Blocked algorithms vs. Algorithms by blocks

Paolo Bientinesi

pauldj@cs.umu.se

June 05, 2023

TU Delft

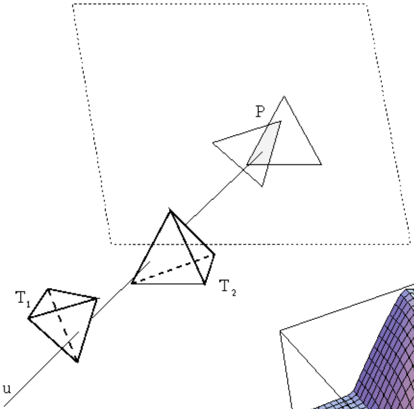
UMEÅ UNIVERSITY

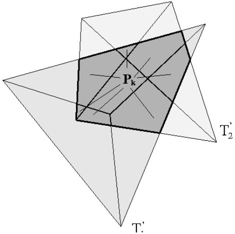
About me

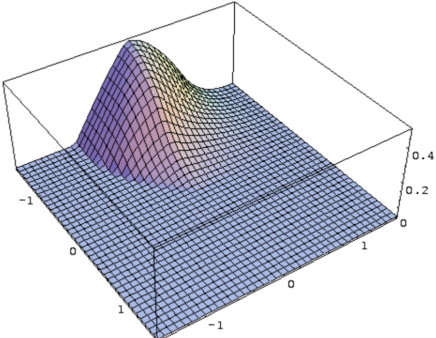
Italy – Tuscany

Italy – University of Pisa

Computational Geometry







USA – UT Austin, TX

- Symmetric eigenproblem
  $AX = X\Lambda$
- FLAME project
  Automatic generation of algorithms

Algorithm $LU = A$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$

where $A_{TL}, L_{TL}, U_{TL},$ are 0×0

While $m(A_{TL}) \leq m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right),$$

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

where $\alpha_{11}, 1, \upsilon_{11}$ are scalars

$\upsilon_{11} := \alpha_{11} - l_{10}^T u_{01}$
$u_{12}^T := a_{12}^T - l_{10}^T U_{02}$
$l_{21} := (a_{21} - L_{20} u_{01}) / \upsilon_{11}$

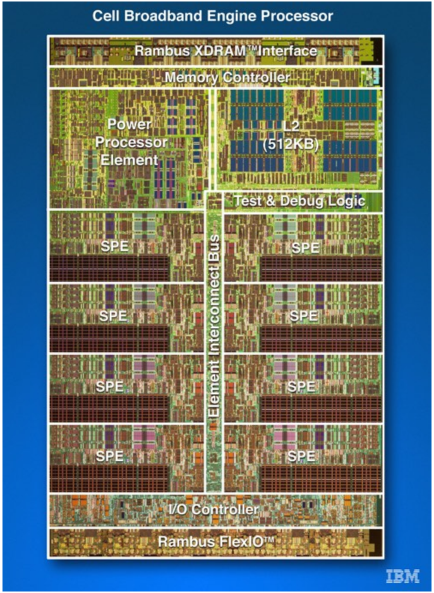
Continue with

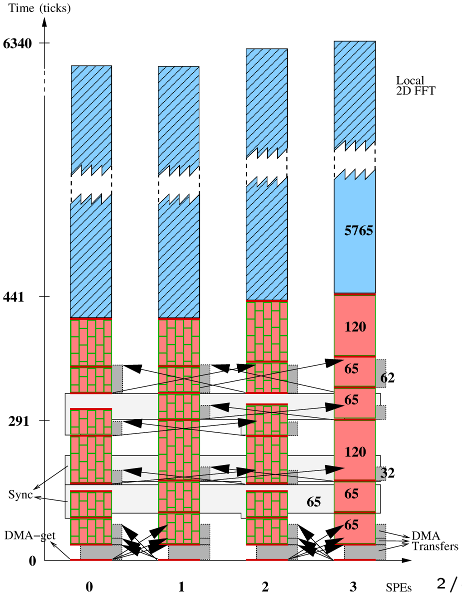
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \cdots$$

endwhile

USA – Duke, NC

- Cell
- FFTs





Germany – RWTH Aachen University

High-Performance &
Automatic Computing

Modern
Inverse
Problems

Sweden – Umeå University

- Matrix & tensor operations
- HPC
- Computer music

HPC2N

High-Performance Computing Center North

Today's outline

- Part 1: HPC & LA fundamentals
- Part 2: Unblocked vs. blocked algorithms
- Part 3: Algorithms by blocks

Fundamentals of HPC & LA

Cholesky: unblocked and blocked algorithms

Cholesky: algorithms by blocks

World of high-performance numerical linear algebra

Where are we?

- Dense vs. sparse
- Linear solvers vs. eigensolvers
- Direct methods vs. iterative methods
- (Shared memory vs. distributed memory)
- (Small vs. medium/large size)
- ...

World of high-performance numerical linear algebra

Where are we?

- **Dense** vs. sparse
- **Linear solvers** vs. eigensolvers
- **Direct methods** vs. iterative methods
- (**Shared memory** vs. distributed memory)
- (Small vs. medium/large size)
- ...

Context: Development of high-performance linear algebra libraries

"Standard" approach (since the '70s):

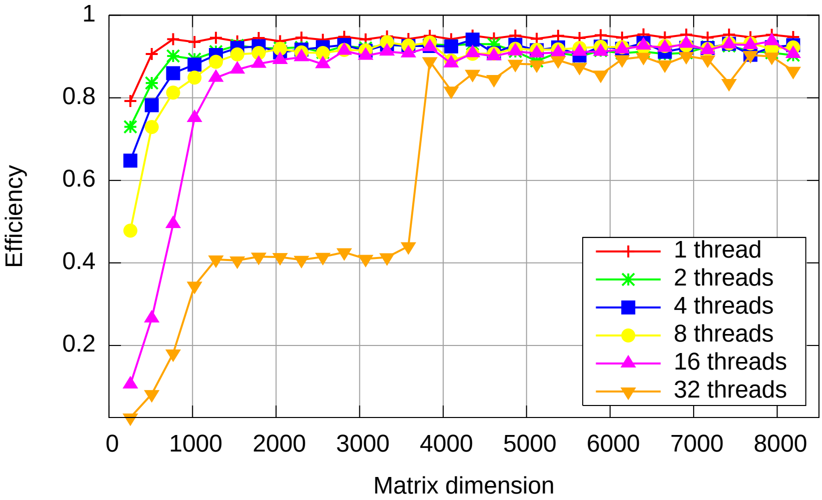
1. Identify building blocks ("kernels")
   e.g., inner product, matrix-vector & matrix-matrix product, linear system, eigenproblem, ...

