# Loop Parallelism I

Cosmin E. Oancea and Troels Henriksen
`[cosmin.oancea,athas]@diku.dk`

Department of Computer Science (DIKU)
University of Copenhagen

September 2015 PMPH Lecture Notes

# Course Organization

| W | HARDWARE | | SOFTWARE | LAB/CUDA |
|---|---|---|---|---|
| 1 | Trends Vector Machine | ⟵ | List HOM (Map-Reduce) | Intro & Simple Map Programming |
| 2 | In Order Processor | ⟶<br>⟵ | VLIW Instr Scheduling | Scan & Reduce |
| 3 | Cache Coherence | | Loop Parallelism I | Sparse Vect Matrix Mult |
| 4 | Interconnection Networks | | Case Studies & Optimizations | Transpose & Matrix Matrix Mult |
| 5 | Memory Consistency | | Optimising Locality | Sorting & Profiling & Mem Optimizations |
| 6 | OoO, Spec Processor | | Thread-Level Speculation | Project Work |

Three narative threads: the path to complex & good design:

- Design Space tradeoffs, constraints, common case, trends.
- Reasoning: from simple to complex, Applying Concepts.

# Motivation

+ So far we reasoned about how to parallelize a known algorithm
+ using a clean, functional approach, e.g., flattening,
+ which provides work and depth guarantees,
- but does NOT account for locality of reference.

Why do we have to look at imperative loops?

# Motivation

+ So far we reasoned about how to parallelize a known algorithm
+ using a clean, functional approach, e.g., flattening,
+ which provides work and depth guarantees,
- but does NOT account for locality of reference.

Why do we have to look at imperative loops?

- A lot of legacy sequential imperative code, C++/Java/Fortran.
- Need to parallelize the implementation of unknown algorithm,
- Need to optimize parallelism, e.g., locality of reference requires subscript analysis.

1. Direction-Vector Analysis
2. Block Tiling: Matrix Multiplication Case Study
3. Coalesced Accesses: Matrix Transposition Case Study
4. Non-Trivial Synchronization: Histogram Case Study
5. Known Recurrences: Tridiagonal Solver Case Study
6. Imperative Context: Summarization of Array Indexes

# Problem Statement

## Three Loop Examples

```
DO i = 1, N             DO i = 2, N                 DO i = 2, N
  DO j = 1, N             DO j = 2, N                 DO j = 1, N
    A[j,i] = A[j,i] ..      A[j,i] = A[j-1,i-1]...        A[i,j] = A[i-1,j+1]...
  ENDDO                     B[j,i] = B[j-1,i]...      ENDDO
ENDDO                   ENDDO ENDDO                 ENDDO
```

Iterations are ordered *lexicographically*, i.e., in the order they occur in the sequential execution, e.g., $\vec{k} = (i{=}2, j{=}4) < \vec{l} = (i{=}3, j{=}3)$.

- Which of the three loop nests is amenable to parallelization?
- Loop interchange is one of the most simple and useful code transformations, e.g., used to enhance locality of reference, parallel-loop granularity, and even to “create” parallelism.
- In which loop nest is it safe to interchange the loops?

# Definition of a Dependency

## Load-Store Classification of Dependencies

```
True Dependency (RAW)    Anti Dependency (WAR)    Output dependency (WAW)
S1    X  = ..            S1    .. = X             S1    X = ...
S2    .. = X             S2    X  = ..            S2    X = ...
```

**Th. Loop Dependence:** There is a dependence from statement $S1$ to $S2$ in a loop nest *iff* $\exists$ iterations $\vec{k}$, $\vec{l}$ such that:

# Definition of a Dependency

## Load-Store Classification of Dependencies

```
True Dependency (RAW)     Anti Dependency (WAR)     Output dependency (WAW)
S1     X  = ..            S1     .. = X             S1     X = ...
S2     .. = X             S2     X  = ..            S2     X = ...
```

**Th. Loop Dependence:** There is a dependence from statement $S1$ to $S2$ in a loop nest *iff* $\exists$ iterations $\vec{k}$, $\vec{l}$ such that:

1. $\vec{k} < \vec{l}$ or $\vec{k} = \vec{l}$ and $\exists$ an execution path from statement $S1$ to statement $S2$ such that:
2. $S1$ accesses memory location $M$ on iteration $\vec{k}$, and
3. $S2$ accesses memory location $M$ on iteration $\vec{l}$, and
4. one of these accesses is a write.

We say that $S1$ is the source and $S2$ is the sink of the dependence, because $S1$ executes before $S2$ in the sequential program execution. Dependence depicted with an arrow pointing from source to sink.

# Definition of a Dependency

## Load-Store Classification of Dependencies

```
True Dependency (RAW)      Anti Dependency (WAR)      Output dependency (WAW)
S1     X  = ..             S1     .. = X              S1     X = ...
S2     .. = X              S2     X  = ..             S2     X = ...
```

**Th. Loop Dependence:** There is a dependence from statement $S1$ to $S2$ in a loop nest *iff* $\exists$ iterations $\vec{k}$, $\vec{l}$ such that:

1. $\vec{k} < \vec{l}$ or $\vec{k} = \vec{l}$ and $\exists$ an execution path from statement $S1$ to statement $S2$ such that:
2. $S1$ accesses memory location $M$ on iteration $\vec{k}$, and
3. $S2$ accesses memory location $M$ on iteration $\vec{l}$, and
4. one of these accesses is a write.

