Compiling Recurrences To Imperative Code

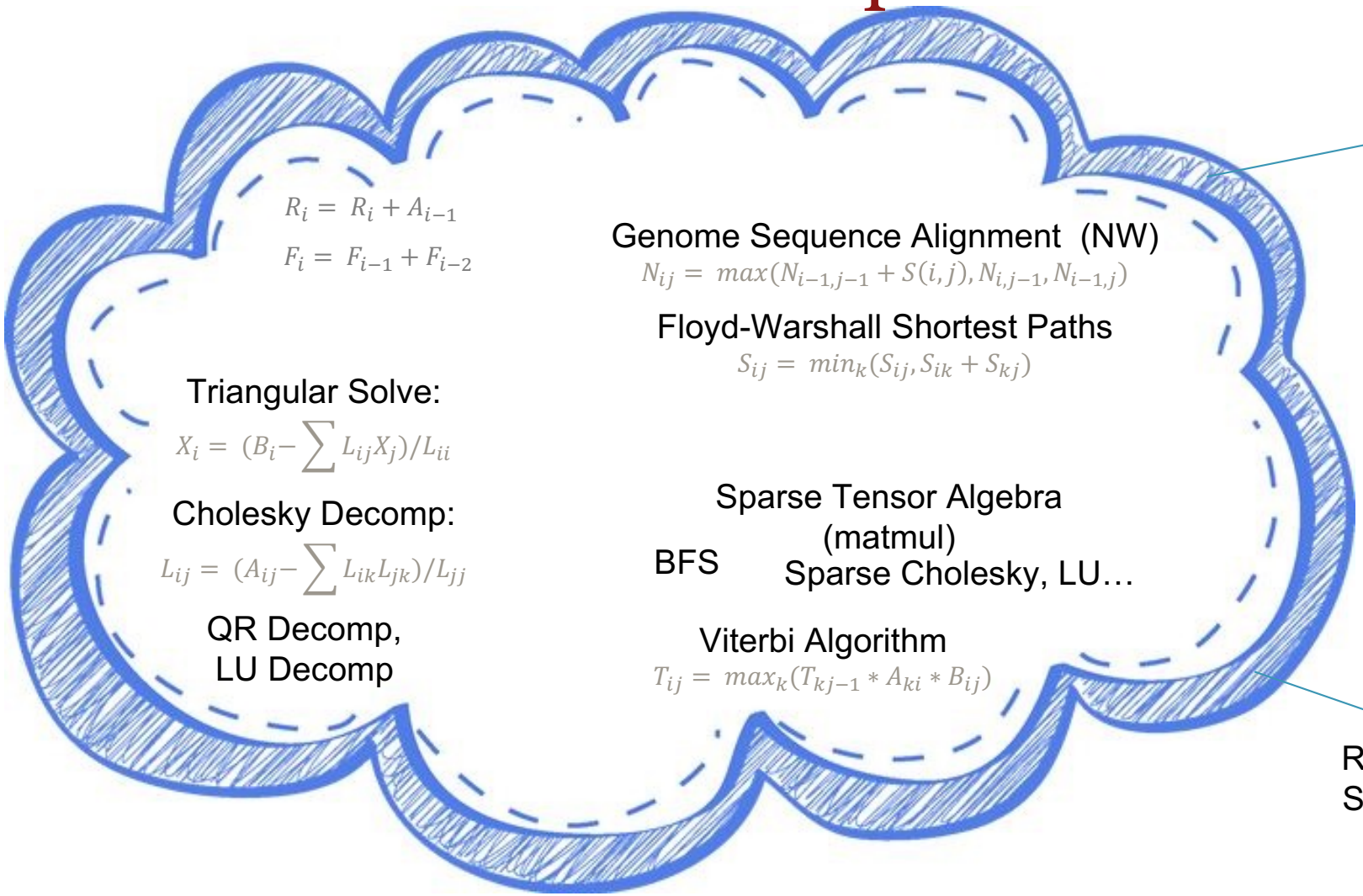
Shiv Sundram, Muhammad Usman Tariq,
Fred Kjolstad

Stanford University

Recurrence Equations:

$$R_i = R_{i-1} + A_i$$

Recurrence Equations:



$R_i = R_i + A_{i-1}$

$F_i = F_{i-1} + F_{i-2}$

Genome Sequence Alignment (NW)

$N_{ij} = max(N_{i-1,j-1} + S(i,j), N_{i,j-1}, N_{i-1,j})$

Floyd-Warshall Shortest Paths

$S_{ij} = min_k(S_{ij}, S_{ik} + S_{kj})$

Triangular Solve:

$X_i = (B_i - \sum L_{ij}X_j)/L_{ii}$

Cholesky Decomp:

$L_{ij} = (A_{ij} - \sum L_{ik}L_{jk})/L_{jj}$

QR Decomp,
LU Decomp

Sparse Tensor Algebra
(matmul)

BFS

Sparse Cholesky, LU…

Viterbi Algorithm

$T_{ij} = max_k(T_{kj-1} * A_{ki} * B_{ij})$

Loop
Interchange

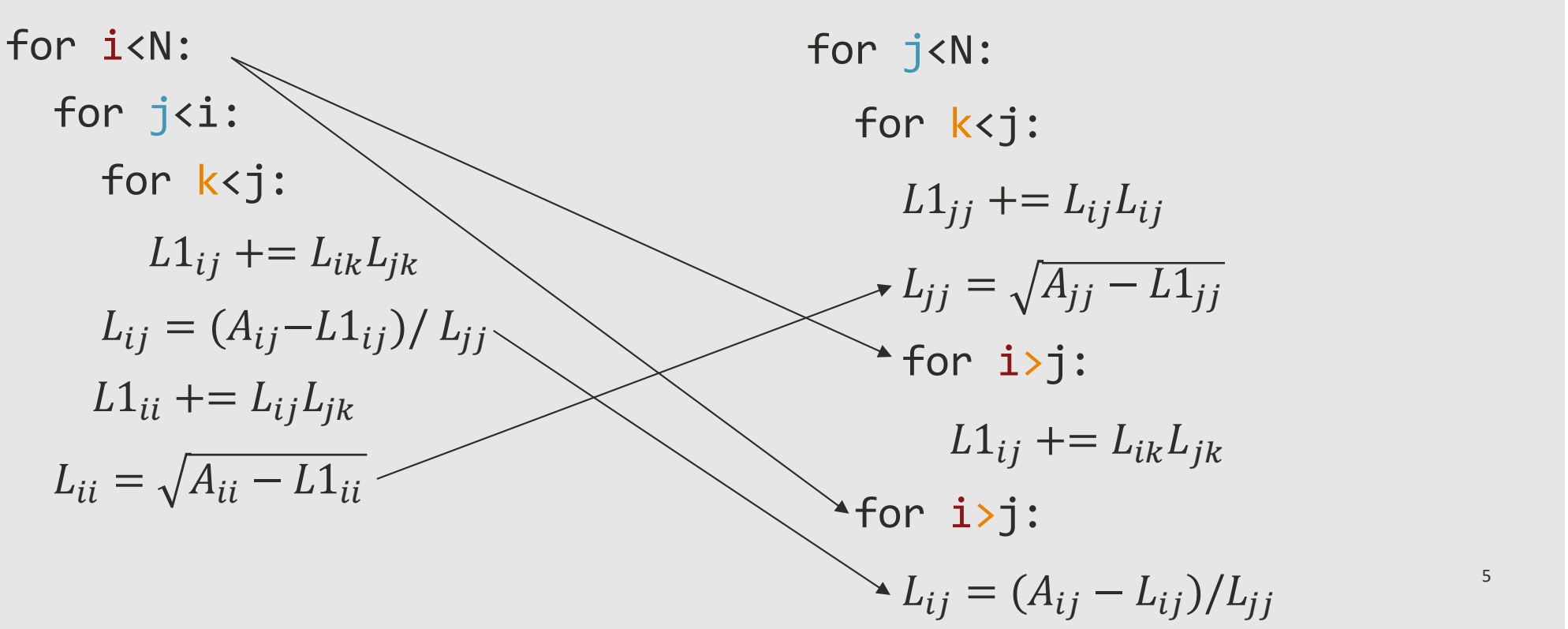
Loop
Fusion

Parallelization

Row-major arrays,
Sparse arrays, etc

Optimizations: Not as trivial for recurrences

Optimizations: Not as trivial for recurrences



RECUMA (Recurrence Compilation Machine)

Recurrence Equations

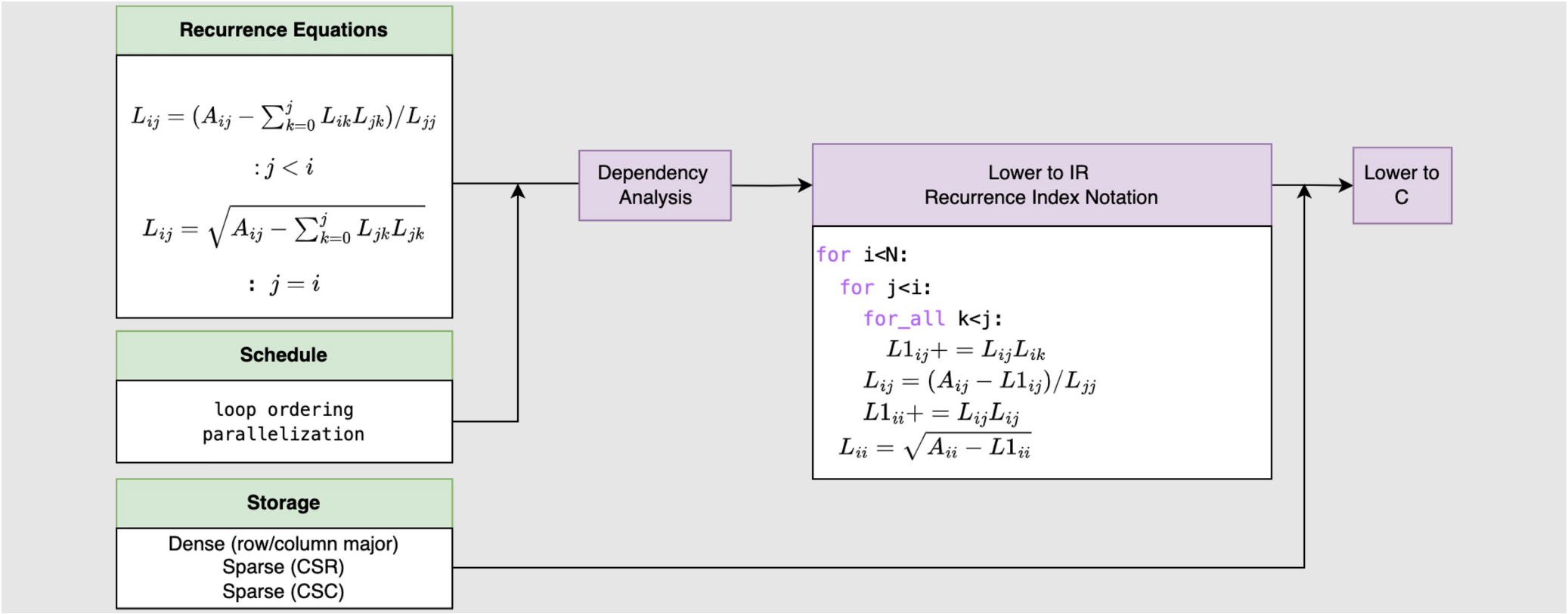
$$L_{ij} = (A_{ij} - \sum_{k=0}^{j} L_{ik} L_{jk}) / L_{jj}$$

$$: j < i$$

$$L_{ij} = \sqrt{A_{ij} - \sum_{k=0}^{j} L_{jk} L_{jk}}$$

$$: \ j = i$$

RECUMA (Recurrence Compilation Machine)



Recurrence Language

$$L_{ij} = (A_{ij} - \sum_k^n B_{ik} B_{jk}) / B_{jj}$$

Constrain Iteration Bounds:

Recurrence Language

Constrain Iteration Bounds:

$$L_{ij} = (A_{ij} - \sum_k^j B_{ik} B_{jk}) / B_{jj} \quad : k < j < i$$

Recurrence Language

Make Computation
in-place:

$$L_{ij} = (A_{ij} - \sum_k^j B_{ik} B_{jk}) / B_{jj} \quad : k < j < i$$

Recurrence Language

Make Computation
in-place:

$$L_{ij} = (A_{ij} - \sum_k^j L_{ik} L_{jk}) / L_{jj} \quad : k < j < i$$

Recurrence Language

Multiple Equations

$$L_{ij} = (A_{ij} - \sum_k^j L_{ik} L_{jk}) / L_{jj} \quad : k < j < i$$

$$L_{ij} = \sqrt{A_{ij} - \sum_k^j L_{ik} L_{jk}} \quad : k < j = i$$

Recurrence Language

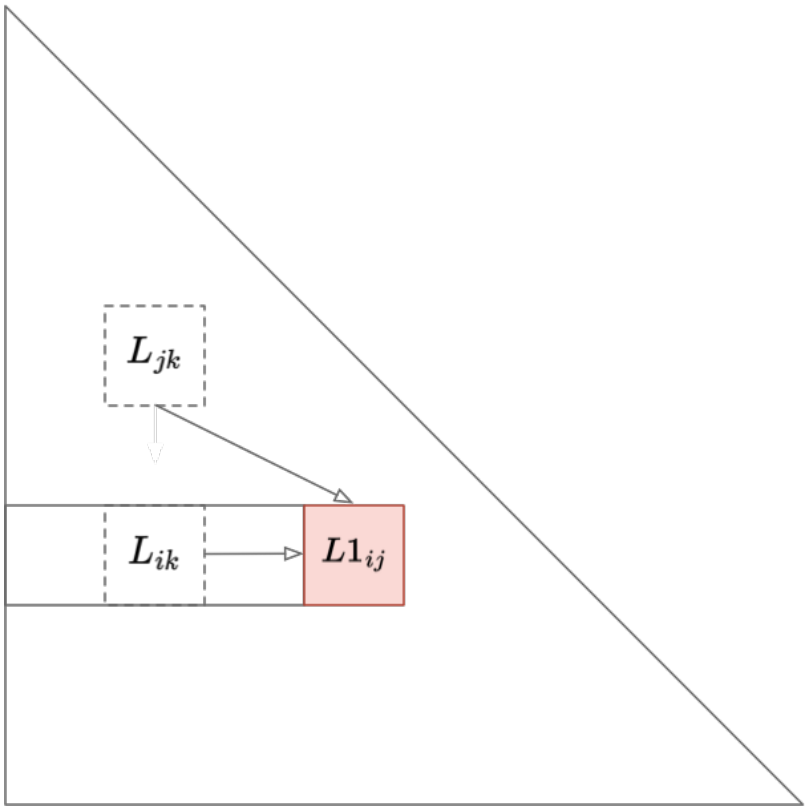
Multiple Equations

$$L_{ij} = (A_{ij} - \sum_k^j L_{ik} L_{jk}) / L_{jj} \quad : k < j < i$$

$$L_{ij} = \sqrt{A_{ij} - \sum_k^j L_{ik} L_{jk}} \quad : k < j = i$$

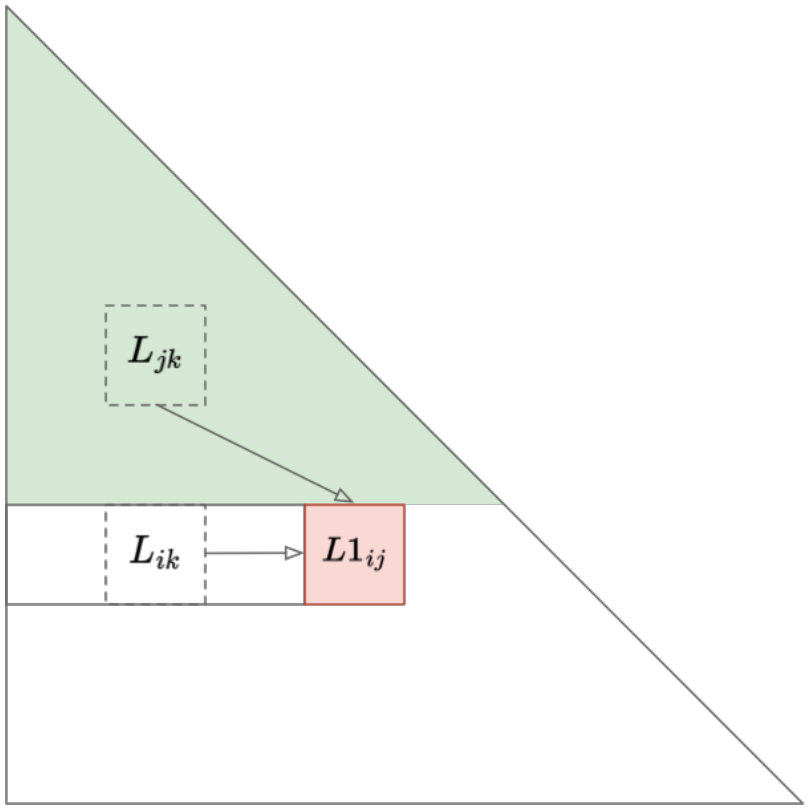
Cholesky Decomposition as a system of mutually dependent recurrences

Cholesky Dependencies



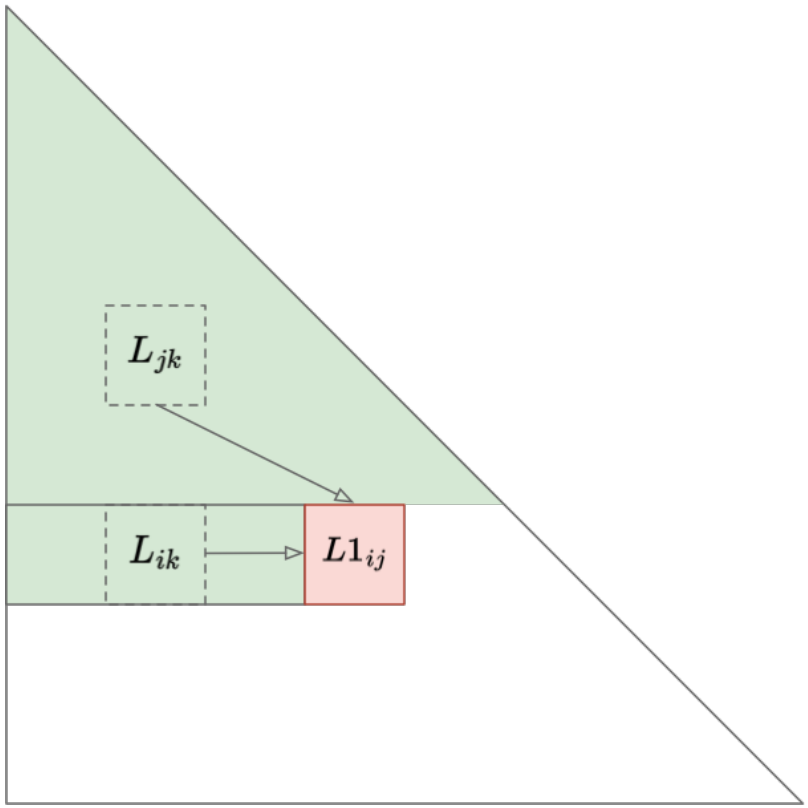
$$L1_{ij} = \sum_{k=0}^{j} L_{ik} L_{jk} : j < i$$

Cholesky Dependencies



$$L1_{ij} = \sum_{k=0}^{j} L_{ik} L_{jk} : j < i$$

Cholesky Dependencies



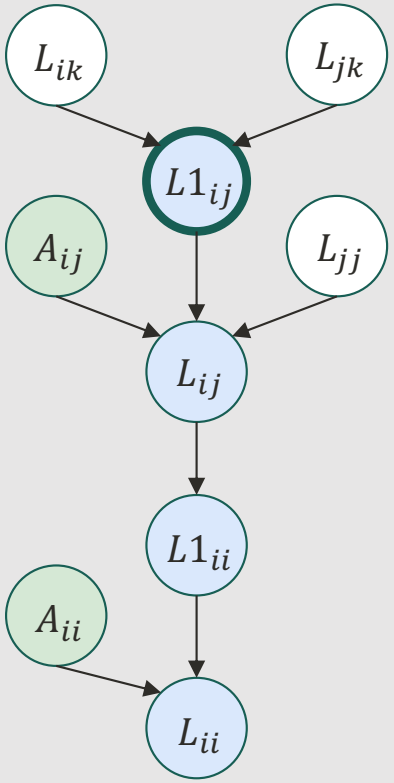
$$L1_{ij} = \sum_{k=0}^{j} L_{ik} L_{jk} : j < i$$

Dependency Abstraction

$$L1_{ij} = \sum_{k=0}^{j} L_{ik} L_{jk} : j < i$$

Code Generation

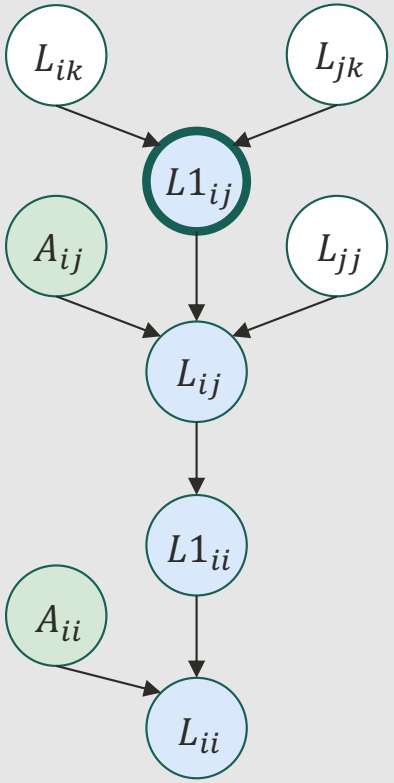
```
for i < N:

  for j < i:

    for k<j
```

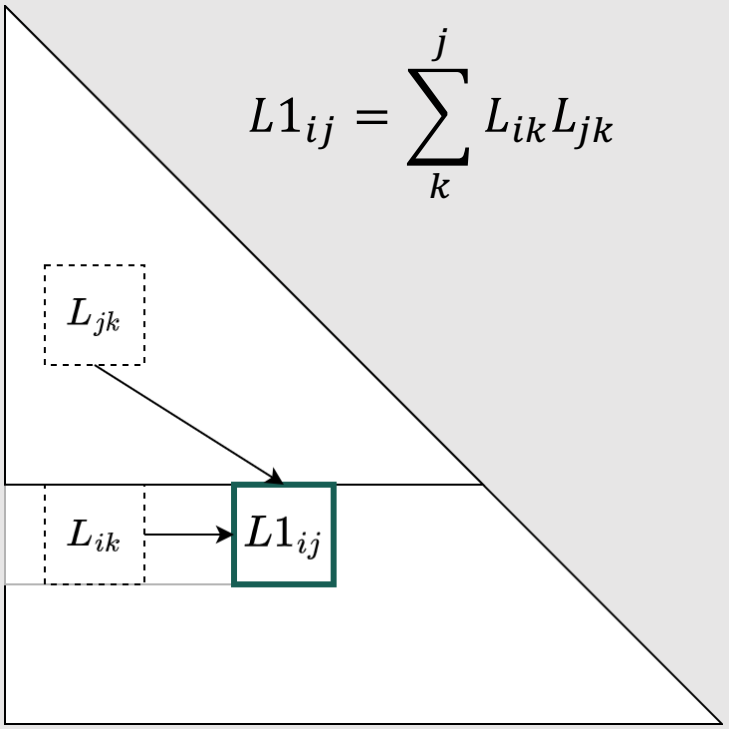


$$L1_{ij} = \sum_{k}^{j} L_{ik} L_{jk}$$

Code Generation

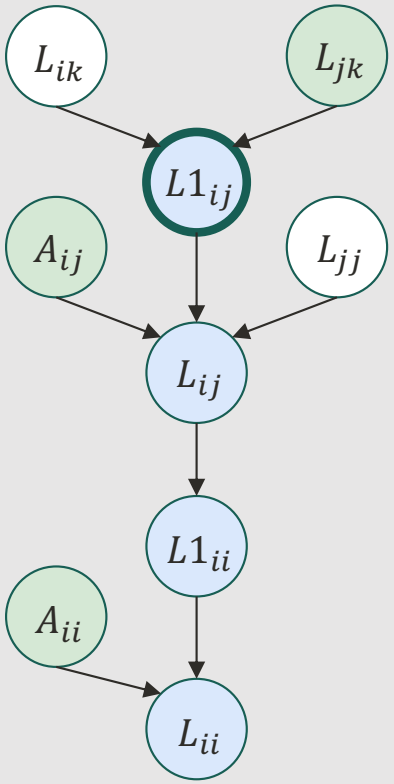
```
for i < N:

  for j < i:

    for k<j
```