2. Encapsulate into a function
   e.g., `ddot`, `gemv`, `gemm`, `dgesv`, `dsyevd`, ...

3. Optimize, specialize

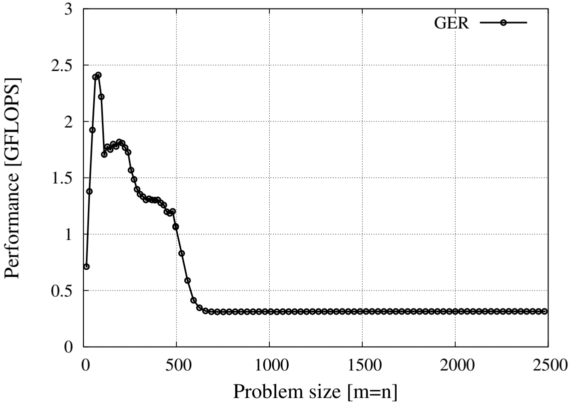
Reference: DGEMM – “speed of light”

Efficiency: Percentage of theoretical peak performance



GER: $A := A + \alpha x y^T$

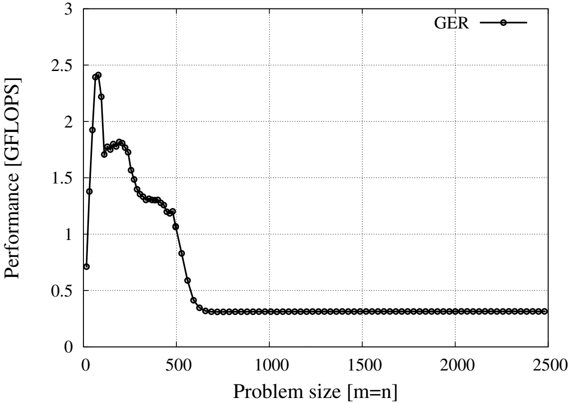
FLOPS = # flops/sec



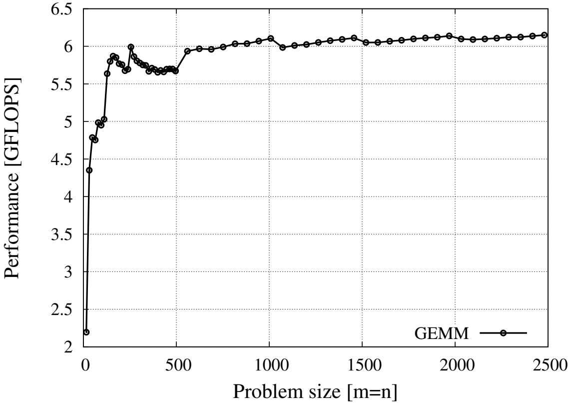
Can you explain these profiles?

GER: $A := A + \alpha x y^T$

FLOPS = # flops/sec



GEMM: $C := \alpha A * B + \beta C$

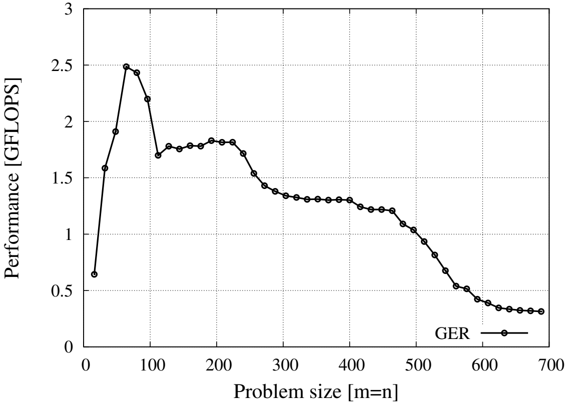


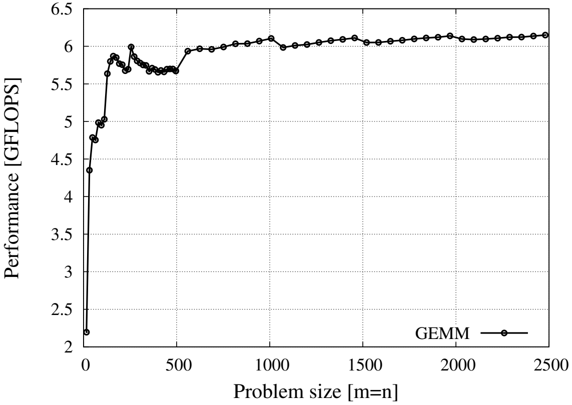
Can you explain these profiles?

GER: $A := A + \alpha x y^T$

FLOPS = # flops/sec

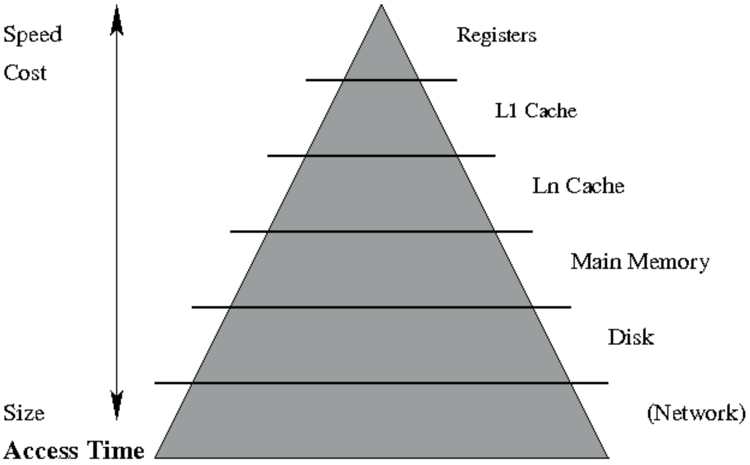
GEMM: $C := \alpha A * B + \beta C$



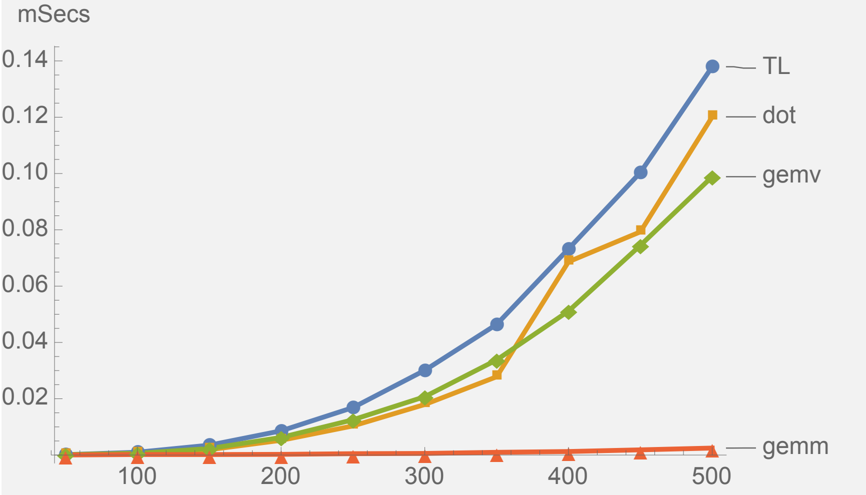


Can you explain these profiles?