We say that $S1$ is the source and $S2$ is the sink of the dependence, because $S1$ executes before $S2$ in the sequential program execution. Dependence depicted with an arrow pointing from source to sink.

We are most interested in cross iteration dependencies, i.e., $\vec{k} < \vec{l}$. Intra iteration dependencies, i.e., $\vec{k} = \vec{l}$ are analysed for ILP.

# Loop-Nest Dependencies

*Lexicographic ordering*, e.g., $\vec{k} =$(i=2,j=4) < $\vec{l} =$(i=3,j=3).

## Three Loop Examples

```
DO i = 1, N             DO i = 2, N                 DO i = 2, N
  DO j = 1, N             DO j = 2, N                 DO j = 1, N
    A[j,i] = A[j,i] ..      A[j,i] = A[j-1,i-1]...        A[i,j] = A[i-1,j+1]...
  ENDDO                     B[j,i] = B[j-1,i]...      ENDDO
ENDDO                   ENDDO ENDDO                 ENDDO
```

# Loop-Nest Dependencies

*Lexicographic ordering*, e.g., $\vec{k}$ =(i=2,j=4) < $\vec{l}$ =(i=3,j=3).

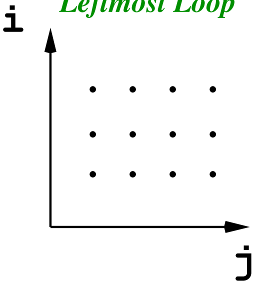
## Three Loop Examples

```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

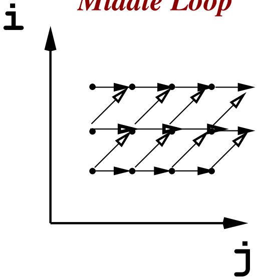
```
DO i = 2, N
  DO j = 2, N
    A[j,i] = A[j-1,i-1]...
    B[j,i] = B[j-1,i]...
ENDDO ENDDO
```

```
DO i = 2, N
  DO j = 1, N
      A[i,j] = A[i-1,j+1]...
  ENDDO
ENDDO
```

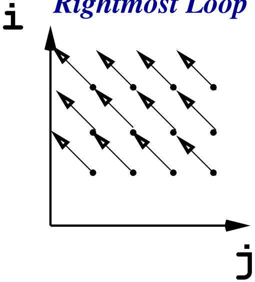
***Leftmost Loop***



***Middle Loop***



***Rightmost Loop***



How can I summarize this information?

# Aggregate Dependencies via Direction Vectors

## Write the Direction Vectors for Each Loop:

```
  DO i = 1, N             DO i = 2, N                  DO i = 2, N
    DO j = 1, N             DO j = 2, N                  DO j = 1, N
S1    A[j,i]=A[j,i]..   S1   A[j,i]=A[j-1,i]...     S1    A[i,j]=A[i-1,j+1]...
    ENDDO               S2   B[j,i]=B[j-1,i-1]...        ENDDO
  ENDDO                   ENDDO ENDDO                  ENDDO
```

Dependencies depicted via an edge *from* the stmt that executes first in the loop nest, i.e., *the source*, *to* the one that executes later, *the sink*.

**Def. Dependence Direction:** Assume $\exists$ a dependence from $S1$ in iteration $\vec{k}$ to $S2$ in $\vec{l}$ ($\vec{k} \leq \vec{l}$). *Dependence-direction vector* $\vec{D}(\vec{k}, \vec{l})$:

1. $\vec{D}(\vec{k}, \vec{l})_m =$ “<” if $\vec{k}_m < \vec{l}_m$,
2. $\vec{D}(\vec{k}, \vec{l})_m =$ “=” if $\vec{k}_m = \vec{l}_m$,
3. $\vec{D}(\vec{k}, \vec{l})_m =$ “>” if $\vec{k}_m > \vec{l}_m$.

If the source is a write and the sink a read then RAW dependency, iIf the source is a read then WAR, if both are writes then WAW.

# Parallelism and Loop Interchange

## Direction Vectors/Matrix for Three Loops

```
  DO i = 1, N
    DO j = 1, N
S1    A[j,i]=A[j,i]..
    ENDDO
  ENDDO
For S1→S1:
    (j1,i1)=(j2,i2)
    i1 = i2 & j1 = j2

Direction matrix:
S1→S1: [=,=]
```

```
  DO i = 2, N
    DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
    ENDDO
  ENDDO
S1→S1: (j1,i1)=(j2-1,i2)
        i1 = i2 & j1 < j2
S2→S2: (j1,i1)=(j2-1,i2-1)
        i1 < i2 & j1 < j2
S1→S1: [=,<]
S2→S2: [<,<]
```

```
  DO i = 2, N
    DO j = 1, N
S1    A[i,j]=A[i-1,j+1]...
    ENDDO
  ENDDO
For S1→S1:
    (i1,j1) = (i1-1,j2+1)
    i1 < i2 & j1 > j2

Direction matrix:
S1→S1: [<,>]
```

**Th. Parallelism:** A loop in a loop nest is parallel *iff* all its directions are either = or there exists an outer loop whose corresp. direction is <.

A direction vector cannot have $>$ as the first non-$=$ symbol, as that would mean that I depend on something in the future.

# Parallelism and Loop Interchange

## Direction Vectors/Matrix for Three Loops

```
  DO i = 1, N
    DO j = 1, N
S1    A[j,i]=A[j,i]..
    ENDDO
  ENDDO
For S1→S1:
    (j1,i1)=(j2,i2)
    i1 = i2 & j1 = j2

