time of the Analysis and Solve Phases
(sparse triangular solve)



- Notice that in an iterative method
  - ✓ The more expensive analysis phase is performed once
  - ✓ The faster solve phase is performed multiple times (at least once at every iteration of the iterative method)

NVIDIA C2050, ECC on
MKL 10.2.3 , Core^{TM} i7 CPU 950 @ 3.07GHz

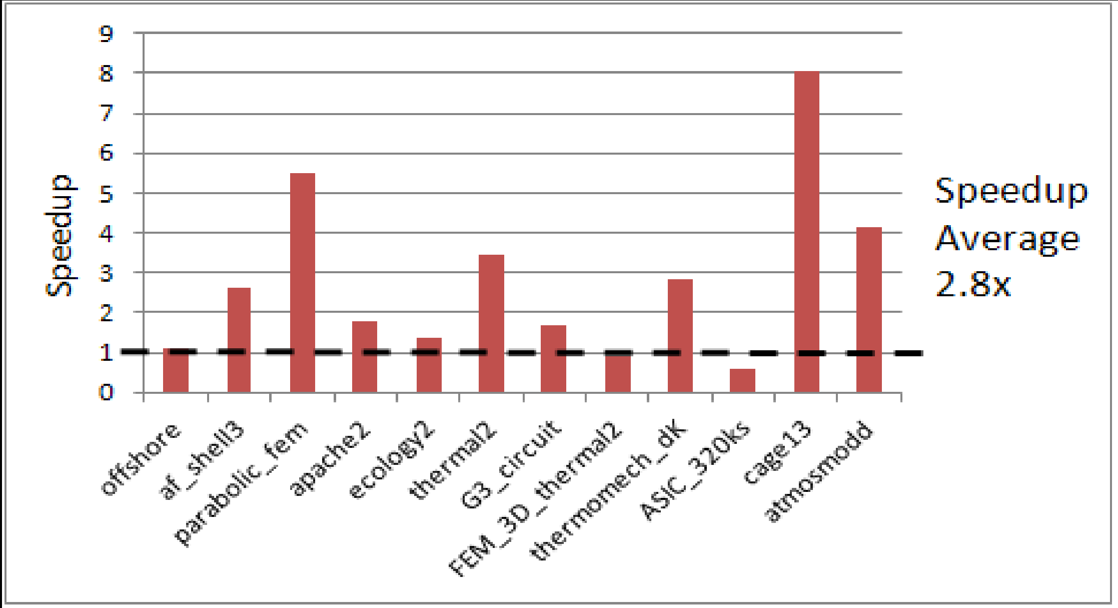
Speedup of the Solve Phase
(sparse triangular solve)



- The performance depends on the sparsity pattern of the matrix
- Usually, we benefit if there are
  - ✓ few dependencies between rows (more available parallelism)
  - ✓ higher number of elements per row (more work to do)