Memory hierarchy ≡ not all flops cost the same!



Many ways to skin a DGEMM



Can you explain the difference in execution time?

Temporal & spatial locality

Also "Locality of references", "Principle of locality"

Temporal & spatial locality

Also "Locality of references", "Principle of locality"

Observations

- Programs exhibit temporal locality

Consequences

Temporal & spatial locality

Also “Locality of references”, “Principle of locality”

Observations

- Programs exhibit temporal locality

Consequences

- Cache memories

Temporal & spatial locality

Also “Locality of references”, “Principle of locality”

Observations

- Programs exhibit temporal locality
- Programs exhibit spatial locality

Consequences

- Cache memories

Temporal & spatial locality

Also "Locality of references", "Principle of locality"

Observations

- Programs exhibit temporal locality
- Programs exhibit spatial locality

Consequences

- Cache memories
- Cachelines & Prefetching

Computational intensity

Also "operational intensity", "arithmetic intensity"

Computational intensity

Also "operational intensity", "arithmetic intensity"

BLAS-1: $y := \alpha x + y$ $\qquad x, y \in \mathbb{R}^n$

$\gamma := \alpha + x^T y$

	#FLOPS	Mem. refs.	Ratio
BLAS-1			

Computational intensity

Also "operational intensity", "arithmetic intensity"

BLAS-1: $y := \alpha x + y$ $\qquad x, y \in \mathbb{R}^n$

$\gamma := \alpha + x^T y$

	#FLOPS	Mem. refs.	Ratio
BLAS-1	$2n$		

Computational intensity

Also "operational intensity", "arithmetic intensity"

BLAS-1: $y := \alpha x + y$ $\quad x, y \in \mathbb{R}^n$

$\gamma := \alpha + x^T y$

	#FLOPS	Mem. refs.	Ratio
BLAS-1	$2n$	$3n$	$2/3$

Computational intensity

Also "operational intensity", "arithmetic intensity"

BLAS-1: $y := \alpha x + y$ $\qquad x, y \in \mathbb{R}^n$

$\gamma := \alpha + x^T y$

BLAS-2: $y := Ax + y$ $\qquad A, L \in R^{n\times n},\ x, y \in R^n$

$y := L^{-1} x$

	#FLOPS	Mem. refs.	Ratio
BLAS-1	$2n$	$3n$	$2/3$
BLAS-2			

Computational intensity

Also "operational intensity", "arithmetic intensity"

BLAS-1: $y := \alpha x + y$ $\qquad x, y \in \mathbb{R}^n$

$\gamma := \alpha + x^T y$

BLAS-2: $y := Ax + y$ $\qquad A, L \in R^{n \times n}, \; x, y \in R^n$

$y := L^{-1} x$

	#FLOPS	Mem. refs.	Ratio
BLAS-1	$2n$	$3n$	$2/3$
BLAS-2	$2n^2$		

Computational intensity

Also “operational intensity”, “arithmetic intensity”

BLAS-1: $y := \alpha x + y$ $\quad x, y \in \mathbb{R}^n$

$\gamma := \alpha + x^T y$

BLAS-2: $y := Ax + y$ $\quad A, L \in R^{n\times n},\ x, y \in R^n$

$y := L^{-1}x$

	#FLOPS	Mem. refs.	Ratio
BLAS-1	$2n$	$3n$	$2/3$
BLAS-2	$2n^2$	n^2	2

Computational intensity

Also "operational intensity", "arithmetic intensity"

BLAS-1:	$y := \alpha x + y$ $\gamma := \alpha + x^T y$	$x, y \in \mathbb{R}^n$
BLAS-2:	$y := Ax + y$ $y := L^{-1}x$	$A, L \in R^{n\times n},\ x, y \in R^n$
BLAS-3:	$C := AB + C$ $C := L^{-1}B$	$A, B, C, L \in R^{n\times n}$

	`#FLOPS`	Mem. refs.	Ratio
BLAS-1	$2n$	$3n$	$2/3$
BLAS-2	$2n^2$	n^2	2
BLAS-3			

Computational intensity

Also "operational intensity", "arithmetic intensity"

BLAS-1:	$y := \alpha x + y$ $\gamma := \alpha + x^T y$	$x, y \in \mathbb{R}^n$
BLAS-2:	$y := Ax + y$ $y := L^{-1}x$	$A, L \in R^{n\times n},\ x, y \in R^n$
BLAS-3:	$C := AB + C$ $C := L^{-1}B$	$A, B, C, L \in R^{n\times n}$

	`#FLOPS`	Mem. refs.	Ratio
BLAS-1	$2n$	$3n$	$2/3$
BLAS-2	$2n^2$	n^2	2
BLAS-3	$2n^3$	$4n^2$	$\mathbf{n/2}$

Summary: Libraries

1970s

- Identification, analysis, optimization of **building blocks**

Summary: Libraries

1970s

- Identification, analysis, optimization of **building blocks**
- Computers difficult to program
  BUT
  Programs "easy" to optimize
- $\textbf{Cost}(\mathcal{A}lg) \;\equiv\; \textbf{\#operations}(\mathcal{A}lg)$

Summary: Libraries

1970s

- Identification, analysis, optimization of **building blocks**
- Computers difficult to program
  BUT
  Programs "easy" to optimize
- **Cost**$(\mathcal{A}lg) \equiv$ **#operations**$(\mathcal{A}lg)$
- Libraries: convenience, portability, separation of concerns, confidence, performance

Summary: Libraries

1970s

- Identification, analysis, optimization of **building blocks**
- Computers difficult to program
  BUT
  Programs "easy" to optimize
- $\textbf{Cost}(\mathcal{A}lg) \equiv \textbf{\#operations}(\mathcal{A}lg)$
- Libraries: convenience, portability, separation of concerns, confidence, performance

since 1980s

- Increasingly complex HW:
  E.g., vector processors, memory hierarchies, prefetching, ...