Direction matrix:
S1→S1: [=,=]
```

```
  DO i = 2, N
    DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
    ENDDO
  ENDDO
S1→S1: (j1,i1)=(j2-1,i2)
        i1 = i2 & j1 < j2
S2→S2: (j1,i1)=(j2-1,i2-1)
        i1 < i2 & j1 < j2
S1→S1: [=,<]
S2→S2: [<,<]
```

```
  DO i = 2, N
    DO j = 1, N
S1    A[i,j]=A[i-1,j+1]...
    ENDDO
  ENDDO
For S1→S1:
    (i1,j1) = (i1-1,j2+1)
    i1 < i2 & j1 > j2

Direction matrix:
S1→S1: [<,>]
```

**Th. Loop Interchange:** A column permutation of the loops in a loop nest is legal *iff* permuting the direction matrix in the same way *does NOT result* in a > direction as the leftmost non-= direction in a row.

# Parallelism and Loop Interchange

## Direction Vectors/Matrix for Three Loops

```
  DO i = 1, N
    DO j = 1, N
S1    A[j,i]=A[j,i]..
    ENDDO
  ENDDO

For S1→S1: j1 = j2
           i1 = i2
(i2,j2)-(i1,j1)=
[=,=]
```

```
  DO i = 2, N
    DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
  ENDDO ENDDO

For S1→S1: j1 = j2-1
           i1 = i2
  (i2,j2)-(i1,j1)=[=,<]
For S2→S2: j1 = j2-1
           i1 = i2-1
  (i2,j2)-(i1,j1)=[<,<]
```

```
  DO i = 2, N
    DO j = 1, N
S1    A[i,j]=A[i-1,j+1]...
    ENDDO
  ENDDO

  For S1→S1: i1 = i2-1
             j1 = j2+1
  (i2,j2)-(i1,j1)=[<,>]
```

Interchange is safe for the first and second nests, but not for the third!

e.g., [=,<] → [<,=] (for the second loop nest)
[<,<] [<,<]

# Parallelism and Loop Interchange

## Direction Vectors/Matrix for Three Loops

```
  DO i = 1, N
    DO j = 1, N
S1    A[j,i]=A[j,i]..
    ENDDO
  ENDDO

For S1→S1: j1 = j2
           i1 = i2
(i2,j2)-(i1,j1)=
[=,=]
```

```
  DO i = 2, N
    DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
  ENDDO ENDDO

For S1→S1: j1 = j2-1
           i1 = i2
  (i2,j2)-(i1,j1)=[=,<]
For S2→S2: j1 = j2-1
           i1 = i2-1
  (i2,j2)-(i1,j1)=[<,<]
```

```
  DO i = 2, N
    DO j = 1, N
S1    A[i,j]=A[i-1,j+1]...
    ENDDO
  ENDDO

   For S1→S1: i1 = i2-1
              j1 = j2+1
   (i2,j2)-(i1,j1)=[<,>]
```

Interchange is safe for the first and second nests, but not for the third!

e.g., [=,<] → [<,=] (for the second loop nest)
[<,<] [<,<]

After interchange, loop *j* of the second loop nest is parallel.

**Corollary: A parallel loop can be always interchanged inwards.**

# Dependency Graph and Loop Distribution

**Def. Dependency Graph:** edges from the source of the dependency, i.e., early iteration, to the sink, i.e., later iteration.

**Th. Loop Distribution:** Statements that are in a dependence cycle remain in one (sequential) loop. The others are distributed to separate loops in graph order; if no cycle then parallel loops.

## Vectorization Example: Remember Vector Machines?

```
  DO i = 3, N
S1  A[i] = B[i-2] ...
S2  B[i] = B[i-1] ...
  ENDDO

For S2→S1: i1 = i2-2, [<]
For S2→S2: i1 = i2-1, [<]
```

# Dependency Graph and Loop Distribution

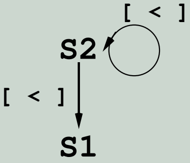
**Def. Dependency Graph:** edges from the source of the dependency, i.e., early iteration, to the sink, i.e., later iteration.

**Th. Loop Distribution:** Statements that are in a dependence cycle remain in one (sequential) loop. The others are distributed to separate loops in graph order; if no cycle then parallel loops.

### Vectorization Example: Remember Vector Machines?

```
  DO i = 3, N
S1  A[i] = B[i-2] ...
S2  B[i] = B[i-1] ...
  ENDDO

For S2→S1: i1 = i2-2, [<]
For S2→S2: i1 = i2-1, [<]
```



```
  DO i = 3, N
S2  B[i] = B[i-1] ...
  ENDDO

  DOALL i = 3, N
S1  A[i] = B[i-2] ...
  ENDDOALL
```

**Corollary:** It is always legal to distribute a parallel loop; but requires array expansion for local variables or if output dependencies are present.

# Loop Distribution May Require Array Expansion

```
float tmp;
for(i=2; i<N; i++) {
  tmp = 2*B[i-2];
  A[i] = tmp;
  B[i] = tmp+B[i-1]
}
```

```
float tmp[N];
for(int i=2; i<N; i++) {
  tmp[i] = 2*B[i-2];
  B[i] = tmp[i]+B[i-1];
}

forall(int i=2; i<N; i++) {
  A[i] = tmp[i];
}
```

No matter where `tmp` is declared (inside or outside the loop) it needs to be expanded into an array in order to do loop distribution.

If `tmp` is declared outside the loop then requires **privatization**,

# Loop Distribution May Require Array Expansion

