Small completion with code generation & scaling is included.

- Dependence matrix of transformed program: $L' D$
  - Legally: $L' D < 0$

  Matrix in which each column is a distance/direction vector

  Dependence matrix: $D'$ (a non-singular matrix)

  Transformation matrix: $L'$ (a non-singular matrix)

  *Theory for reordering applies to other loop transformations that can be modeled as linear transformations: scaling, reordering, etc.*

---

Let's base our work on what we did for HP's compiler, Product Line.

1. A method for improving compiler performance
2. Making a loop nest fully parallel

- Two key ideas:
  - Locality matrix: model for temporal and spatial locality
  - Composition of linear loop transformations: matrix representation

- Linear loop transformations: permutation, skewing

Overview of lecture

---

- Transformed code can be generated using ILP calculator
- Dependence matrix of the loop nest
- Legally of permutation can be determined from the higher transformation on the iteration space of the loop nest
- Dependence of perfectly nested loop can be modeled as a permutation
- Cache performance can be improved by timing and permutation

---

For Locality Enhancement

---

Leaf Loop Transformations
This is a pseudo-Hermitian normal form of matrix.

- Elements of \( \Omega \) are non-negative.
- Special form of this decomposition: \( \Omega = I \alpha \beta \) where \( \alpha \) and \( \beta \) form an orthogonal.
- Unit modulus matrices are the orthogonal matrices in reals.
- And once again a matrix decomposition: \( \Omega = I \alpha \beta \).
- We can also view this as a matrix decomposition: \( \Omega = I \alpha \beta \).
- We have used reduction to column-echelon form, \( \Omega = I \).

Some observations:

### Key Algorithms

1. Null space basis for null space of matrix in column-echelon form:

\[
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\]

2. Null space basis for null space of matrix in column-echelon form:

\[
\begin{bmatrix}
0 \\
1
\end{bmatrix}
\]

### Dependence betwen two iterations

\[
\begin{bmatrix}
X_0/Z \mid X_{1/2}
\end{bmatrix}
\]

- Dependence between two iterations:

\[
\begin{bmatrix}
0 & 1 \\
0 & 2 \\
0 & 1
\end{bmatrix}
\]

### Example:

\[
(1 \ 1 \ 1 \ \cdot \ 2^-) = ((1 \ 0 \ 0) \ \cdot \ M = \Omega = I)
\]

\[
\begin{bmatrix}
0 & 1 & 1 \\
0 & 2 & 0 \\
0 & 1 & 1
\end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 0 \\ 3 & 0 & 1 \end{bmatrix}
\]

Example: \( M \Omega = I \)
Both are easy to add to basic model as we will see below.

Spatial Locality

Interactions that read from the same location: Input dependences

There are other kinds of locality that are important.

This explores only one kind of temporal locality

For now, focus only on reuse opportunities between dependent iterations.

Loop skewing followed by loop reversal

Can we view this in terms of loop transformation?

Locality
- Dependent iterations are scheduled close together, so good for
- Transformation is local

Note:

One solution: schedule iterations along the degree sequence

Permutation and dilation are both illegal.

We have studied two transformations: permutation and dilation.
Skewing of inner loop by outer loop:

Always legal

New dependence vectors: compute $T \times D$

In this example, $D = T \times D$

This skewing has changed dependence vector but it has not brought dependent iterations closer together.

Outer loop skewing:

Skewing of outer loop by inner loop: not necessarily legal

In this example, $D = T \times D$

Incorrect

Dependent iterations are closer together (good) but program is illegal (bad).

How do we fix this?

Composition of linear transformations

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Composition of linear transformations is another linear transformation!

Composite transformation matrix is

$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

How do we synthesize this composite transformation?

In final program, dependent iterations are closer together!

Transformation: skewing followed by reversal
When about second row?
So first row of \( L \) can be chosen to be \( (1 \quad 0 \quad 0) \).
This says that row of \( L \) is orthogonal to
\[
\begin{pmatrix}
\ell_1 \\
\ell_2 \\
0
\end{pmatrix} = \begin{pmatrix} 1 & -1 \end{pmatrix}
\]
So \( L \) is self-adjoint.