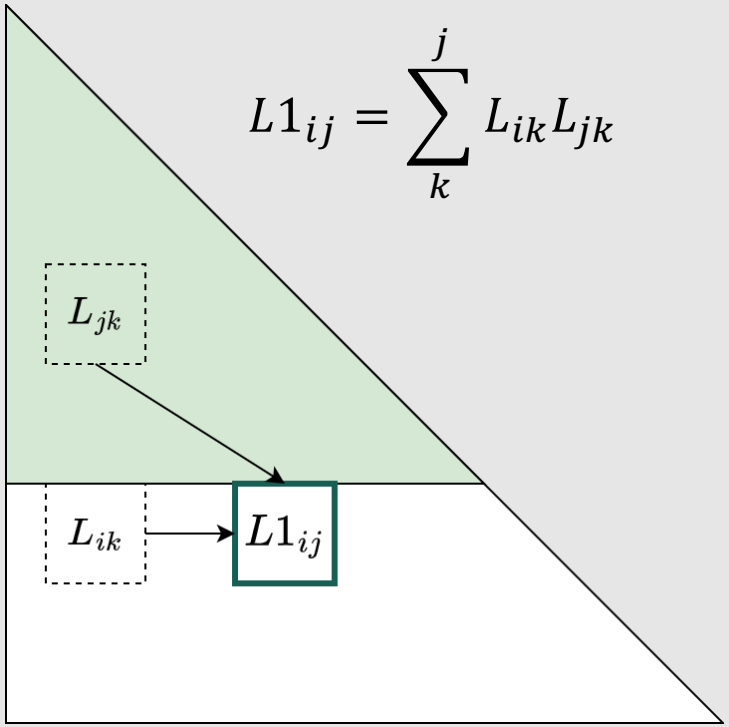




Code Generation

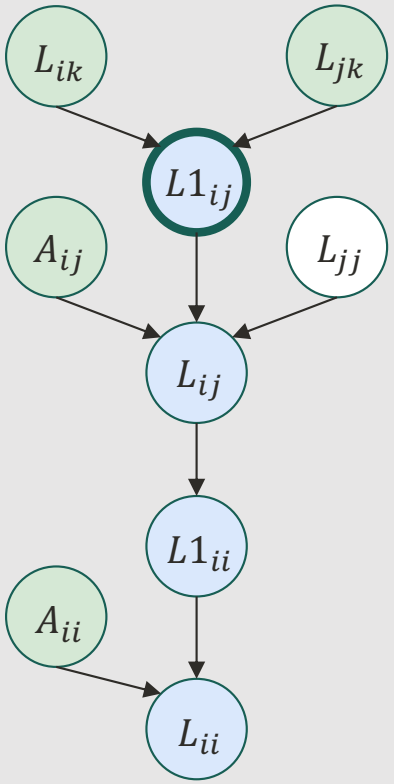
```
for i < N:
  #L(:i ,:) ready
  for j < i:

    for k<j
```

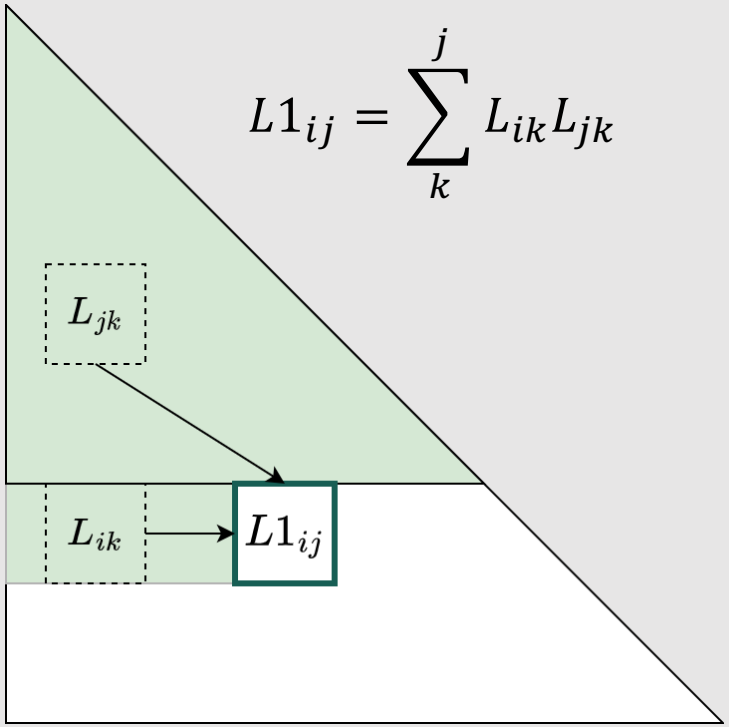




Code Generation

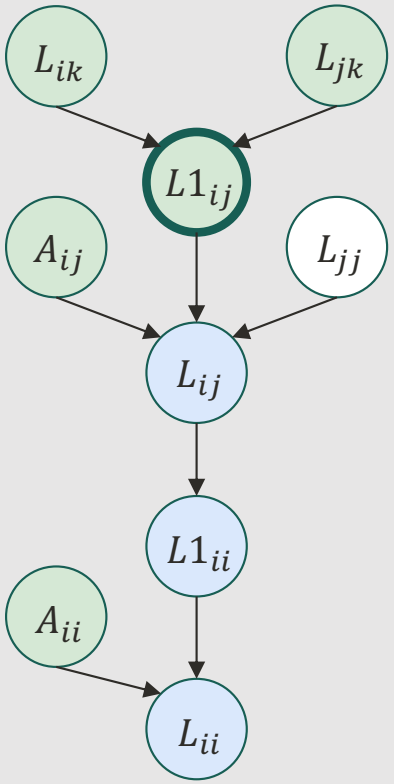
```
for i < N:
   #L(:i,:) ready
   for j < i:
      #L(i,:j) ready
      for k<j
```

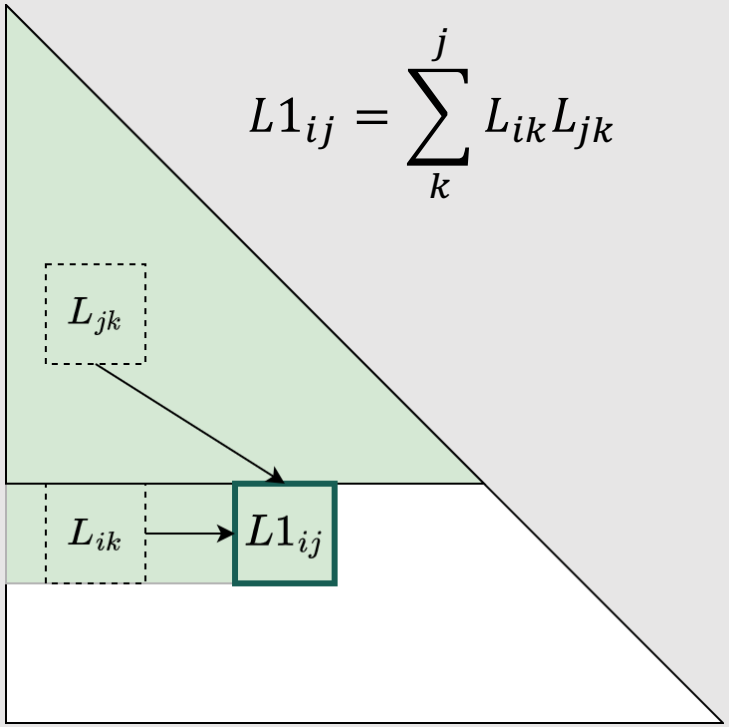




Code Generation

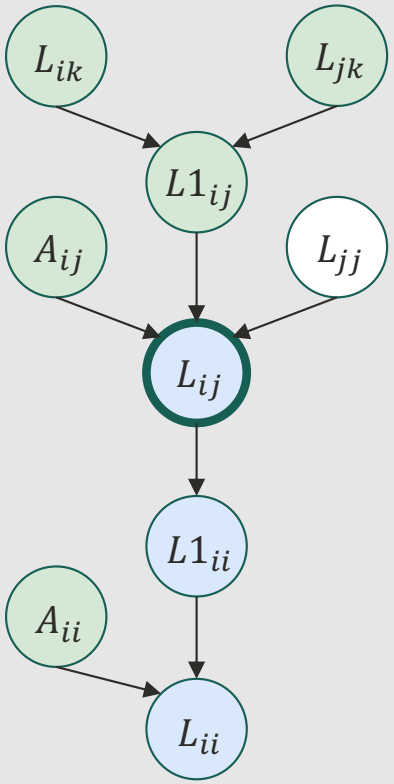
```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += Ljk Lik
```

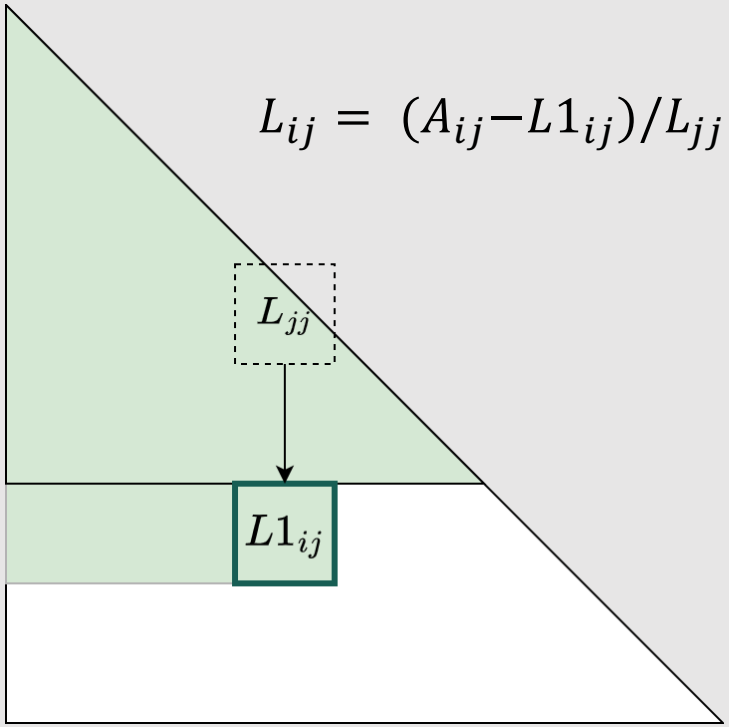




Code Generation

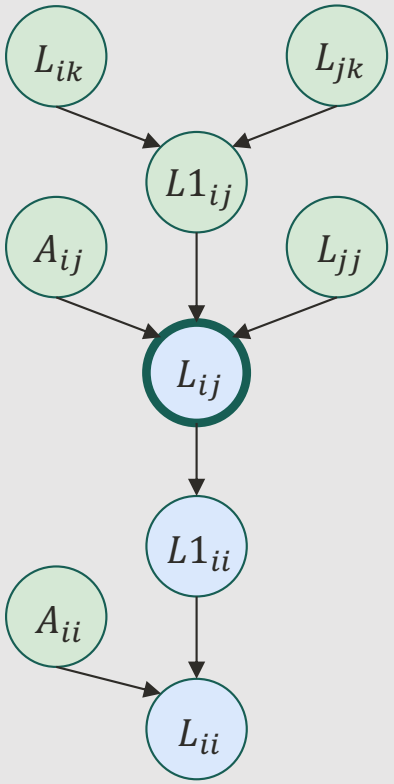
```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += LjkLik
```