```
float tmp;
for(i=2; i<N; i++) {
  tmp = 2*B[i-2];
  A[i] = tmp;
  B[i] = tmp+B[i-1]
}
```

```
float tmp[N];
for(int i=2; i<N; i++) {
  tmp[i] = 2*B[i-2];
  B[i] = tmp[i]+B[i-1];
}

forall(int i=2; i<N; i++) {
  A[i] = tmp[i];
}
```

No matter where `tmp` is declared (inside or outside the loop) it needs to be expanded into an array in order to do loop distribution.

If `tmp` is declared outside the loop then requires **privatization**, because it actually causes frequent WAW dependencies. However its value is written before being used within the same iteration. Hence it is semantically equivalent to a locally declared variable, which will remove the output (WAW) dependency.

Distribution requires array expansion of the scalar `tmp`.

# False Dependencies (WAR/WAW)

- Cross-Iteration Anti Dependencies (WAR) correspond to a read from the array as it was before the loop ⇒ can be eliminated by reading from a copy of the array.
- Cross-Iteration WAW Dependencies (WAW):
  If they correspond to the case in which every **read** from an scalar or array location is covered by a **previous same-iteration write** ⇒ can be eliminated **privatization (renaming)**, which semantically moves the declaration of the variable (scalar or array) inside the loop.
- Direction-vectors reasoning is limited to relatively simple loop nests, e.g., difficult to reason about privatization in such a way.

# False Dependencies (WAR/WAW)

## Anti Dependency (WAR) and Output Dependency (WAW)

```
// OpenMP code:
// compile with g++ -fopenmp ...
  float tmp = A[1];
  for (i=0; i<N-1; i++)
S1  A[i] = A[i+1];
  A[N-1] = tmp;
//S1→S1: i1+1=i2, [<] WAR
```

# False Dependencies (WAR/WAW)

## Anti Dependency (WAR) and Output Dependency (WAW)

```
// OpenMP code:
// compile with g++ -fopenmp ...
  float tmp = A[1];
  for (i=0; i<N-1; i++)
S1  A[i] = A[i+1];
  A[N-1] = tmp;
//S1→S1: i1+1=i2, [<] WAR

// Solution: copy A into A'
// and use A' for the reads!
float Acopy[N];
#pragma omp parallel for
  for(i=0; i<N; i++) {
    Acopy[i] = A[i];
  }
  tmp = A[1];
#pragma omp parallel for private(i)
  for (i=0; i<N-1; i++) {
    A[i] = Acopy[i+1];
  }
  A[N-1] = tmp;
```

```
int A[M];
for(i=0; i<N; i++){
  for(int j=0, j<M; j++)
      A[j] = (4*i+4*j) % M;
  for(int k=0; k<N; k++)
      X[i,k]=X[i,k-1] * A[A[(2*i+k)%M]];
}

// The write to A[j] causes multiple WAWs,
```

# False Dependencies (WAR/WAW)

## Anti Dependency (WAR) and Output Dependency (WAW)

```
// OpenMP code:
// compile with g++ -fopenmp ...
  float tmp = A[1];
  for (i=0; i<N-1; i++)
S1  A[i] = A[i+1];
  A[N-1] = tmp;
//S1→S1: i1+1=i2, [<] WAR

// Solution: copy A into A’
// and use A’ for the reads!
float Acopy[N];
#pragma omp parallel for
  for(i=0; i<N; i++) {
    Acopy[i] = A[i];
  }
  tmp = A[1];
#pragma omp parallel for private(i)
  for (i=0; i<N-1; i++) {
    A[i] = Acopy[i+1];
  }
  A[N-1] = tmp;
```

```
  int A[M];
  for(i=0; i<N; i++){
    for(int j=0, j<M; j++)
        A[j] = (4*i+4*j) % M;
    for(int k=0; k<N; k++)
        X[i,k]=X[i,k-1] * A[A[(2*i+k)%M]];
  }

// The write to A[j] causes multiple WAWs,
// but A is fully written in the inner loop
#pragma omp parallel{
  int A[M];
#pragma omp for
  for(int i=0; i<N; i++){
    for(int j=0, j<M; j++)
        A[j] = (4*i+4*j) % M;
    for(int k=0; k<N; k++)
        X[i,k]=X[i,k-1] * A[A[(2*i+k)%M]];
  }
}
```

# Reduction is Typically Easy To Recognize

If all the statements in which a scalar variable x appears are of the form x $\oplus$ = *exp*, where x does not appear in exp and $\oplus$ is associative then the cross-iteration RAWs on x can be resolved by:

- privatizing x initialized with the neutral element,
- computing the per-processor partial values of x,
- reducing the xs across processors and with the initial value.

```
// compilation requires g++ -fopenmp ...
  float x = 6.0;
#pragma omp parallel for reduction(+:x) private(i,j)
  for(i=1; i<N; i++) {
    for(j=1; j<N; j++) {
      if ( A[i,j] >= 2.0 )    x += 2*A[i,j-1];
      else if( A[i,j] > 0.0 ) x += A[i-1,j+1];
    }
    if (i % (j+1) == 3) x += A[i,i];
  }
```

# Scan and Segmented Scan Are Difficult!

Compilers cannot recognize and parallelize even simple scans:

- they raise a cross-iteration true dependency (RAW),
- they appear in a multitude of forms,
- hence they are difficult to analyze.