So dependence vector in transformed program should look like
\[
\begin{pmatrix}
\ell_1 \\
\ell_2 \\
0
\end{pmatrix} = \begin{pmatrix} 1 & -1 \end{pmatrix}
\]

Dependence vector is

Higher reduction: move reuse into inner loops

\( q > \gamma_{21} \) \( \gamma \in J \) occurs \( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)

\( \gamma \text{ A B C D} \) \( \gamma \text{ E F G} \) \( \gamma \text{ H I J K} \) \( \gamma \text{ L M N O} \) \( \gamma \text{ P Q R S} \) \( \gamma \text{ T U V W} \) \( \gamma \text{ X Y Z} \)
With 0's in the bottom row
algorithm for dimension vectors, and pad remaining null space vectors
Consider plane, knock out rows of \( D \) with direction entries, use
Pad with zeros to get partial transformation: \( \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \)
remaining rows of \( D \)
Ignore last 2 and pad rows of \( D \) and find basis for null space of
If this is orthogonal to \( D \), it is clear that \( T \) and \( T^\perp \) must be zero.
Let first row of transformation be \( \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \)
Consider \( D \)
Exclusion to direction vectors: easy

2. How do we fill in the rest of the matrix to get unimodular matrix? How do we push rows down?

1. How do we choose the first few rows of the transformation matrix?

Example: study running example in previous slides.

Transformation matrix
- Use transpose of basis vectors as first few rows of
- \( D \) be a basis for null space of \( D^\perp \)

Algorithm:
For now, assume dependence matrix \( D \) has only distances in it
Choose first few rows of transformation matrix

Example:
- a general non-square matrix
- Problem: how do we generate code if transformation matrix is
- First few rows of this matrix
- Consider procedure: generate a non-singular matrix \( D \)
- Turn out to be easier if we drop unimodularity requirement
- Filling in rest of matrix
\[
\begin{pmatrix}
\hat{e} & \hat{e} & \hat{e} \\
+ & + & 1 \\
0 & 0 & 0
\end{pmatrix}
\]

New partial dependence matrix is
\[
(0 \ 0 \ 1) = \text{Next row of transformation}
\]
Next row of transformation is added by partial transformation.
\[
(0 \ 0 \ 0) = \begin{pmatrix}
1 & - & 0 & 1 & - \\
1 & 0 & 1 & 0 & 1 & 0
\end{pmatrix}
\]

\[
\text{Example:}
\]

<table>
<thead>
<tr>
<th>0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Completion Procedure

1. Repeat from step 1, zero every other entry as the next row of the transformation.
2. Otherwise, let \( n \) be the first row of (reduced) \( D \) that has a non-zero entry. Use \( n \) with row vector with 1 in column 1 and 0 elsewhere. Let \( n^j \) be the row vector with 1 in column 0 and 0 elsewhere. Use \( n \) in place of \( n^j \) in the non-vanishing columns of \( D \).
3. If \( D \) is now empty, use null space basis determination to add \( D \). Defer all dependences for which \( pD \neq 0 \).

Completion Algorithm: Iterative Algorithm

<table>
<thead>
<tr>
<th>0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
</tr>
</tbody>
</table>

No dependences are now satisfied, so we are left with the problem.
non-singular matrices that are not unimodular.

Example: How do we know this is legal, and is good for

Let $A$ be a transformation matrix

Develop code generation technique for non-singular matrices.

Two solutions: based on $I - L$ pseudo-Hermitian normal form

may not be unimodular.

Problem: Transformation matrix $I$ is optimal in non-singular but

Key problem here: $I^{-1}$ is not necessarily an integer matrix

New bounds: compute from non-transformation

Transformation: $T \cdot \Omega$

q $\geq T_1 \cdot \Omega \cdot V$.

Code generation for non-unimodular matrix $I$.

Code generation with a non-singular transformation $I$.

(0 0)

(0 1)

Search matrix: non-unimodular transformation

\[
\begin{align*}
10 & \text{ } 0 (y/2) \\
1 & -1 \\
10 & = 2.200.2
\end{align*}
\]

Loop search change step size of loop.
How do we determine step sizes?
(Requiring bounds is not correct)

Key Problems:

How do we determine inner lower and upper bounds?

\[
A(4I-2J+3, I+J) = J;
\]

DO I = 1, 3

DO J = 1, 3

DO V = -U/2 + 3 max(1, ceil(u/2 + 1/2))

DO U = -2, 10, 2

Auxiliary Iteration Space

Initial Iteration Space

Final Iteration Space

Diffusion example: Heat equation normal form decomposition \( P \rightarrow J \)

Ligne non-singular matrix \( P \) transformation matrix.