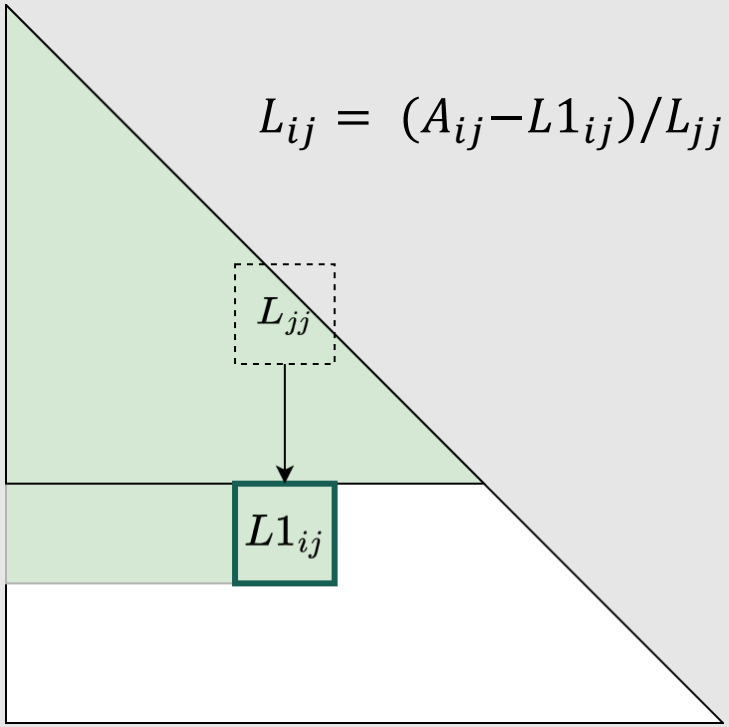




Code Generation

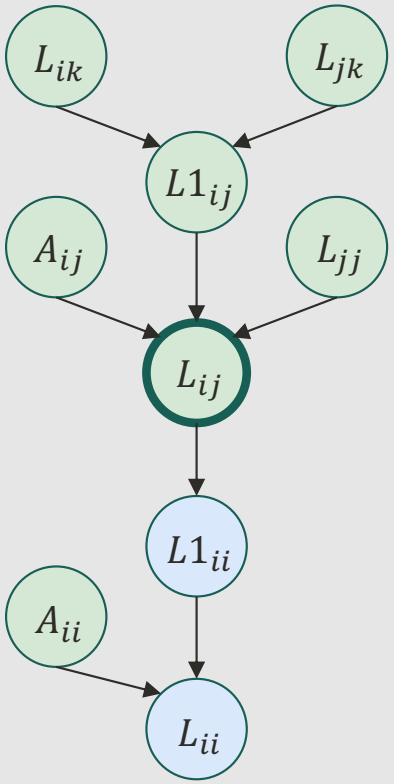
```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += LjkLik
```

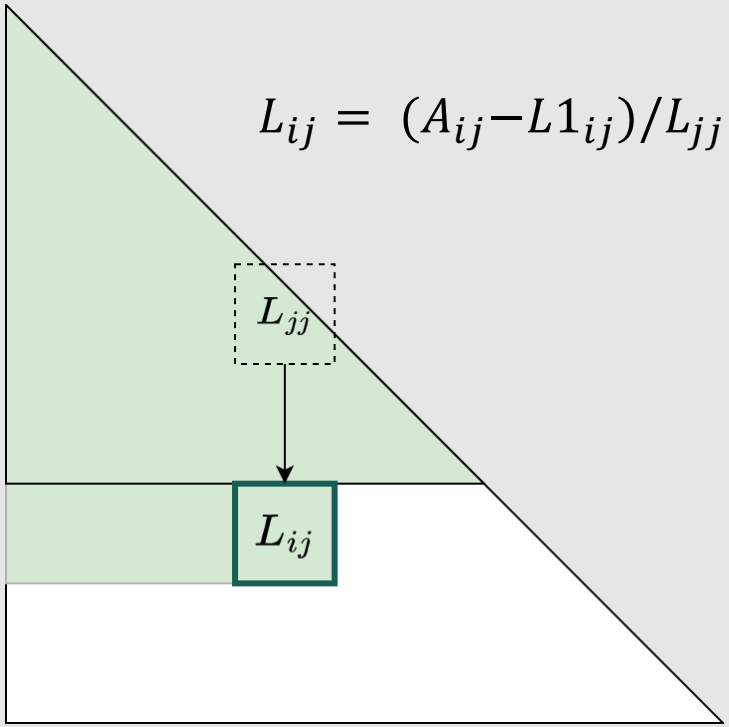




Code Generation

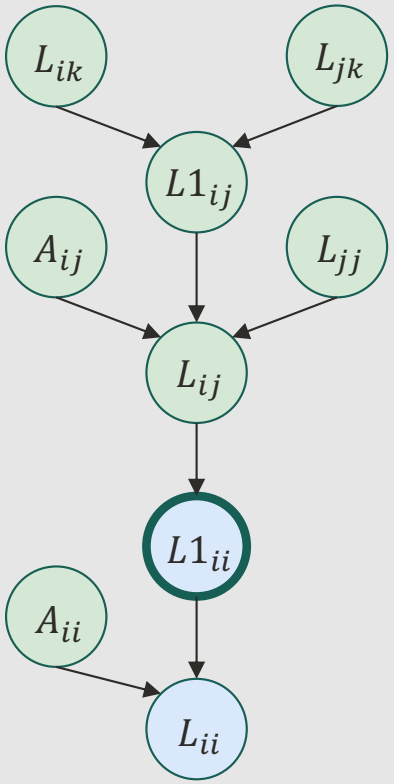
```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += Ljk Lik
    Lij = (Aij − L1ij)/Ljj
```

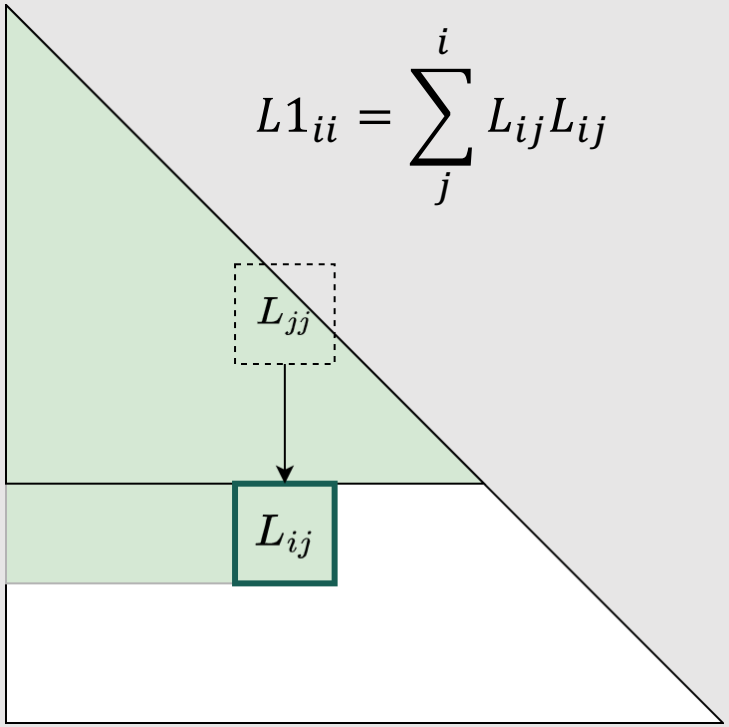




Code Generation

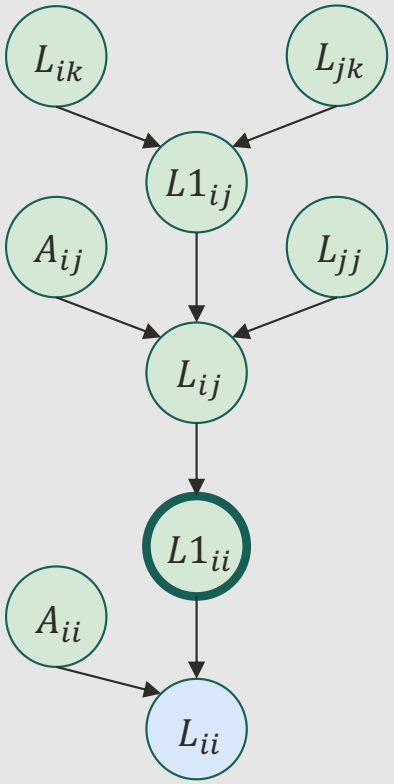
```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += Ljk Lik
    Lij = (Aij − L1ij)/Ljj
```

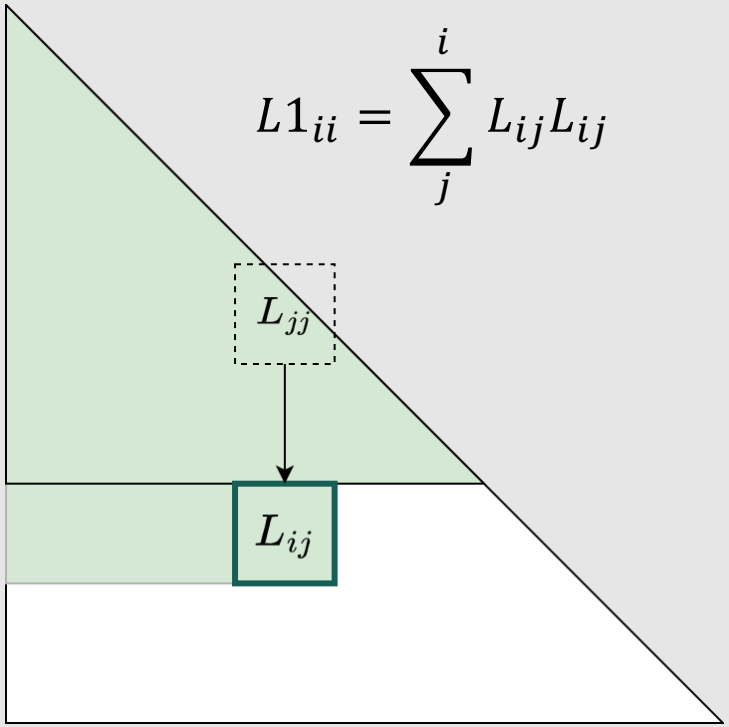




Code Generation

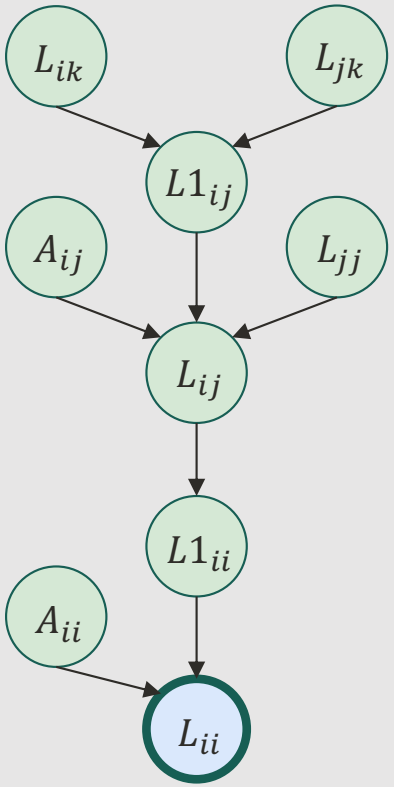
```
for i < N:
   #L(:i,:) ready
   for j < i:
      #L(i,:j) ready
      for k<j
         L1ij += Ljk Lik
      Lij = (Aij − L1ij)/Ljj
      L1ii += Lij Lij
```