```
// What kind of scans are these?
1. A[0] = B[0];
   for(i=1; i<N; i++) {
     A[i] = A[i-1] + B[i];
   }
2. acc = 0;
   for(i=0; i<N; i++){
     acc = acc xor i;
     A[i] = acc;
   }
3. for(j=0; j<M; j++)
     A[0,j] = B[0,j];
   for(i=1; i<N; i++) {
     for(j=0; j<M; j++)
       A[i,j] = A[i-1,j] + B[i,j];
   }
```

# Scan and Segmented Scan Are Difficult!

Compilers cannot recognize and parallelize even simple scans:

- they raise a cross-iteration true dependency (RAW),
- they appear in a multitude of forms,
- hence they are difficult to analyze.

```
// What kind of scans are these?
1. A[0] = B[0];
   for(i=1; i<N; i++) {
     A[i] = A[i-1] + B[i];
   }
2. acc = 0;
   for(i=0; i<N; i++){
     acc = acc xor i;
     A[i] = acc;
   }
3. for(j=0; j<M; j++)
     A[0,j] = B[0,j];
   for(i=1; i<N; i++) {
     for(j=0; j<M; j++)
       A[i,j] = A[i-1,j] + B[i,j];
   }
```

```
1. let A = scanInc (+) 0 B

2. let A = scanInc (xor) 0 [0..N-1]

3. let A = scanInc (\ a b -> zipWith (+) a b)
                   (replicate M 0.0) B ≡
   let A = transpose $
           map (scanInc (+) 0.0) $
           transpose B
```

1. Direction-Vector Analysis
2. Block Tiling: Matrix Multiplication Case Study
3. Coalesced Accesses: Matrix Transposition Case Study
4. Non-Trivial Synchronization: Histogram Case Study
5. Known Recurrences: Tridiagonal Solver Case Study
6. Imperative Context: Summarization of Array Indexes

# Matrix Multiplication: Loop Strip Mining

```
DOALL i = 1, M, 1    // Parallel
  DOALL j = 1, N, 1  // Parallel
    float tmp = 0.0
    DO k = 1, U, 1 // Reduction
      tmp += A[i,k]*B[k,j]
    ENDDO
    C[i,j] = tmp;
  ENDDO
ENDDO
```

←Matrix Multiplication. Matrices:

- input A has M rows and U columns
- input B has U rows and N columns
- result C has M rows and N columns

Loops of indices i and j are parallel (can be proved by direction vectors).

Accesses to A and B invariant to loops i and j ⇒ Block Tiling to optimize locality of reference!

# Matrix Multiplication: Loop Strip Mining

```
DOALL i = 1, M, 1     // Parallel
  DOALL j = 1, N, 1   // Parallel
    float tmp = 0.0
    DO k = 1, U, 1 // Reduction
      tmp += A[i,k]*B[k,j]
    ENDDO
    C[i,j] = tmp;
  ENDDO
ENDDO
```

←Matrix Multiplication. Matrices:

- input A has M rows and U columns
- input B has U rows and N columns
- result C has M rows and N columns

Loops of indices i and j are parallel (can be proved by direction vectors).

Accesses to A and B invariant to loops i and j ⇒ Block Tiling to optimize locality of reference!

First step: Strip Mining, always safe since the transformed loop executes the same instructions in the same order as the original loop:

```
DO i = 1, N, 1  // stride 1
  loop_body(i)
ENDDO
```

```
DO ii = 1, N, T         // stride T
  DO i = ii, MIN(ii+T-1,N), 1
    loop_body(i)
  ENDDO
ENDDO
```

# Matrix Multiplication: Loop Interchange

After strip mining all loops with a tile of size T:

```
DOALL ii = 1, M, T
  DOALL i = ii, MIN(ii+T-1,M), 1     // loop
    DOALL jj = 1, N, T               // interchange
      DOALL j = jj, MIN(jj+T-1,N), 1
        float tmp = 0.0
        DO kk = 1, U, T
          DO k = kk, MIN(kk+T-1,U), 1
            tmp += A[i,k]*B[k,j]
        ENDDO ENDDO
        C[i,j] = tmp;
ENDDO ENDDO ENDDO ENDDO
```

The second step is to apply loop interchange between the loops of indices i and jj. This is safe because loop i is parallel, hence it can always be interchanged inwards!

# Matrix Multiplication: Summarizing Read Subscripts

After loop interchange we have a grid shape, as in CUDA:

```
DOALL ii = 1, M, T                    // grid.y
  DOALL jj = 1, N, T                  // grid.x
    DOALL i = ii, MIN(ii+T-1,M), 1    // block.y
      DOALL j = jj, MIN(jj+T-1,N), 1  // block.x
        float tmp = 0.0
        DO kk = 1, U, T
          DO k = kk, MIN(kk+T-1,U), 1
            tmp += A[i,k]*B[k,j]
          ENDDO
        ENDDO
        C[i,j] = tmp;
ENDDO ENDDO ENDDO ENDDO
```

The third step is to summarize the subscripts of A and B read inside the loop of index k, for fixed ii, jj and kk (x:y denotes [x...y]):

- A subscripts [ii : MIN(ii+T-1,M), kk : MIN(kk+T-1,U)]
- B subscripts [kk : MIN(kk+T-1,U), jj : MIN(jj+T-1,N)]
- Summaries have size at most $T^2$ & independent on i, j, and k ⇒ CUDA-block threads cooperatively copy-in data to shared mem!

# Block Tiled Matrix Multiplication CUDA Kernel

Shared memory padded with zeros to remove the branch from loop `k`!