Ligne iterates and \( P \) are unitary.

Ligne iterates is \( P \) a unitary.

Ligne iterates does not apply.

Ligne iterates neither unitary nor singular.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.

Ligne iterates: Unitary is not important.
The memory system behavior of a full program
memory system behavior of a full program is same as
orthogonally normal in that space.

Auxiliary space relocations are performed in same order as
vector and show that d is lexicographically positive as well.

Proof: Consider the vector \( \mathbf{v} \) where \( \mathbf{v} \ast \mathbf{T} = \mathbf{p} \) where \( \mathbf{v} \) is a lexicographically positive vector.

Initial: \( \mathbf{T} = 1 \times n \) and \( \mathbf{p} = (1, 2, 3, \ldots, n) \) where the elements of \( \mathbf{p} \) are done.

Lemma: If \( \mathbf{T} \) is a \( n \times n \) and \( \mathbf{p} \) is a lexicographically positive vector.

Auxiliary space relocations into

corners, \( \mathbf{T} \) maps lexicographically positive vectors into

corners.\n
Solution: \( \mathbf{T} \) is a non-singular with non-singular transformation

General picture

(adjacent dependence) + spatial locality + read-only data reuse

Gaussian picture

\[
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}
\]

Full transformation vector.

Orthogonality is maintained by parallelism. We would add row (0).

Applying the full transformation to parallel transformations to
orthogonally normal in that space.

Auxiliary space relocations are performed in same order as

Proof: Consider the vector \( \mathbf{v} \) where \( \mathbf{v} \ast \mathbf{T} = \mathbf{p} \) where \( \mathbf{v} \) is a lexicographically positive vector.

Initial: \( \mathbf{T} = 1 \times n \) and \( \mathbf{p} = (1, 2, 3, \ldots, n) \) where the elements of \( \mathbf{p} \) are done.

Lemma: If \( \mathbf{T} \) is a \( n \times n \) and \( \mathbf{p} = (1, 2, 3, \ldots, n) \) where the elements of \( \mathbf{p} \) are done.

Auxiliary space relocations into

corners, \( \mathbf{T} \) maps lexicographically positive vectors into

corners.\n
Solution: \( \mathbf{T} \) is a non-singular with non-singular transformation

General picture

(adjacent dependence) + spatial locality + read-only data reuse

Gaussian picture

\[
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}
\]
Spatial locality matrix is:

\[
\begin{pmatrix}
1 & 0 \\
1 & 0 \\
0 & 1
\end{pmatrix}
\]

Dimension of \( A \) for the most part

Example:

\[
\begin{pmatrix}
0 & 0 \\
1 & 0 \\
0 & 1
\end{pmatrix}
\]

So spatial locality matrix is:

\[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\]

Transport matrix is:

\[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\]

Reference matrix is:

\[
(I') X^*(I') \lambda = (I') \lambda
\]

For \( I' = I \),

\[
(I') X^*(I')\lambda = (I')\lambda
\]

Matrix access:

Example from Assignment 1: exploiting spatial locality in \( y \).

Spatial locality matrix containing a basis for null space of \( A \):

Solution: \( \lambda = \text{null} \text{ of } A \).

Let \( \lambda = \text{null} \text{ of } A \).

We want these two access in the same column of \( X \).

\[
\begin{pmatrix}
0 \\
\vdots \\
0 \\
\end{pmatrix} = (I') \lambda
\]

Example array access in column-major order, reference \( X^* \).

Example array access in column-major order, reference \( X^* \).

Example array access in column-major order, reference \( X^* \).
\[
\begin{pmatrix}
  + & 0 & 0 \\
  0 & 0 & + \\
  + & 0 & 0
\end{pmatrix}
\]

Considering all the references, we get the following locality matrix:
\[
\begin{pmatrix}
  0 \\
  + \\
  0
\end{pmatrix}
\]

- House vector is

That is, we would perform the loops which are correct
\[
\begin{pmatrix}
  1 \\
  0 \\
  1
\end{pmatrix} = \begin{pmatrix}
  0 & 1 \\
  1 & 0
\end{pmatrix}
\]

For our example, we get the transformation matrix

Performing spatial locality: perform heap reduction on locality

\[
\begin{pmatrix}
  1 & 0 \\
  0 & 1
\end{pmatrix}
\]

Vectors and spatial locality vectors from consideration

be good to drop some of the more important read-only reuse

Precondition: It will not pass through only the zero vector in the inner

matrix and use the unimportant parts as transformation.