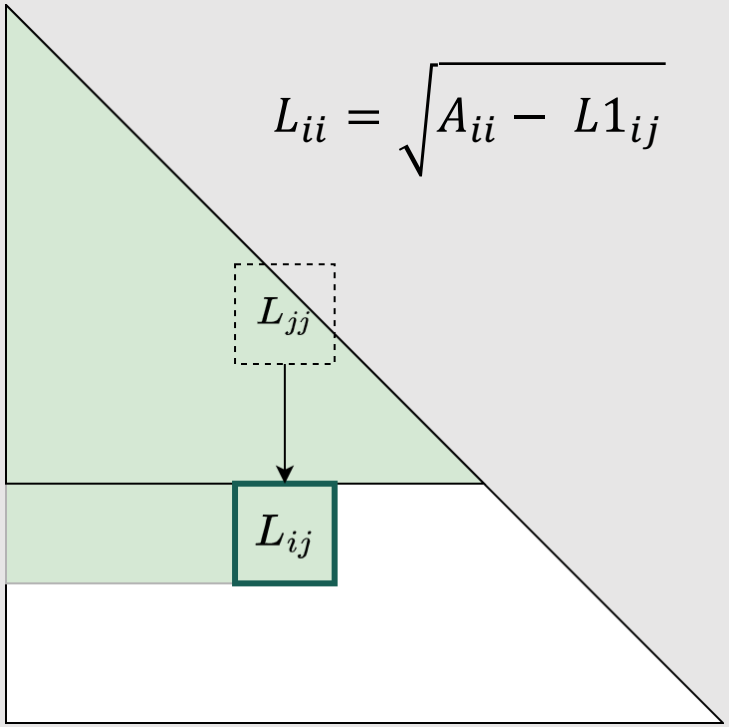




Code Generation

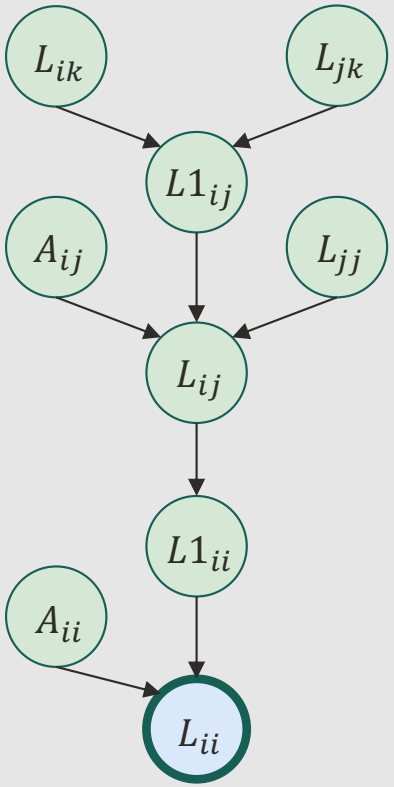
```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += Ljk Lik
    Lij = (Aij − L1ij)/Ljj
    L1ii += Lij Lij
```

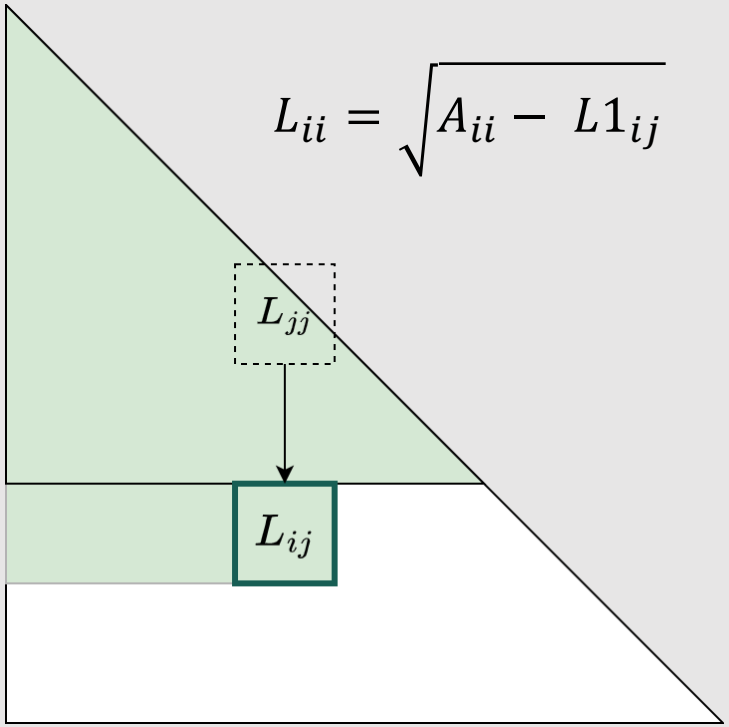




Code Generation

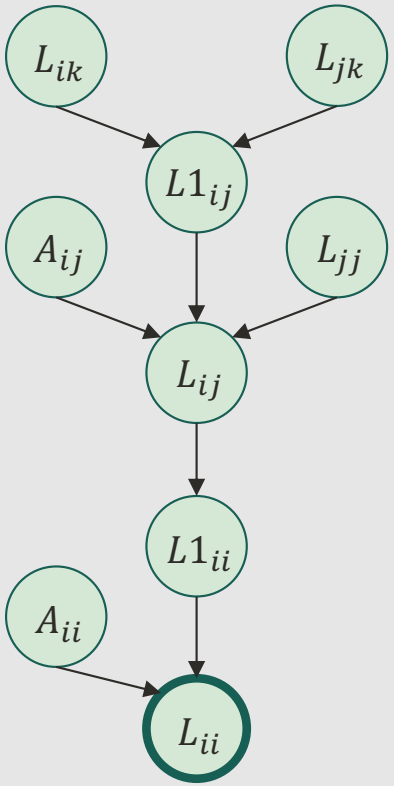
```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += Ljk Lik
    Lij = (Aij − L1ij)/Ljj
    L1ii += Lij Lij
  Lii = √(Aii − L1ii)
```

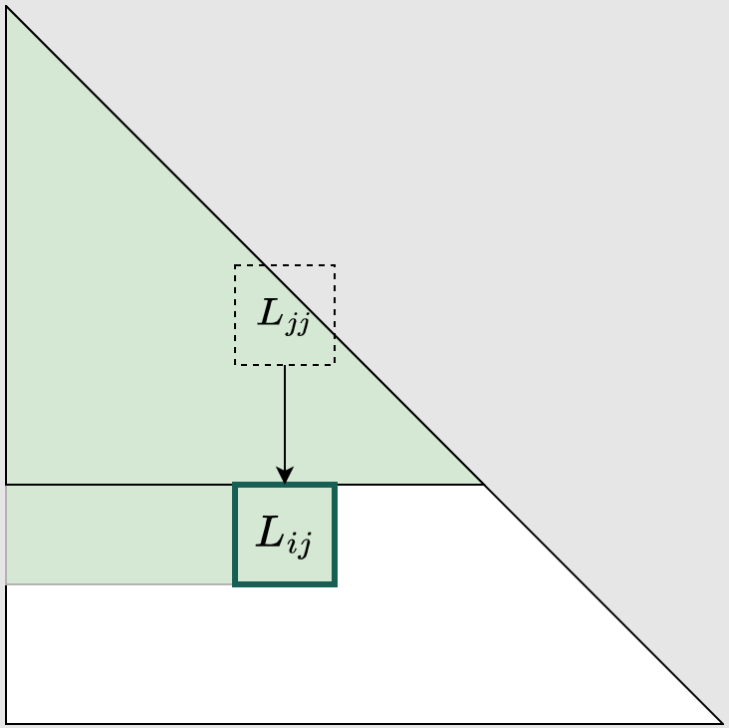




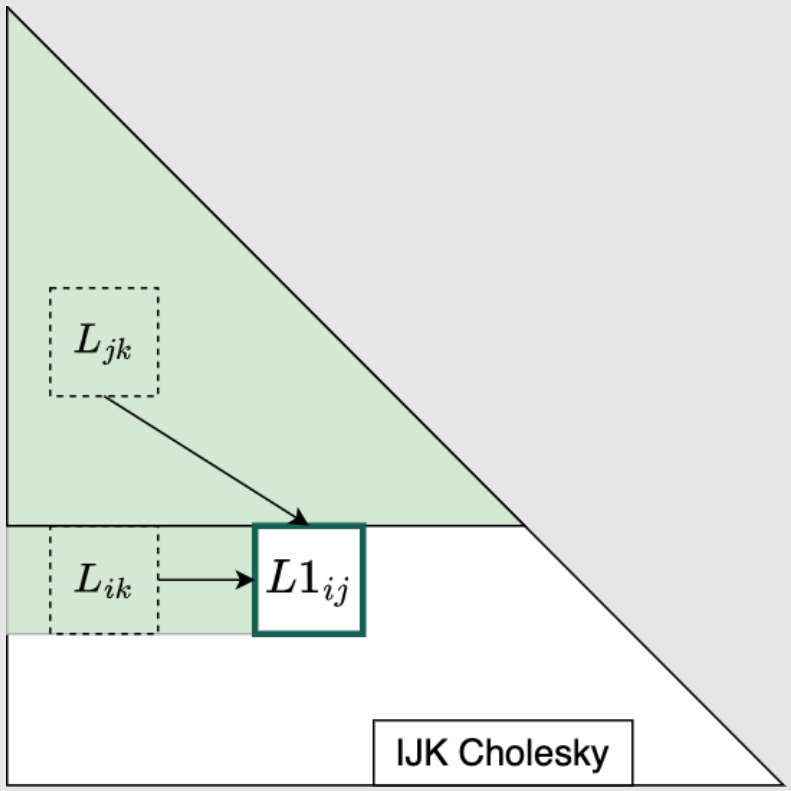
Code Generation

```
for i < N:
  #L(:i,:) ready
  for j < i:
    #L(i,:j) ready
    for k<j
      L1ij += Ljk Lik
    Lij = (Aij − L1ij)/Ljj
    L1ii += Lij Lij
  Lii = √(Aii − L1ii)
```

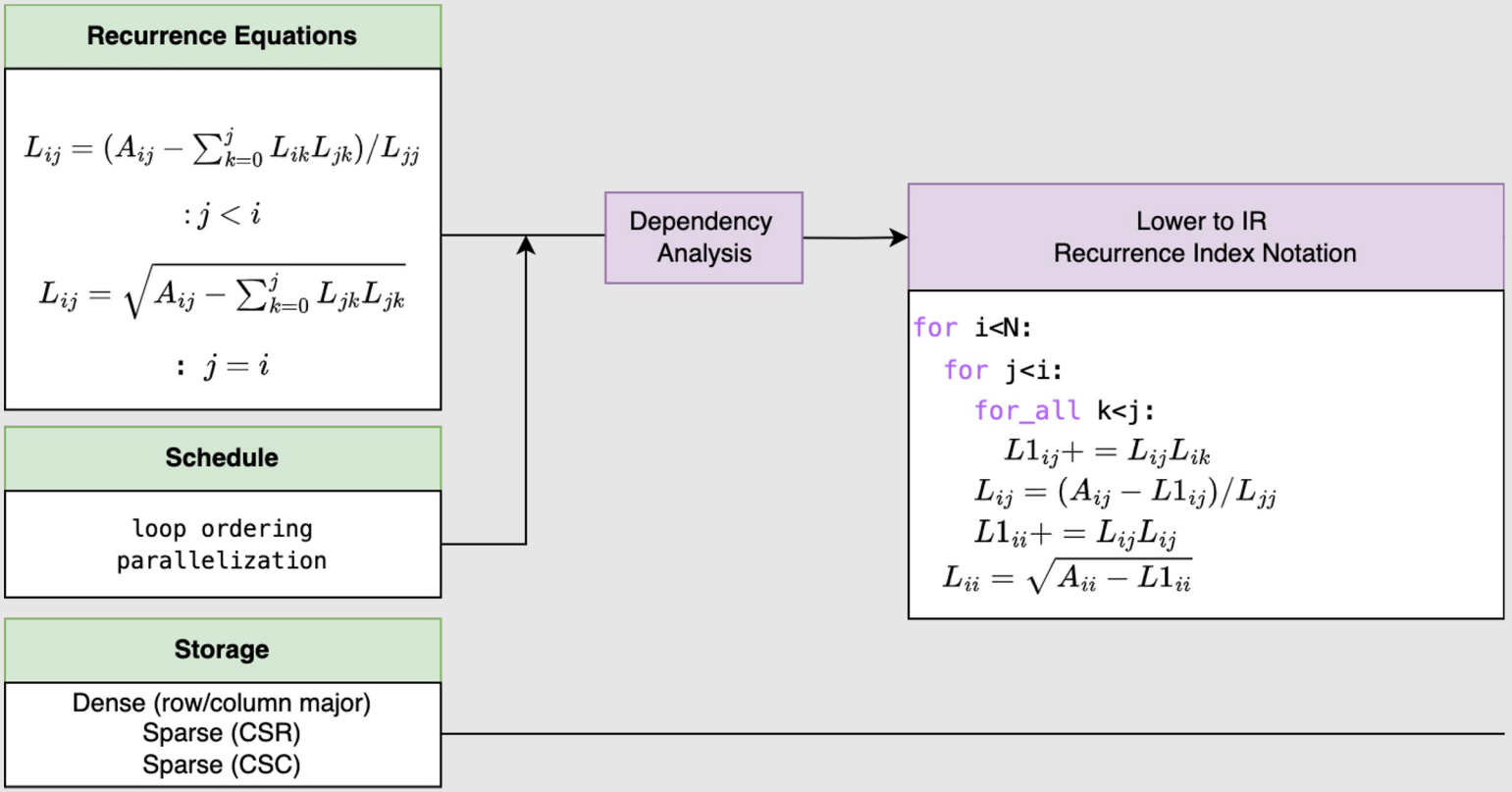




Different schedules



RECUMA (Recurrence Compilation Machine)