```
DOALL ii = 1, M, T   // grid.y
  DOALL jj = 1, N, T // grid.x
    DOALL i = ii, MIN(ii+T-1,M), 1
      DOALL j = jj, MIN(jj+T-1,N), 1
        float tmp = 0.0
        DO kk = 1, U, T
          //we would like to copy
          //to shared memory here
          //& use it inside loop k
          DO k = kk, MIN(kk+T-1,U), 1
            tmp += A[i,k]*B[k,j]
          ENDDO
        ENDDO
        C[i,j] = tmp;
ENDDO ENDDO ENDDO ENDDO
```

# Block Tiled Matrix Multiplication CUDA Kernel

Shared memory padded with zeros to remove the branch from loop `k`!

```
DOALL ii = 1, M, T   // grid.y
  DOALL jj = 1, N, T // grid.x
    DOALL i = ii, MIN(ii+T-1,M), 1
      DOALL j = jj, MIN(jj+T-1,N), 1
        float tmp = 0.0
        DO kk = 1, U, T
          //we would like to copy
          //to shared memory here
          //& use it inside loop k
          DO k = kk, MIN(kk+T-1,U), 1
            tmp += A[i,k]*B[k,j]
          ENDDO
        ENDDO
        C[i,j] = tmp;
ENDDO ENDDO ENDDO ENDDO
```

```
__global__ void matMultTiledKer( ... ) {
  __shared__ T Ash[T][T], Bsh[T][T];
  int ii = blockIdx.y * T; //blockDim.x==T
  int jj = blockIdx.x * T; //blockDim.y==T
  int tidy = threadIdx.y, i = tidy+ii;
  int tidx = threadIdx.x, j = tidx+jj;
  float tmp = 0.0;

  for(int kk=0; kk<U; kk+=T) {
    Ash[tidy,tidx] = (i<M && kk+tidx<U) ?
                     A[i,kk+tidx] : 0.0 ;
    Bsh[tidy,tidx] = (j<N && kk_tidy<U) ?
                     B[kk+tidy,j] : 0.0 ;
    __syncthreads();
    for(int k=0; k<T; k++) {
      tmp += Ash[tidy][k] * Bsh[k][tidx]
    } __syncthreads();
  } if (i<M && j<N) C[i,j] = tmp;
}
```

A global memory access amortized by (T-1) shared memory accesses.

# Measuring GFlops Performance

Sequential matrix multiplication $\sim 2 \times M \times N \times U$ floating point operations. What is the GFlops performance of our implementation?

```
// CPU code
int  dimy = ceil( ((float)M) / T );
int  dimx = ceil( ((float)N) / T );
dim3 block(T,T,1), grid(dimx,dimy,1);

unsigned long int elapsed;
struct timeval t_start,t_end,t_diff;
gettimeofday(&t_start, NULL);
  // ignoring generic shared mem problems
  matMultTiledKer<T><<<grid, block>>>
            (d_A, d_B, d_C, U, M, N);

gettimeofday(&t_end, NULL);
timeval_subtract(&t_diff,
                 &t_end,&t_start);
elapsed=(t_diff.tv_sec*1e6 +
         t_diff.tv_usec);
double flops = 2.0 * M * N * U;
double gigaFlops=(flops*1.0e-3f) /
                 elapsed;
```

```
template <int T> // KERNEL
__global__ void matMultTiledKer( ... ) {
  __shared__ float Ash[T][T], Bsh[T][T];
  int ii = blockIdx.y * T; //blockDim.x==T
  int jj = blockIdx.x * T; //blockDim.y==T
  int tidy = threadIdx.y, i = tidy+ii;
  int tidx = threadIdx.x, j = tidx+jj;
  float tmp = 0.0;

  for(int kk=0; kk<U; kk+=T) {
    Ash[tidy,tidx] = (i<M && kk+tidx<U) ?
                     A[i,kk+tidx] : 0.0 ;
    Bsh[tidy,tidx] = (j<N && kk_tidy<U) ?
                     B[kk+tidy,j] : 0.0 ;
    __syncthreads();
    for(int k=0; k<T; k++) {
      tmp += Ash[tidy][k] * Bsh[k][tidx]
    } __syncthreads();
  } if (i<M && j<N) C[i,j] = tmp;
}
```

1. Direction-Vector Analysis
2. Block Tiling: Matrix Multiplication Case Study
3. Coalesced Accesses: Matrix Transposition Case Study
4. Non-Trivial Synchronization: Histogram Case Study
5. Known Recurrences: Tridiagonal Solver Case Study
6. Imperative Context: Summarization of Array Indexes

# Matrix Transposition: Motivation

```
// Non-Coalesced Memory Access
// Transposition to coalesce it ⇒
DOALL i = 0 to N-1  // parallel
  tmpB = A[i,0] * A[i,0]
  B[i,0] = tmpB
  DO j = 1, 63 // sequential
    tmpA   = A[i, j]
    accum  = tmpB*tmpB + tmpA*tmpA
    B[i,j] = accum
    tmpB   = accum
  ENDDO
ENDDO
```