Do a pseudo-Heapsort on decomposition of transformation

Call the decomposition algorithm to generate a complete

Transpose the first few rows of the transformation.

Determine a basis for the null space of the transformation.

read-only data reuse + spatial locality

which we would like to exploit locally (dependence +

Compute the locality matrix \( T \) which contains all vectors above

General Algorithm for Heap Reduction

Consider tense vectors for reference \( A' \):

\[
C(1, j) = C(1, j) + A(1, k)*B(k, j)
\]

DO i = 1 TO n
DO j = 1 TO n
DO k = 1 TO n

Read-only data reuse: add reuse directions to locality matrix
When we can, how do we do it?

Perrinable loop nest

Can we always convert a perfectly nested loop into a fully perrinable loop nest?

Theorem: If all dependence vectors are distance vectors, we can convert each loop into a fully perrinable loop nest.

Example: We want to exploit locality in both matrix A and vectors X. We can still exploit locality by localizing the loops.

In some cases, helping reduction may fail since locality vectors span entire space.

Example: Given an upper triangular matrix A:

$$\begin{bmatrix}
1 & 1 & 1 \\
0 & 1 & 1 \\
0 & 0 & 1
\end{bmatrix}$$

Second row of transformation (in 1) for any in < 0.

So first row of transformation can be (1, 0).

Theorem: Perrinable loops in transformed program must have all positive dependence matrix entries.

Example: Matrix is NOT perrinable.

**Proof:**

When we can, how do we do it?
pairs must be compared to number of integer points.

Locality enhancement except at boundaries (out boundary)
Not as nice as helpful reduction solution, but it will work fine for

1. Find negative a 2 deep loop nest.

Result of this flattened warport

2. [Diagram of loop nest]

3. [Diagram of loop nest]

[Transformation equations]

For our example, multiple of row a = \( \text{ceiling}(n/p_2) \)
rows \( a, b, \ldots \)

1. Find the first positive number in the corresponding column
for each negative entry in the first row with negative entries,

Original loop Tiled fully permutable loop

1 2 3 5 4

\[ \begin{pmatrix} 1 & 2 & 3 & 5 & 4 \\ \end{pmatrix} \]

1. Otherwise, if step (1):

2. If no negative entries, done.

2. Generalize new dependence matrix

1. Apply algorithm on previous slice to first row with

Non-negative entries

If all entries in dependence matrix are non-negative, done.

General algorithm for making loop nest fully permutable
Second band of fully permutable loops:

In this case, all dependences are satisfied, so last two loops form dependencies, and second band cannot make band any bigger, so terminate band, drop all satisfied dependencies, and start second band.

\[
\begin{pmatrix}
  + & - & + \\
  - & + & - \\
  7 & 3 & 0 \\
  1 & 3 & 1 \\
  0 & 1 & 0 \\
\end{pmatrix}
\]

Example for general algorithm:

Second row has direction which cannot be knocked out. But we can interchange fifth row with second row to continue band.

\[
\begin{pmatrix}
  + & - & + \\
  - & + & - \\
  1 & -3 & -2 \\
  - & + & + \\
  0 & 1 & 0 \\
\end{pmatrix}
\]

Proof sketch: \( \Omega \) is your transformation matrix. At the end, \( \text{inv}(L) \) is your transformation matrix. Apply same transformation to \( L \).

Every time you apply a transformation to the dependency matrix, if you have \( p \) loops, start with \( n \times p \) matrix. How do we determine transformation matrix for performing this?
For the previous example, we get

\[
\begin{pmatrix}
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

After the second and third rows and

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]
6. The full permutation-based approach achieves the same performance and parallelism as the original 1-dimensional approach, but at the cost of increased communication overhead.

2. Permute the data before the decomposition.

4. Otherwise, apply the algorithm to convert to bands of fully permutable blocks.

If not, then:

1. If space is too small, you have the option of dropping the data.

2. Perform a height reduction by finding a base for full space of

I. Compute locally.

II. Compute locally.

III. Compute locally.

IV. Compute locally.

V. Compute locally.

VI. Compute locally.

VII. Compute locally.

VIII. Compute locally.

IX. Compute locally.

The overall strategy is:

1. Compute locally.

2. Compute locally.

3. Compute locally.

4. Compute locally.

5. Compute locally.

6. Compute locally.

7. Compute locally.

8. Compute locally.

9. Compute locally.

10. Compute locally.

11. Compute locally.

12. Compute locally.

13. Compute locally.


The full permutation-based approach achieves the same performance and parallelism as the original 1-dimensional approach, but at the cost of increased communication overhead.