NVIDIA C2050, ECC on
MKL 10.2.3 , Core™ i7 CPU 950 @ 3.07GHz

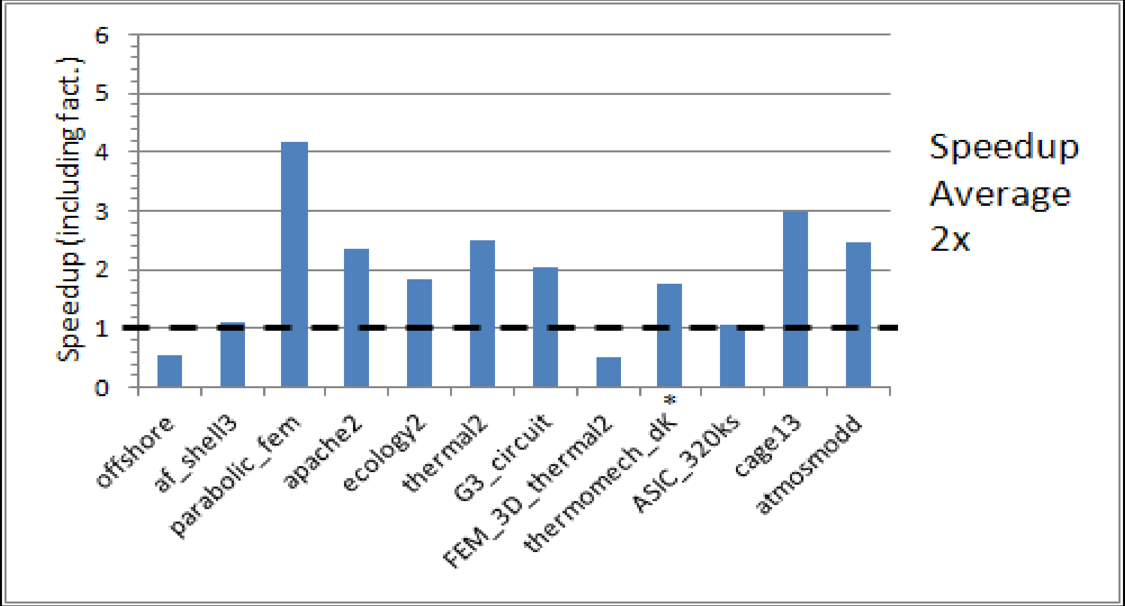
Speedup of the Factorization Phase
(incomplete-LU/Cholesky with 0 fill-in)



- ➢ The performance depends on the sparsity pattern of the matrix
- ➢ Usually, we benefit if there are
  - ✓ few dependencies between rows (more available parallelism)
  - ✓ higher number of elements per row (more work to do)

NVIDIA C2050, ECC on
MKL 10.2.3 , Core™ i7 CPU 950 @ 3.07GHz

Speedup of the Preconditioned CG and BiCGStab



- The performance depends on the sparsity pattern of the matrix
- In our numerical experiments,
  - ✓ We have on average outperformed the MKL implementation by 2x
  - ✓ The MKL settings were set to allow it to use all 4 CPU cores

*: indicates that for this particular matrix the method did not converge to required tolerance.

NVIDIA C2050, ECC on
MKL 10.2.3 , Core^{TM} i7 CPU 950 @ 3.07GHz

Conclusions and Future Work

- Sparse triangular solve
    - ✓ Illustrates how to approach problems with static parallelism
    - ✓ In our numerical experiments, obtained on average 2x speedup

Conclusions and Future Work

- Sparse triangular solve
  - ✓ Illustrates how to approach problems with static parallelism
  - ✓ In our numerical experiments, obtained on average 2x speedup

- Look at explicit and implicit reordering techniques
  - ✓ Solve $(P^T A Q)(Q^T x) = P^T f$,
    where P and Q are permutation matrices
  - ✓ May affect parallelism and convergence

Conclusions and Future Work

- Sparse triangular solve
  - ✓ Illustrates how to approach problems with static parallelism
  - ✓ In our numerical experiments, obtained on average 2x speedup
- Look at explicit and implicit reordering techniques
  - ✓ Solve $(P^T A Q)(Q^T x) = P^T f$, where P and Q are permutation matrices
  - ✓ May affect parallelism and convergence
- Block-iterative methods
  - ✓ Handle multiple right-hand-sides
  - ✓ May converge in less iterations

GPU TECHNOLOGY CONFERENCE

Thank You!

[1] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU", NVIDIA Technical Report, NVR-2011-001, 2011.
http://research.nvidia.com/publication/parallel-solution-sparse-triangular-linear-systems-preconditioned-iterative-methods-gpu

[2] M. Naumov, "Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU", NVIDIA Technical Report, NVR-2012-003, 2012.
http://research.nvidia.com/publication/incomplete-lu-and-cholesky-factorization-preconditioned-iterative-methods-gpu

[3] NVIDIA CUSPARSE and CUBLAS Libraries,
http://developer.nvidia.com/cuda-downloads

Sparse triangular solve and Incomplete-LU/Cholesky factorization
are part of the CUSPARSE 5.0 library