```
A' = transpose(A)
DOALL i = 0 to N-1  // parallel
  tmpB = A'[0,i] * A'[0,i]
  B'[0,i] = tmpB
  DO j = 1, 63 // sequential
    tmpA    = A'[j, i]
    accum   = tmpB*tmpB + tmpA*tmpA
    B'[j, i] = accum
    tmpB     = accum
  ENDDO
ENDDO
B = transpose(B')
```

The transformed program performs about twice the number of accesses to global memory than the original.

But exhibits only coalesced accesses!

What else could we have done to achieve the same effect?

# Matrix Transposition: Motivation

```
// Non-Coalesced Memory Access
// Transposition to coalesce it ⇒
DOALL i = 0 to N-1  // parallel
  tmpB = A[i,0] * A[i,0]
  B[i,0] = tmpB
  DO j = 1, 63 // sequential
    tmpA   = A[i, j]
    accum  = tmpB*tmpB + tmpA*tmpA
    B[i,j] = accum
    tmpB   = accum
  ENDDO
ENDDO
```

```
A' = transpose(A)
DOALL i = 0 to N-1  // parallel
  tmpB = A'[0,i] * A'[0,i]
  B'[0,i] = tmpB
  DO j = 1, 63 // sequential
    tmpA    = A'[j, i]
    accum   = tmpB*tmpB + tmpA*tmpA
    B'[j, i] = accum
    tmpB     = accum
  ENDDO
ENDDO
B = transpose(B')
```

The transformed program performs about twice the number of accesses to global memory than the original.

But exhibits only coalesced accesses!

What else could we have done to achieve the same effect?
Loop Interchange. Does it works in general?

# Transposition: Strip Mining, Interchange & Kernel

```
//Both loops are parallel
//Strip mining & interchange⇒
for(i = 0; i < rowsA; i++) {
  for(j = 0; j < colsA; j++) {
    trA[j*rowsA+i] = A[i*colsA+j];
} }
```

```
for(ii=0; ii<rowsA; ii+=T) {
  for(jj=0; jj<colsA; jj+=T) {
    for(i=ii; i<min(ii+T,rowsA); i++) {
      for(j=jj; j<min(jj+T,colsA); j++) {
        trA[j*rowsA+i] = A[i*colsA+j];
} } } }
```

```
__global__ void matTranspose(
        float* A,  float* trA,
        int rowsA, int colsA  ) {
  __shared__ float tile[T][T];
  int tidx = threadIdx.x;
  int tidy = threadIdx.y;
  int j    = blockIdx.x*T + tidx;
  int i    = blockIdx.y*T + tidy;
  if( j < colsA && i < rowsA )
    tile[tidy][tidx] = A[i*colsA+j];
  __syncthreads();
  i = blockIdx.y*T + threadIdx.x;
  j = blockIdx.x*T + threadIdx.y;
  if( j < colsA && i < rowsA )
    trA[j*rowsA+i] = tile[tidx][tidy];
}
```

- Trick is to write the element of the symmetric thread in the same block.
- What is the problem?

# Transposition: Strip Mining, Interchange & Kernel

```
//Both loops are parallel
//Strip mining & interchange⇒
for(i = 0; i < rowsA; i++) {
  for(j = 0; j < colsA; j++) {
    trA[j*rowsA+i] = A[i*colsA+j];
} }
```

```
for(ii=0; ii<rowsA; ii+=T) {
  for(jj=0; jj<colsA; jj+=T) {
    for(i=ii; i<min(ii+T,rowsA); i++) {
      for(j=jj; j<min(jj+T,colsA); j++) {
        trA[j*rowsA+i] = A[i*colsA+j];
} } } }
```

```
__global__ void matTranspose(
        float* A,  float* trA,
        int rowsA, int colsA  ) {
  __shared__ float tile[T][T];
  int tidx = threadIdx.x;
  int tidy = threadIdx.y;
  int j    = blockIdx.x*T + tidx;
  int i    = blockIdx.y*T + tidy;
  if( j < colsA && i < rowsA )
    tile[tidy][tidx] = A[i*colsA+j];
  __syncthreads();
  i = blockIdx.y*T + threadIdx.x;
  j = blockIdx.x*T + threadIdx.y;
  if( j < colsA && i < rowsA )
    trA[j*rowsA+i] = tile[tidx][tidy];
}
```

- Trick is to write the element of the symmetric thread in the same block.
- What is the problem?
- Number of shared memory banks typically 16 or 32.
- T is also either 16 or 32 ⇒
- 16 consecutive threads will read the same memory bank at the same time.
- Solution: `tile[T][T+1];`

# Lessons Learned So Far

- Tiled transposition is less that 3× faster than the naive version,
- but the motivating example runs more than 8× faster when transposition coalesces accesses to arrays `A` and `B`. Why?

# Lessons Learned So Far

- Tiled transposition is less that 3× faster than the naive version,
- but the motivating example runs more than 8× faster when transposition coalesces accesses to arrays `A` and `B`. Why?
- Better to eliminate rather than hide latency. Impact of hardware multi-threading limited by the amount of available resources!
- Generic Array of Tuples using shared memory:

# Lessons Learned So Far

- Tiled transposition is less that 3× faster than the naive version,
- but the motivating example runs more than 8× faster when transposition coalesces accesses to arrays `A` and `B`. Why?
- Better to eliminate rather than hide latency. Impact of hardware multi-threading limited by the amount of available resources!

- Generic Array of Tuples using shared memory:requires a class with a "volatile" assignment operator [Esben Skaarup].
- I believe it is safe to ignore the warnings; they can be eliminated at the expense of writing awkward code.

- Shared-memory of generic type [Aske Dorge]:

# Lessons Learned So Far

- Tiled transposition is less that 3× faster than the naive version,
- but the motivating example runs more than 8× faster when transposition coalesces accesses to arrays `A` and `B`. Why?
- Better to eliminate rather than hide latency. Impact of hardware multi-threading limited by the amount of available resources!

- Generic Array of Tuples using shared memory:requires a class with a "volatile" assignment operator [Esben Skaarup].
- I believe it is safe to ignore the warnings; they can be eliminated at the expense of writing awkward code.

- Shared-memory of generic type [Aske Dorge]:empirically, $\sim 8$ words of shared memory per thread does not degrade perform.
- Simple solution: declare in kernel a shared memory array of type `char`, then cast it to the generic type. Set the size when calling the kernel to `block_size*32` bytes.

# Constant (Read-Only) Memory in CUDA

- 64KB of `__constant__` memory on device (global/slow), cached in each multiprocessor, e.g., 8KB (fast).
- May reduce the required memory bandwidth:
  - if found in cache, then no extra traffic,
  - if a (half) warp accesses the same location and misses in cache ⇒ only one request is sent and the result is broadcast back to all,
  - serialized accesses if a warp of threads read different locations!
  - latency can range from one to hundreds of cycles.
- Best Use: when an entire block accesses the same location in the same SIMD instruction: even on a miss, the first warp brings the data in cache @ minimal traffic, the rest find it in cache.