2. Permute the data before the decomposition.

4. Otherwise, apply the algorithm to convert to bands of fully permutable blocks.

If not, then:

1. If space is too small, you have the option of dropping the data.

2. Perform a height reduction by finding a base for full space of

I. Compute locally.

II. Compute locally.

III. Compute locally.

IV. Compute locally.

V. Compute locally.

VI. Compute locally.

VII. Compute locally.

VIII. Compute locally.

IX. Compute locally.

X. Compute locally.

XI. Compute locally.

XII. Compute locally.

XIII. Compute locally.

XIV. Compute locally.

XV. Compute locally.

XVI. Compute locally.

XVII. Compute locally.

XVIII. Compute locally.

XIX. Compute locally.

XX. Compute locally.

XXI. Compute locally.

XXII. Compute locally.

XXIII. Compute locally.

XXIV. Compute locally.

XXV. Compute locally.

XXVI. Compute locally.

XXVII. Compute locally.

XXVIII. Compute locally.

XXIX. Compute locally.

XXX. Compute locally.

XXXI. Compute locally.

XXXII. Compute locally.

XXXIII. Compute locally.

XXXIV. Compute locally.

XXXV. Compute locally.

XXXVI. Compute locally.

XXXVII. Compute locally.

XXXVIII. Compute locally.

XXXIX. Compute locally.

XL. Compute locally.

XLI. Compute locally.

XLII. Compute locally.

XLIII. Compute locally.

XLIV. Compute locally.

XLV. Compute locally.

XLVI. Compute locally.

XLVII. Compute locally.

XLVIII. Compute locally.

XLIX. Compute locally.

L. Compute locally.

II. Compute locally.

III. Compute locally.

IV. Compute locally.

V. Compute locally.

VI. Compute locally.

VII. Compute locally.

VIII. Compute locally.

IX. Compute locally.

X. Compute locally.

XI. Compute locally.

XII. Compute locally.

XIII. Compute locally.

XIV. Compute locally.

XV. Compute locally.

XVI. Compute locally.

XVII. Compute locally.

XVIII. Compute locally.

XIX. Compute locally.

XX. Compute locally.

XXI. Compute locally.

XXII. Compute locally.

XXIII. Compute locally.

XXIV. Compute locally.

XXV. Compute locally.

XXVI. Compute locally.

XXVII. Compute locally.

XXVIII. Compute locally.

XXIX. Compute locally.

XXX. Compute locally.

XXXI. Compute locally.

XXXII. Compute locally.

XXXIII. Compute locally.

XXXIV. Compute locally.

XXXV. Compute locally.

XXXVI. Compute locally.

XXXVII. Compute locally.

XXXVIII. Compute locally.

XXXIX. Compute locally.

XL. Compute locally.

XLI. Compute locally.

XLII. Compute locally.

XLIII. Compute locally.

XLIV. Compute locally.

XLV. Compute locally.

XLVI. Compute locally.

XLVII. Compute locally.

XLVIII. Compute locally.

XLIX. Compute locally.

L. Compute locally.

II. Compute locally.

III. Compute locally.

IV. Compute locally.

V. Compute locally.

VI. Compute locally.

VII. Compute locally.

VIII. Compute locally.

IX. Compute locally.

X. Compute locally.

XI. Compute locally.

XII. Compute locally.

XIII. Compute locally.

XIV. Compute locally.

XV. Compute locally.

XVI. Compute locally.

XVII. Compute locally.

XVIII. Compute locally.

XIX. Compute locally.

XX. Compute locally.

XXI. Compute locally.

XXII. Compute locally.

XXIII. Compute locally.

XXIV. Compute locally.

XXV. Compute locally.

XXVI. Compute locally.

XXVII. Compute locally.

XXVIII. Compute locally.

XXIX. Compute locally.

XXX. Compute locally.

XXXI. Compute locally.

XXXII. Compute locally.

XXXIII. Compute locally.

XXXIV. Compute locally.