Loop Fusion

Code gen/placement algorithm is greedy → automatic loop fusion

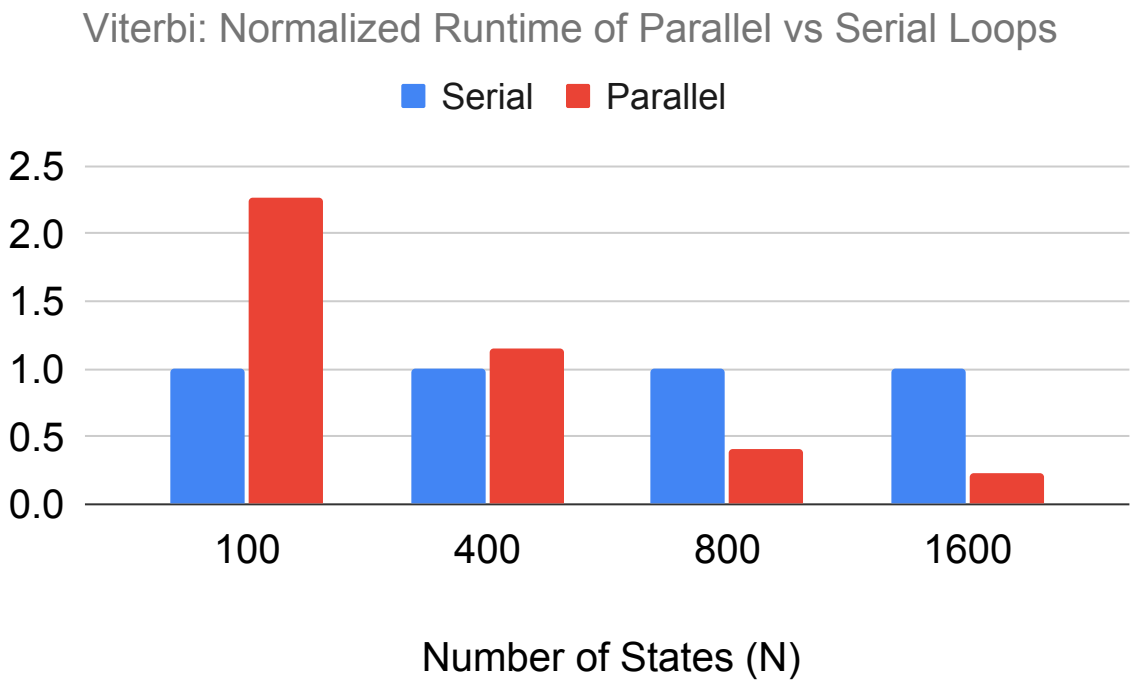
$$Ax_1 = b_1$$

$$Ax_2 = b_2$$

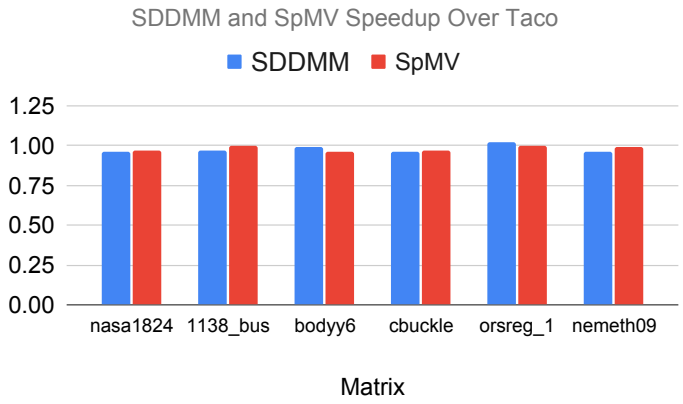
1.98-1.92x speedup when fused

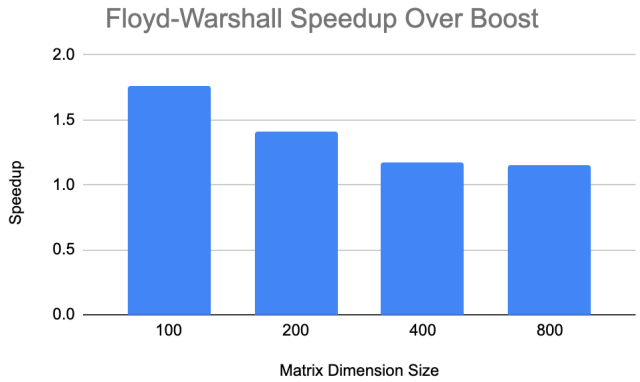
Parallelization

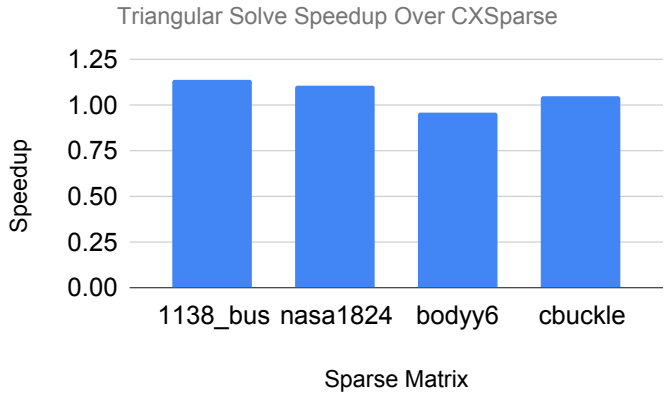
Auto-parallelization of dense Viterbi equation

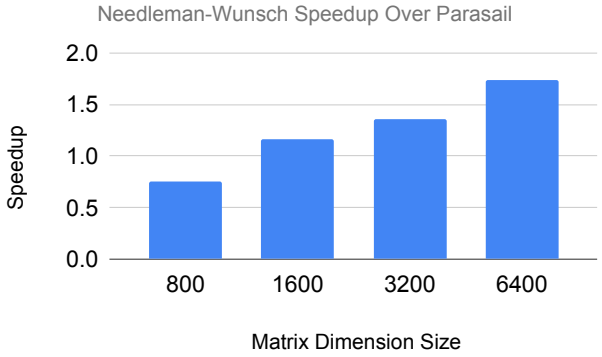


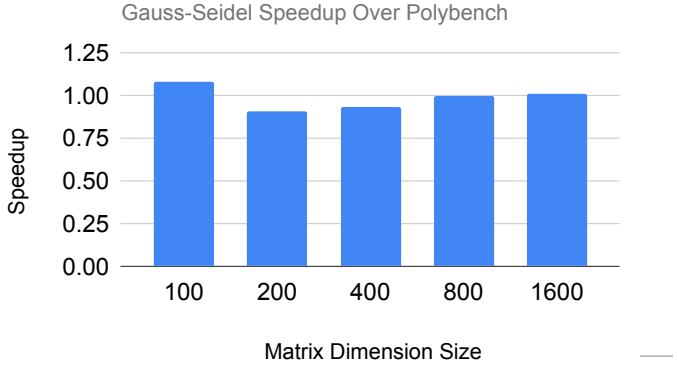
Performance parity with handwritten libraries

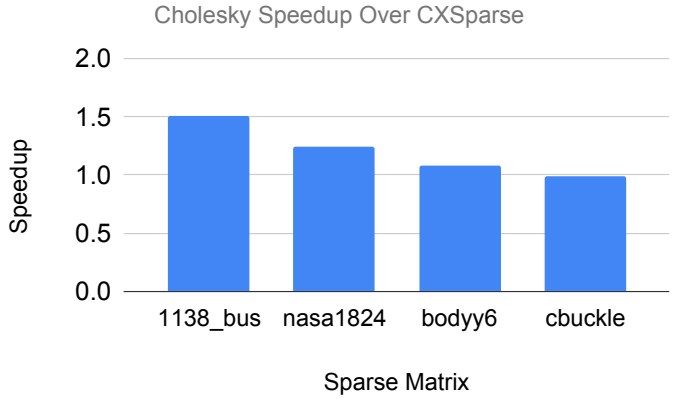




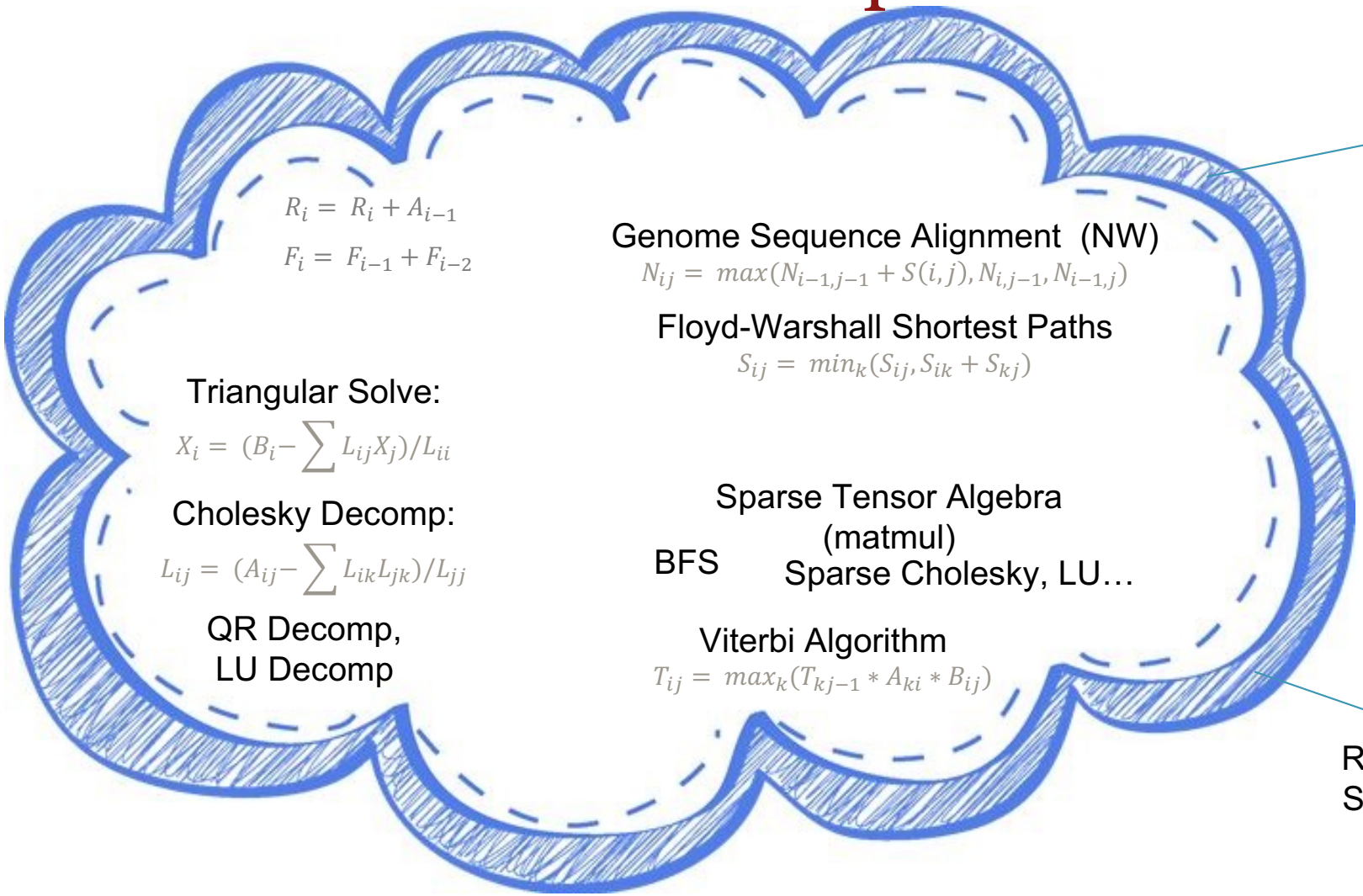








Recurrence Equations:



$R_i = R_i + A_{i-1}$

$F_i = F_{i-1} + F_{i-2}$

Genome Sequence Alignment (NW)

$N_{ij} = max(N_{i-1,j-1} + S(i,j), N_{i,j-1}, N_{i-1,j})$

Floyd-Warshall Shortest Paths

$S_{ij} = min_k(S_{ij}, S_{ik} + S_{kj})$

Triangular Solve:

$X_i = (B_i - \sum L_{ij}X_j)/L_{ii}$

Cholesky Decomp:

$L_{ij} = (A_{ij} - \sum L_{ik}L_{jk})/L_{jj}$

QR Decomp,
LU Decomp

Sparse Tensor Algebra
(matmul)

BFS

Sparse Cholesky, LU…

Viterbi Algorithm

$T_{ij} = max_k(T_{kj-1} * A_{ki} * B_{ij})$

Loop
Interchange

Loop
Fusion

Parallelization

Row-major arrays,
Sparse arrays, etc

Solvers, Sequence Alignment, Graphs, Dynamic Programs...