Summary: Libraries

1970s

- Identification, analysis, optimization of **building blocks**
- Computers difficult to program
  BUT
  Programs "easy" to optimize
- **Cost**$(\mathcal{A}lg) \equiv$ **#operations**$(\mathcal{A}lg)$
- Libraries: convenience, portability, separation of concerns, confidence, performance

since 1980s

- Increasingly complex HW:
  E.g., vector processors, memory hierarchies, prefetching, ...
- Computers easier to program
  BUT
  Programs difficult to optimize
- **Cost**$(\mathcal{A}lg) \not\equiv$ **#operations**$(\mathcal{A}lg)$

Summary: Libraries

1970s

- Identification, analysis, optimization of **building blocks**
- Computers difficult to program
  BUT
  Programs "easy" to optimize
- **Cost**($\mathcal{A}lg$) $\equiv$ **#operations**($\mathcal{A}lg$)
- Libraries: convenience, portability, separation of concerns, confidence, performance

since 1980s

- Increasingly complex HW:
  E.g., vector processors, memory hierarchies, prefetching, ...
- Computers easier to program
  BUT
  Programs difficult to optimize
- **Cost**($\mathcal{A}lg$) $\not\equiv$ **#operations**($\mathcal{A}lg$)
- Libraries: as before +
  necessity for performance

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

$$\left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c|c} L_{TL}^T & L_{BL}^T \\ \hline & L_{BR}^T \end{array}\right) = \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array} \right) = ?$$

$$\left(\begin{array}{c|c} L_{TL}L_{TL}^T = A_{TL} & \\ \hline L_{BL}L_{TL}^T = A_{BL} & L_{BL}L_{BL}^T + L_{BR}L_{BR}^T = A_{BR} \end{array} \right)$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Partitioned Matrix Expression (PME):

$$\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & \\ \hline L_{BL} = A_{BL}\, L_{TL}^{-T} & L_{BR} = \Gamma\big(A_{BR} - L_{BL}L_{BL}^T\big) \end{array}\right)$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Operations:

$$\left(\begin{array}{c|c} \text{1) } L_{TL} = \mathrm{CHOL} & \\ \hline \text{2) } L_{BL} = \mathrm{TRSM} & \text{3) } L_{BR} = \mathrm{CHOL(SYRK)} \end{array}\right)$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

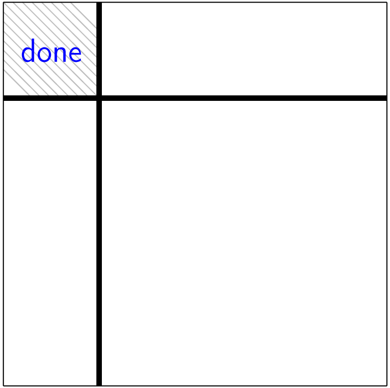
$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Dependencies:

$$\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & \\ \hline L_{BL} = A_{BL}\, L_{TL}^{-T} & L_{BR} = \Gamma\left(A_{BR} - L_{BL} L_{BL}^{T}\right) \end{array}\right)$$

Algorithm #1

Iteration i: completed



Algorithm #1

State of the matrix:

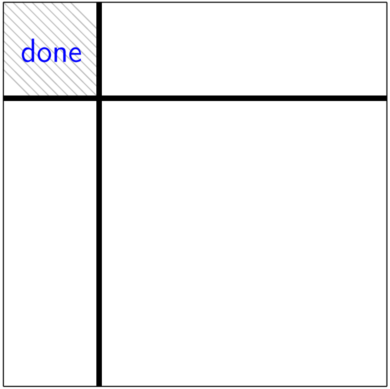
$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline & \end{array}\right)$$

Final state:

$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline L_{BL} = \mathrm{TRSM} & L_{BR} = \mathrm{CHOL(SYRK)} \end{array}\right)$$

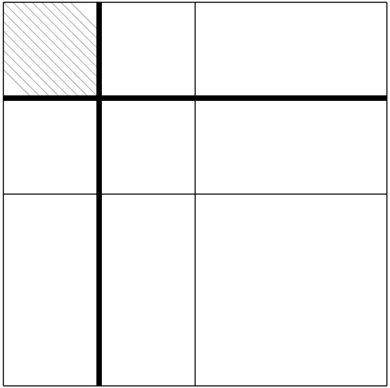
Algorithm #1

Iteration i: completed



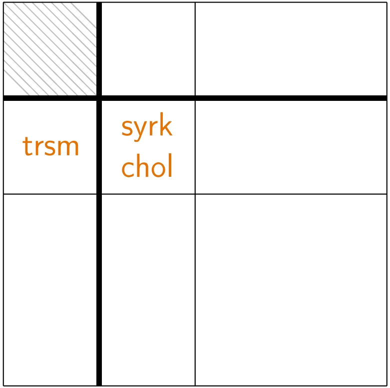
Algorithm #1

Iteration i+1: repartitioning. Blocked vs. unblocked!



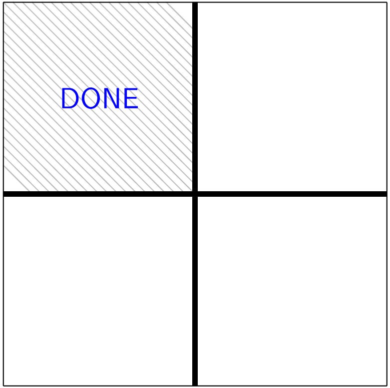
Algorithm #1

Iteration i+1: computation



Algorithm #1

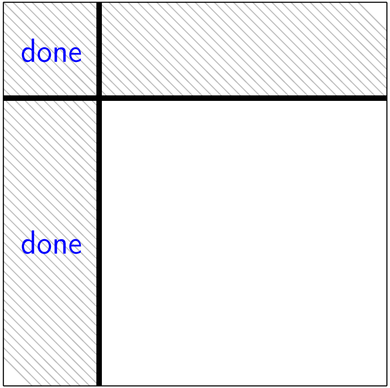
Iteration i+1: completed (boundary shift)



A Different Algorithm?

Algorithm #2

Iteration i: completed



Algorithm #2

State of the matrix:

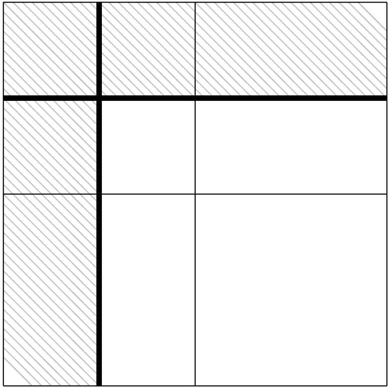
$$\left(\begin{array}{c|c} L_{TL} = \text{CHOL} & \\ \hline L_{BL} = \text{TRSM} & \end{array}\right)$$

Final State:

$$\left(\begin{array}{c|c} L_{TL} = \text{CHOL} & \\ \hline L_{BL} = \text{TRSM} & L_{BR} = \text{CHOL(SYRK)} \end{array}\right)$$

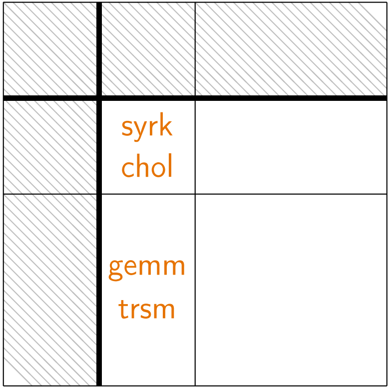
Algorithm #2

Iteration i+1: repartitioning



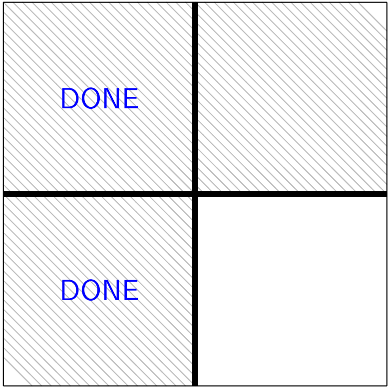
Algorithm #2

Iteration i+1: computation



Algorithm #2

Iteration i+1: completed (boundary shift)



Yet Another Algorithm!

Algorithm #3

State of the matrix:

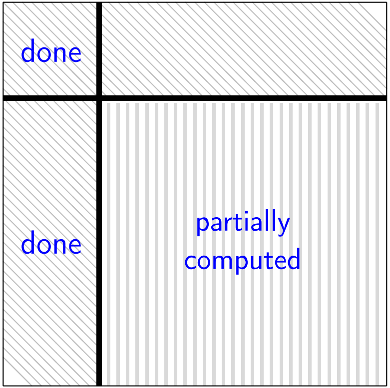
$$\left(\begin{array}{c|c} L_{TL} = \text{CHOL} & \\ \hline L_{BL} = \text{TRSM} & L_{BR} = \text{SYRK} \end{array}\right)$$

Final state:

$$\left(\begin{array}{c|c} L_{TL} = \text{CHOL} & \\ \hline L_{BL} = \text{TRSM} & L_{BR} = \text{CHOL(SYRK)} \end{array}\right)$$

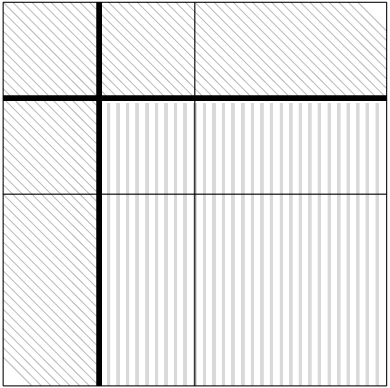
Algorithm #3

Iteration i: completed



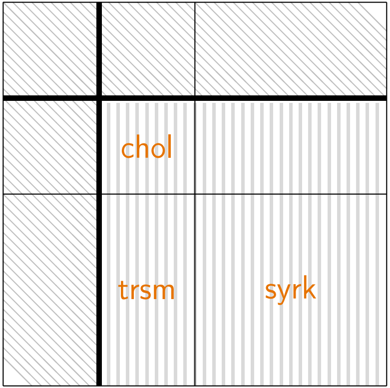
Algorithm #3

Iteration i+1: repartitioning



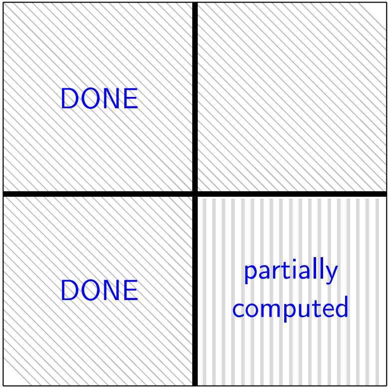
Algorithm #3

Iteration i+1: computation



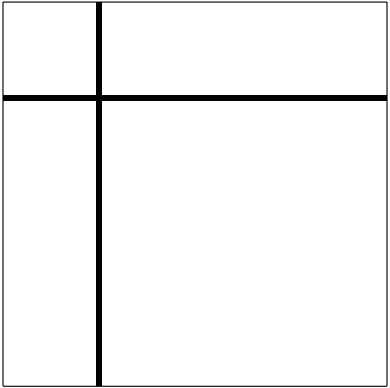
Algorithm #3

Iteration i+1: completed (boundary shift)



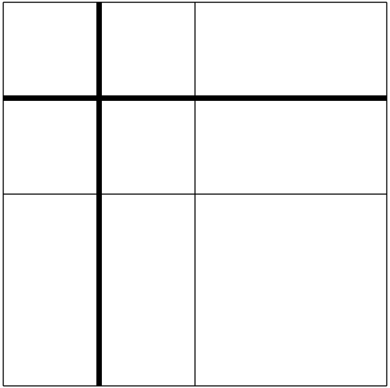
Algorithm Progression

Iteration i: completed



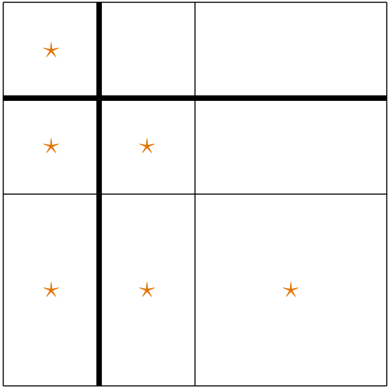
Algorithm Progression

Iteration i+1: repartitioning



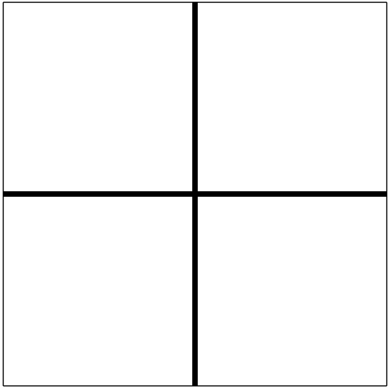
Algorithm Progression

Iteration i+1: computation



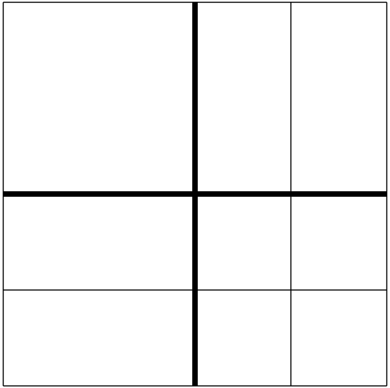
Algorithm Progression

Iteration i+1: completed (boundary shift)