Path to putting these on a common foundation

Optimizations shared & reused across different algorithms

Future: Tiling, wavefront parallelism, supernodal cholesky, new hardware

Thank you

Solvers, Sequence Alignment, Graphs, Dynamic Programs…

Path to putting these on a common foundation

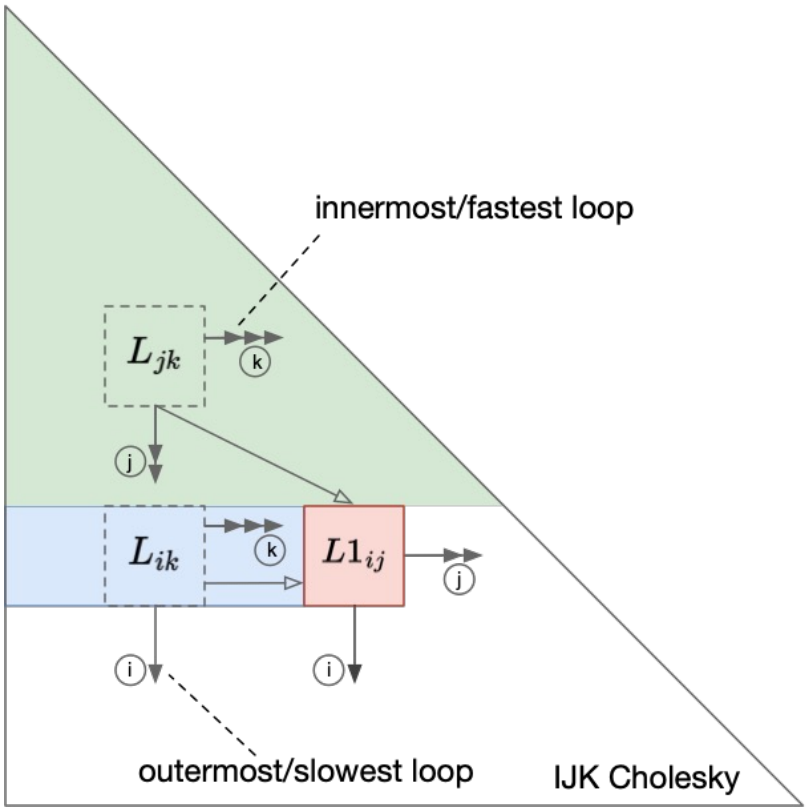
Optimizations shared and reused across different algorithms

Future: Tiling, wavefront parallelism, supernodal algorithms

Thank you

End

Visualizing Dependencies



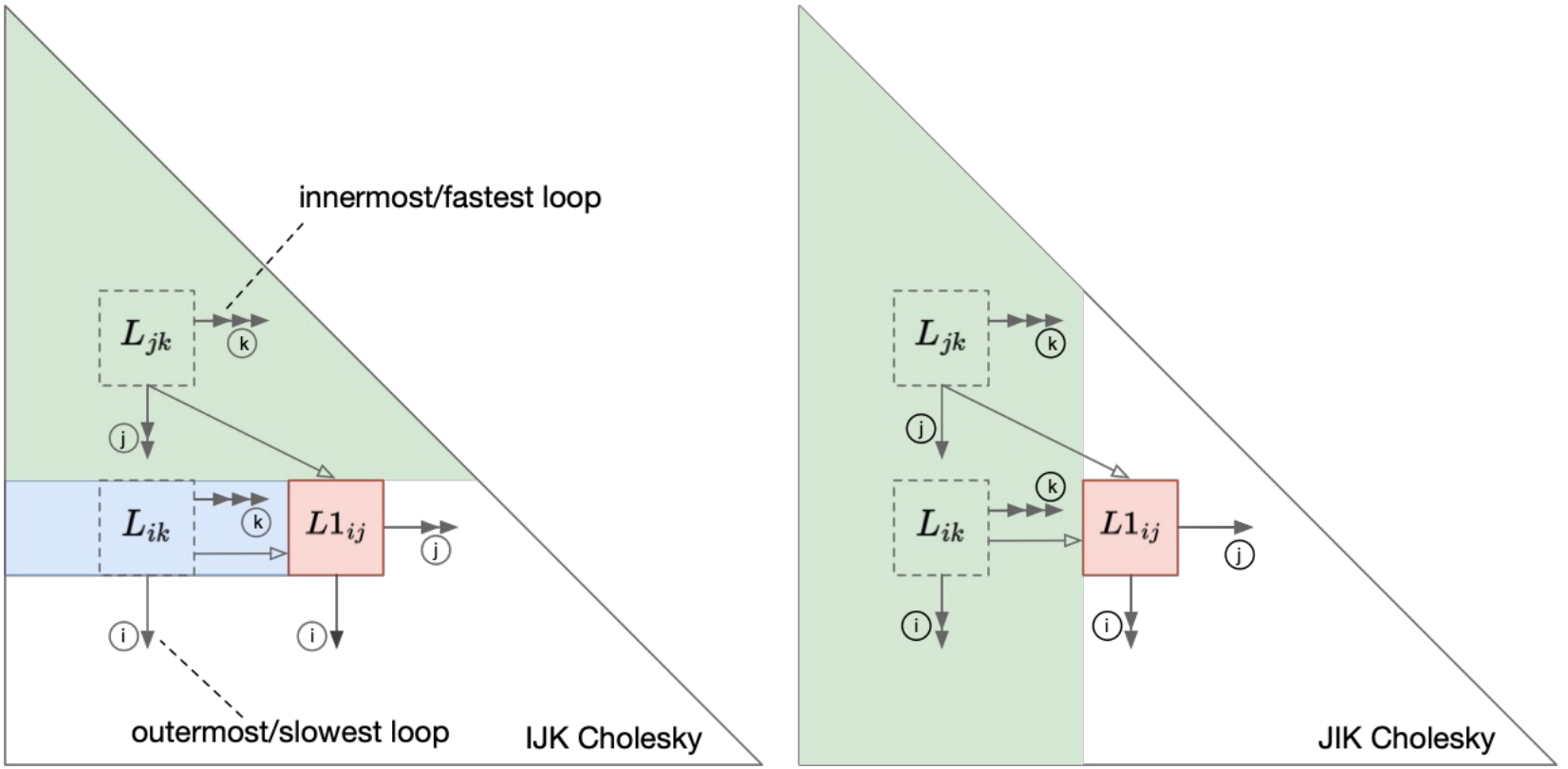
Cholesky Decomp:

$$L_{ij} = (A_{ij} - \sum L_{ik}L_{jk})/L_{jj}$$

$$L_{ij} = \sqrt{A_{ii} - \sum L_{ij}L_{ij}}$$

$$L1_{ij} = \sum_{k=0}^{j} L_{ik}L_{jk} : j < i$$

Visualizing Dependencies



Path to portability over data structures and machines

-show petsc diagram?

Recurrences: Two Dependencies

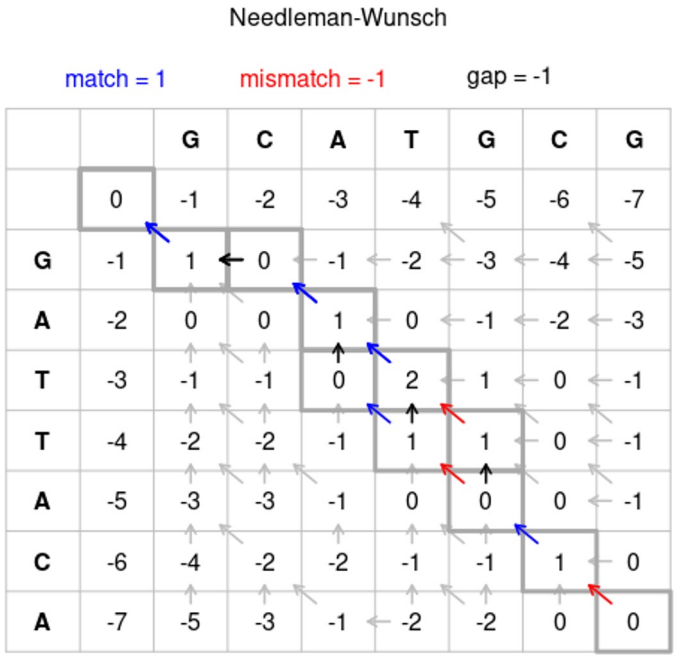
$$F_i = F_{i-1} + F_{i-2}$$

Recurrences: Multidimensional

$$
\begin{aligned}
N_{ij} = max(& \\
& N_{i-1,j-1} + (A_i == B_j), \\
& N_{i,j-1}, \\
& N_{i-1,j} \\
) &
\end{aligned}
$$