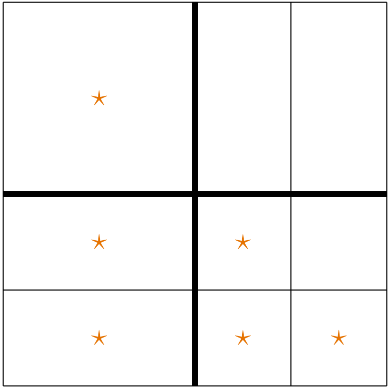
Algorithm Progression

Iteration i+2: repartitioning



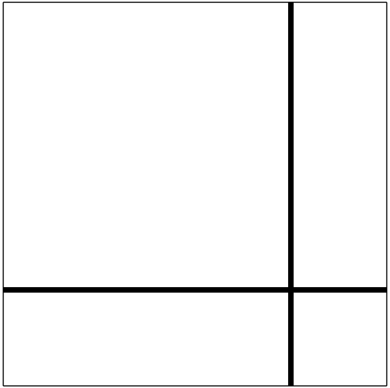
Algorithm Progression

Iteration i+2: computation



Algorithm Progression

Iteration i+2: complete (boundary shift)



Traditional code

- C, triple loop, unblocked.

```
for ( j = 0; j < n; j++ )
{
  A[j,j] = sqrt( A[j,j] );

  for ( i = j+1; i < n; i++ )
    A[i,j] = A[i,j] / A[j,j];

  for ( k = j+1; k < n; k++ )
    for ( i = k; i < n; i++ )
      A[i,k] = A[i,k] - A[i,j] * A[k,j];
}
```

Traditional code

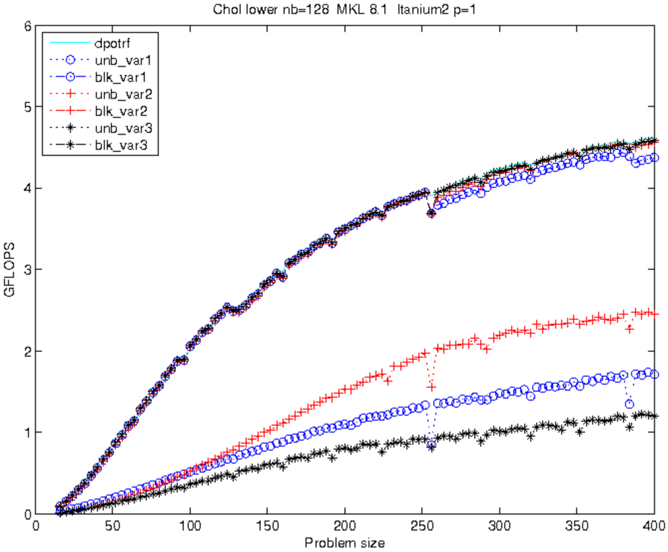
- Matlab, blocked.

```
for j = 1:nb:n,
  b = min( n-j+1, nb );

  A(j:j+b-1, j:j+b-1) = Chol( A(j:j+b-1, j:j+b-1) );

  A(j+b:n,   j:j+b-1) = A(j+b:n, j:j+b-1)/A(j:j+b-1, j:j+b-1)';

  A(j+b:n,   j+b:n  ) = A(j+b:n, j+b:n) -
                        tril(A(j+b:n, j:j+b-1)) A(j+b:n, j:j+b-1)';
end
```

Traditional code: LAPACK, blocked

```
      SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )
      [..]
            DO 20 J = 1, N, NB
*
               JB = MIN( NB, N-J+1 )
               CALL DSYRK( 'Lower', 'No transpose', JB, J-1, -ONE,
     $                     A( J, 1 ), LDA, ONE, A( J, J ), LDA )
               CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
               IF( INFO.NE.0 )
     $            GO TO 30
               IF( J+JB.LE.N-1 ) THEN
*
                  CALL DGEMM( 'No transpose', 'Transpose', N-J-JB+1, JB,
     $                        J-1, -ONE, A( J+JB, 1 ), LDA, A( J, 1 ),
     $                        LDA, ONE, A( J+JB, J ), LDA )
                  CALL DTRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
     $                        N-J-JB+1, JB, ONE, A( J, J ), LDA,
     $                        A( J+JB, J ), LDA )
               END IF
```

Unblocked vs. Blocked Algorithms



Later in Lab #1: LU anyone?

- Without pivoting:
  Precisely the same steps as Cholesky; different PME; different dependencies; 5 algorithms

- With pivoting:
  Same steps, a tad more complicated PME; 3 algorithms

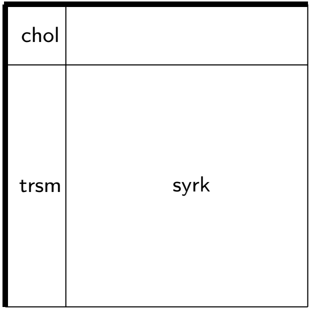
Fundamentals of HPC & LA

Cholesky: unblocked and blocked algorithms

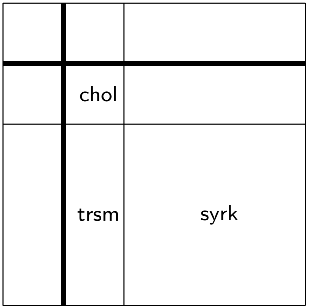
Cholesky: algorithms by blocks

Shared-memory parallelization: Can we do better?

Fork-join $\Rightarrow$ unnecessary synchronizations



Iteration 1



Iteration 2

Synchronization at each iteration; in fact, at each kernel!