Recurrences: Multidimensional

$$N_{ij} = max(\\ N_{i-1,j-1} + (A_i == B_j), \\ N_{i,j-1}, \\ N_{i-1,j} \\ )$$



		G	C	A	T	G	C	G
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

Recurrences: Multiple Equations

$$L_{ij} = (A_{ij} - \sum_k^j L_{ik} L_{jk}) / L_{jj} \quad : k < j < i$$

$$L_{ij} = \sqrt{A_{ij} - \sum_k^j L_{ik} L_{jk}} \quad : k < j = i$$

Cholesky Decomposition

…LU Decomposition, QR Decomposition, Triangular Solve

Sparsity

$$T_{ij} = \max_k(T_{k,j-1} * A_{ki} * B_{ij})$$

$$X_i = \sum_j^n (A_{ij} * X_j)$$

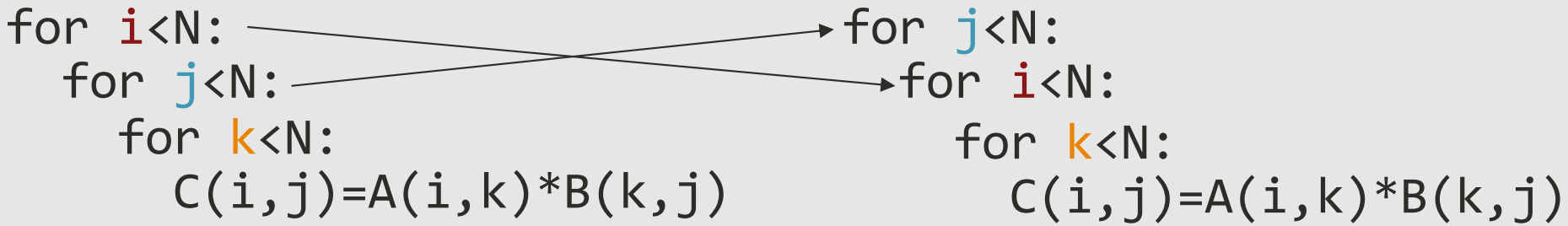
Common Software Optimizations

Loop Interchange

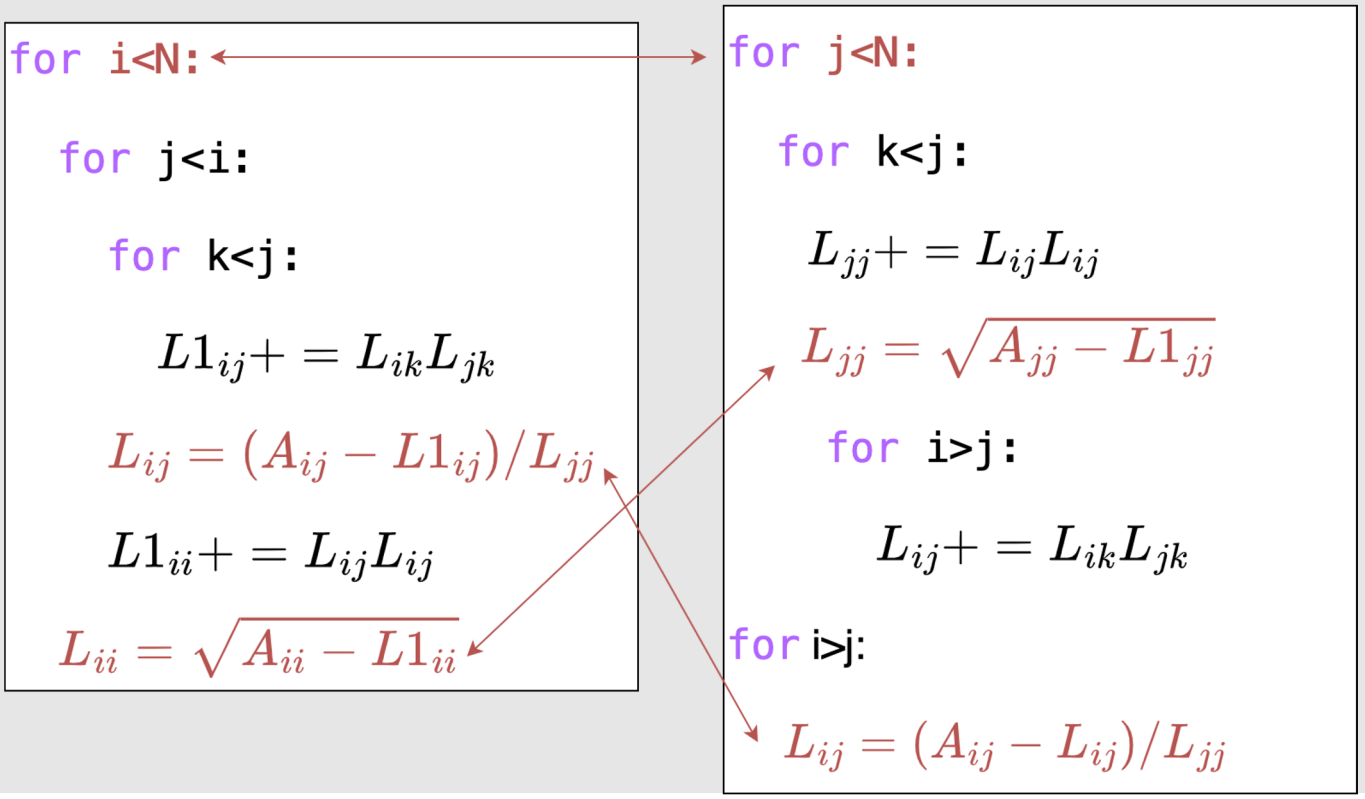
Loop Fusion

Parallelization

Optimizations: Not as trivial for recurrences

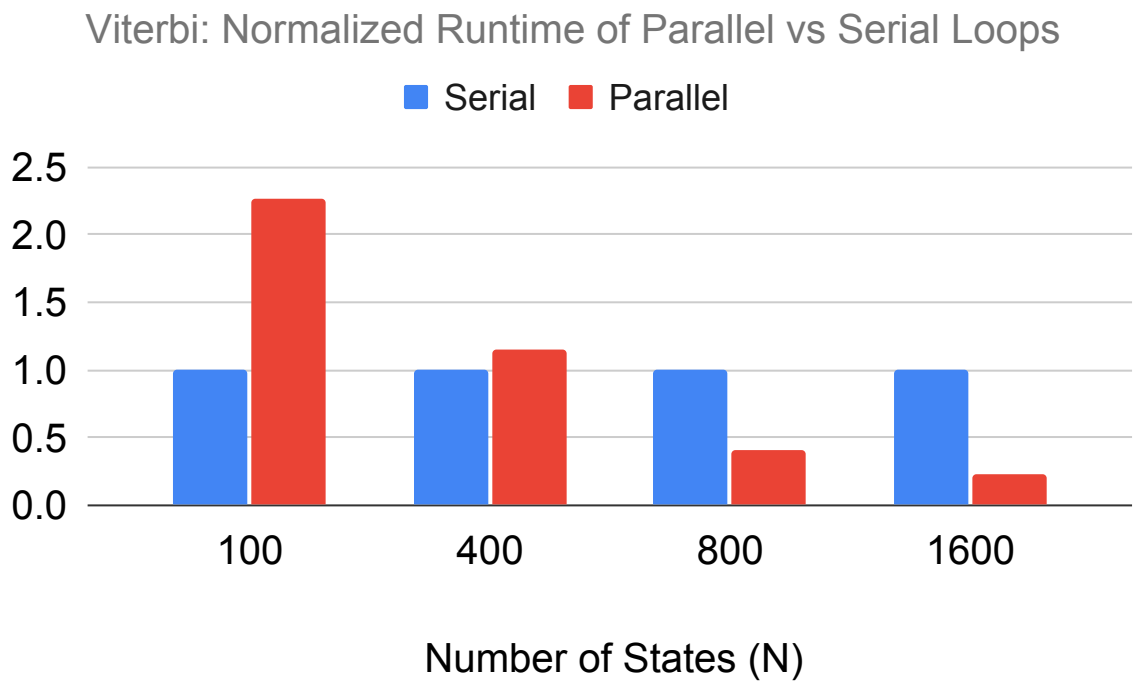


Loop Interchange For Recurrences (Cholesky)



Parallelization

Auto-parallelization of dense Viterbi equation



Recurrence Language

Cholesky Decomposition:

Solves Ax=b for certain A's

What needs to be added to Tensor Index Notation to express it?

Language: Tensor Index Notation ->Recurrences

Iteration Bounds:

k<N, j<N, i<N

$$L_{ij} = (A_{ij} - \sum_k^n B_{ik} B_{jk}) / B_{jj}$$

Language: Tensor Index Notation ->Recurrences

Constrain Iteration Bounds:

k < j < i <N

$$L_{ij} = (A_{ij} - \sum_{k}^{n} B_{ik} B_{jk}) / B_{jj}$$

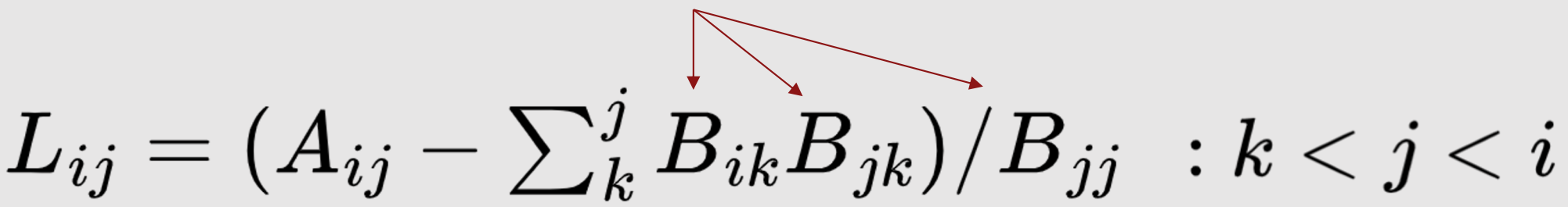
Language: Tensor Index Notation ->Recurrences

Make Computation in-place:
Replace B's with L's

$$L_{ij} = (A_{ij} - \sum_k^j B_{ik} B_{jk}) / B_{jj} \quad : k < j < i$$

Language: Tensor Index Notation ->Recurrences

Make Computation in-place:



$$L_{ij} = (A_{ij} - \sum_k^j B_{ik} B_{jk}) / B_{jj} \quad : k < j < i$$

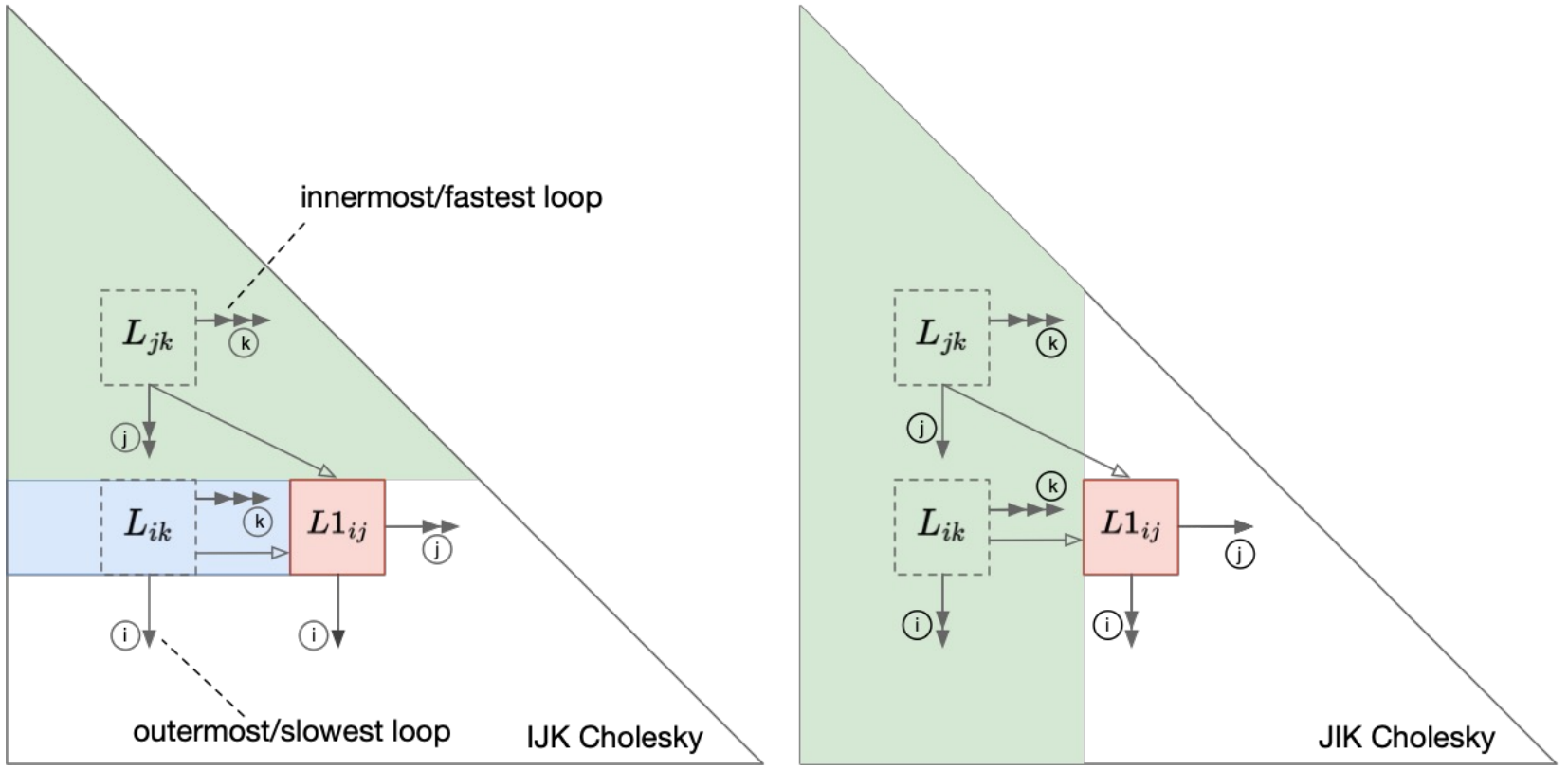
Language: Tensor Index Notation ->Recurrences

Multiple Equations

$$L_{ij} = (A_{ij} - \sum_k^j L_{ik} L_{jk}) / L_{jj} \quad : k < j < i$$

$$L_{ij} = \sqrt{A_{ij} - \sum_k^j L_{ik} L_{jk}} \quad : k < j = i$$

Different Loop Orders



Results: Loop Fusion

Code gen/placement algorithm is greedy → automatic loop fusion

$$Ax_1 = b_1$$

$$Ax_2 = b_2$$

1.92-1.98x speedup when fused