Shared-Memory Parallelization

- Traditional (and pipelined) parallelizations are limited by the dependencies dictated by the code.
- Parallelism should be limited only by the data dependencies.
- Idea: imitate a superscalar processor; dynamic detection of data dependencies + out of order execution.

Dependencies (1/3): "True dependency"

Also, "Flow dependency"

```
                    {x = 1, y = 2, a = 3}

...
y := a * x + y
w := 3 * y
...

{y = 5, w = 15}
```

Dependencies (1/3): "True dependency"

Also, "Flow dependency"

```
{x = 1, y = 2, a = 3}


...
y := a * x + y
w := 3 * y
...

{y = 5, w = 15}
```

- The value of `w` depends on the updated value of `y`

Dependencies (1/3): “True dependency”

Also, “Flow dependency”

```
                {x = 1, y = 2, a = 3}


...                             ...
y := a * x + y          |       w := 3 * y
w := 3 * y              |       y := a * x + y
...                             ...

{y = 5, w = 15}                 {y = 5, w = 6}
```

- The value of `w` depends on the updated value of `y`
- The semantics of the program depends on the **order** of the statements

Dependencies (2/3): “Anti dependency”

```
                {x = 1, y = 2, a = 3}

  ...
  w := 3 * y
  y := a * x + y
  ...

  {y = 5, w = 6}
```

- The value of `w` depends on the initial value of `y`

Dependencies (2/3): “Anti dependency”

```
                   {x = 1, y = 2, a = 3}

...                                      ...
w := 3 * y                  |            y := a * x + y
y := a * x + y              |            w := 3 * y
...                                      ...

{y = 5, w = 6}                           {y = 5, w = 15}
```

- The value of `w` depends on the initial value of `y`
- The semantics of the program depends on the **order** of the statements

Dependencies (3/3): "Output dependency"

Also "Write dependency"

```
              {x = 1, y = 2, a = 3}

...
w := 3 * y
w := a * x
...

{w = 3}
```

- The value of `w` depends on the order of the statements

Dependencies (3/3): “Output dependency”

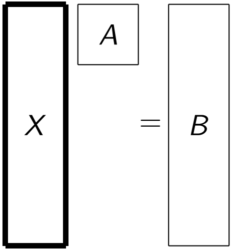
Also “Write dependency”

```
                 {x = 1, y = 2, a = 3}

...                                 ...
w := 3 * y              |           w := a * x
w := a * x              |           w := 3 * y
...                                 ...

{w = 3}                             {w = 6}
```

- The value of `w` depends on the order of the statements
- The semantics of the program depends on the **order** of the statements

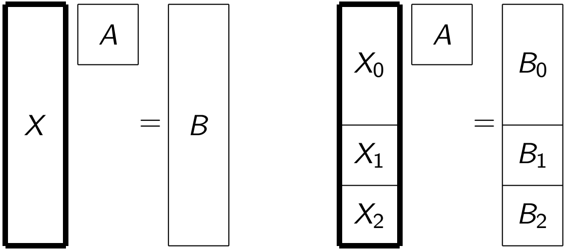
Back to Cholesky: How to create parallelism?

- Idea: decompose the tasks



Back to Cholesky: How to create parallelism?

- Idea: decompose the tasks



Back to Cholesky: How to create parallelism?

- Idea: decompose the tasks

$$X \; A = B \qquad\qquad \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} A = \begin{bmatrix} B_0 \\ B_1 \\ B_2 \end{bmatrix}$$

$$X_0 A = B_0$$

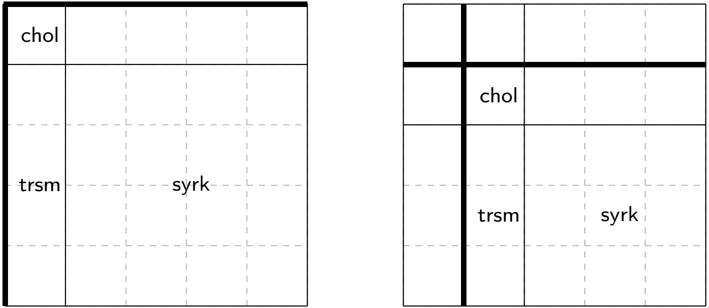
$$X_1 A = B_1$$

$$X_2 A = B_2$$

Algorithms by blocks

Also, "Tiled algorithms". Not "blocked"!

Goal: Create small tasks, feed all processors as early as possible



Iteration i   Iteration i+1

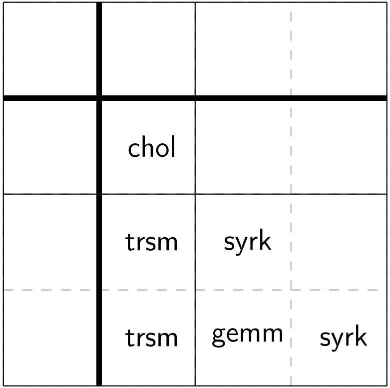
Breaking down the computation

Decomposition in tiles (iteration 1)

chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

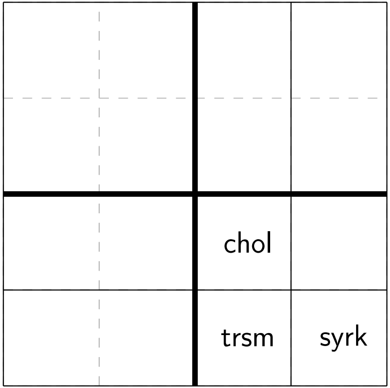
Breaking down the computation

Decomposition in tiles (iteration 2)



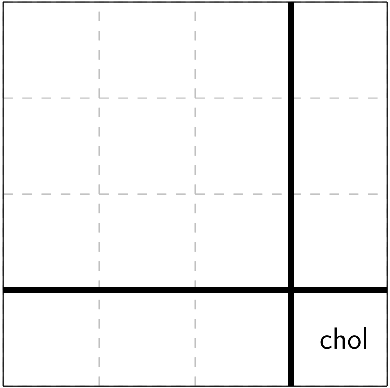
Breaking down the computation

Decomposition in tiles (iteration 3)



Breaking down the computation

Decomposition in tiles (iteration 4)



Dependencies

chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

Dependencies

chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

Dependencies

chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

Dependencies

chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

Dependencies

chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

Dependencies

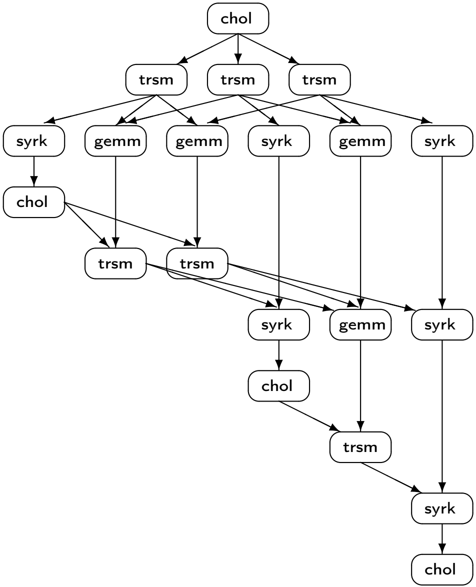
chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

Dependencies

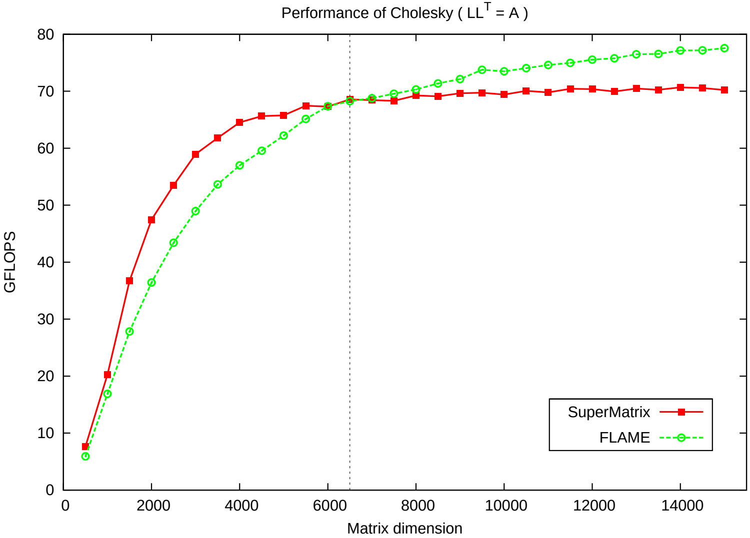
chol			
trsm	syrk		
trsm	gemm	syrk	
trsm	gemm	gemm	syrk

DAG - Dependencies

4 $\times$ 4-tile matrix



Crossover, 16 cores



Runtime systems, e.g., SuperMatrix, StarPU, Quark, ...

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, ...

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, . . .

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, ...

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.
- Threads asynchronously dequeue tasks from the queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, ...

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.
- Threads asynchronously dequeue tasks from the queue.
- Upon termination of a task, the thread notifies dependent tasks and updates the queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, . . .

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.
- Threads asynchronously dequeue tasks from the queue.
- Upon termination of a task, the thread notifies dependent tasks and updates the queue.
- Loop until all tasks complete execution.

Task Execution

4×4-tile matrix

Stage	Scheduled Tasks			
1	chol			
2	trsm	trsm	trsm	trsm
3	syrk	gemm	syrk	gemm
4	gemm	syrk	gemm	gemm
5	gemm	syrk	chol	
6	trsm	trsm	trsm	
7	syrk	gemm	syrk	gemm
8	gemm	syrk	chol	
9	trsm	trsm		
10	syrk	gemm	syrk	
11	chol			
12	trsm			
13	syrk			
14	chol			

SPD Inv: 1) Chol 2) Inv 3) Mat Mat Mult.

5×5-tile matrix

$$A := A^{-1}$$

$$A := \left(LL^T\right)^{-1}$$

$$A := L^{-T}L^{-1}$$

SPD Inv: 1) Chol 2) Inv 3) Mat Mat Mult.

5 × 5-tile matrix

Stage	Scheduled Tasks			
1	chol			
2	trsm	trsm	trsm	trsm
3	syrk	gemm	syrk	gemm
4	gemm	syrk	gemm	gemm
5	gemm	syrk	chol	trsm
6	trsm	trsm	trsm	trsm
7	trsm	trsm	trinv	syrk
8	gemm	syrk	gemm	gemm
9	syrk	ttmm	chol	trsm
10	trsm	trsm	trsm	trsm
11	gemm	gemm	gemm	syrk
12	gemm	syrk	trsm	chol
13	trsm	trsm	trinv	syrk
14	trsm	gemm	gemm	gemm
15	gemm	trmm	syrk	trsm
16	trsm	ttmm	chol	trsm
17	syrk	trinv	gemm	syrk
18	gemm	gemm	gemm	trmm
19	trmm	trsm	trsm	trsm
20	trsm	trsm	trsm	trsm
21	ttmm	syrk	gemm	syrk
22	trinv	gemm	gemm	trinv
23	syrk	syrk	gemm	syrk
24	trmm	gemm	trmm	gemm
25	trmm	syrk	gemm	gemm
26	ttmm	gemm	trmm	trmm
27	syrk	trmm		
28	trmm			
29	ttmm			

Not quite so simple

Not quite so simple

- Storage by blocks

Not quite so simple

- Storage by blocks
- Critical path

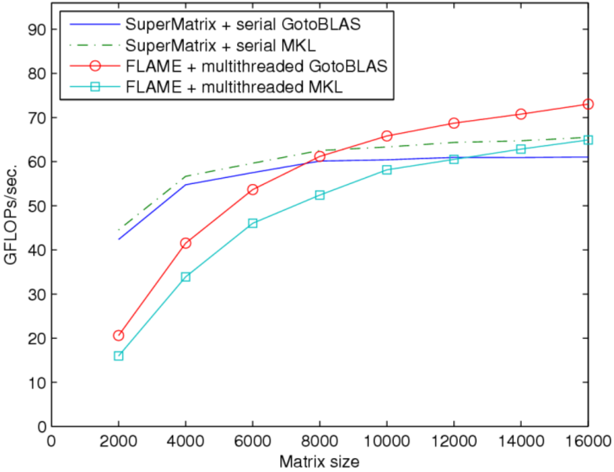
Not quite so simple

- Storage by blocks
- Critical path
- Cache “simulator”

Not quite so simple

- Storage by blocks
- Critical path
- Cache "simulator"
- Tension between size of blocks and number of blocks

SPD Inverse, 16 cores



Multithreaded BLAS vs. Algorithms-by-blocks

No absolute winner: crossover!

Multithreaded BLAS	Algorithms-by-blocks
✓ Ease of use	✓ Out of order execution
✖ Synchronization	✓ Parallelism dictated by data dependencies
	✖ Plateux