```
// C in __constant__ memory: Good!
DO i = 1, N, 1   // grid
  DO j = 1, M, 1 // block(s)
    A[i,j] = A[i,j] % C[i]
ENDDO
```

```
// C in __constant__ memory: Bad!
DO i = 1, N, 1   // grid
  DO j = 1, M, 1 // block(s)
    A[i,j] = A[i,j] % C[j]
ENDDO
// Either global memory or loop interchange
```

1. Direction-Vector Analysis
2. Block Tiling: Matrix Multiplication Case Study
3. Coalesced Accesses: Matrix Transposition Case Study
4. Non-Trivial Synchronization: Histogram Case Study
5. Known Recurrences: Tridiagonal Solver Case Study
6. Imperative Context: Summarization of Array Indexes

# Histogram: Problem Statement

Sequential Pseudocode & Notation/Parameters:

```
for(int i = 0; i < N; i++ ) {          H = histogram
    (ind, val) = fun(image,i);         B = sizeof(H), e.g., 4×4 ... 64×64
    H[ ind ] += val;                   N = sizeof(image), e.g., 256×256×256
}                                      P = degree of parallelism (# of cores)
```

Typically, the additions to a histogram are 1, but in this case two locations of the histogram are incremented with `val` and `1.0-val` (not shown).

`ind` values are unstructured and may generate data races if loop iterations are executed out of order, i.e., in parallel.

However the code looks like a reduction on arrays ⇒ can be parallelized.

# Histogram: OpenMP Code

## One Histogram/Thread & Reduce Across Histograms

```
int   P  = omp_get_num_threads();
REAL* pH = new REAL[P*B];
#pragma omp parallel
{ int   th_id = omp_get_thread_num();
  REAL* privH = pH + th_id * B;
  for(int i=0; i<B; i++) privH[i]=0.0;

#pragma omp for schedule(static)
  for( int i=0; i < N; i++ ) {
    (ind, val) = fun(image,i);
    privH[ ind ] += val;
  } // end par for
```

```
// Assuming B big enough
pH' = parallel_transpose(pH);

#pragma omp for schedule(static)
  for( int i=0; i<B; i++ ) {
     for( int p=0; p<P; p++ ) {
       H[i] += pH'[i*P + p];
     }
  } // end par for
} // end OMP parallel region
```

Work Efficient: $O(\texttt{P} \times \texttt{B}) \leq O(\texttt{N})$. Typically only the outer loop of the (image) nest is executed in parallel, since P is relatively small.

The alternative is to not privatize & reduce, but instead to provide lock-free (fine-grain) synchronization for each index in the histogram.

# What Is Different for GPGPU Parallelization ?

Differences:

- Quite a few more cores, hence need to exploit all levels of parallelism,
- No (illusion of) constant-time random-memory access, e.g., global memory accesses are up to two order of magnitude more expensive than (scarce) local memory,
- Synchronization primitives limited to local workgroup, i.e., barriers, and can simulate CAS-like instructions at warp level.

# Intuition

- If possible, keep histograms in local memory, e.g., about 16 “local” floats per thread:

- If $\mathtt{B} \leq 16$ then 1 histogram is maintained per thread,
- if $\mathtt{B} = 32$ then two threads build one histogram, using compare-and-swap CAS like locking.

- In general if $16 < \mathtt{B} \leq 32 * 8 = 512$ then up to 32 (warp size) threads cooperate at building a histogram.

- Difficulty: up to $\mathtt{B} = 16 \times 128$ one workgroup builds a histogram; unfortunately no efficient CAS-like sync. at workgroup level.

- Difficulty: for $\mathtt{B} \geq 64 * 64$ need to keep the histogram in global memory: prohibitively expensive due to irregular accesses.

For simplicity use fix $L = 128$ the size of the local workgroup, $G = 512$ the number of workgroups, and unroll the loop: $N/(L \times G)$. Work Efficient $\sim \mathtt{B} \leq \mathtt{N} \ / \ \mathtt{G}$.

# When L / WARP Histograms Fit in Shared Memory

## CUDA Pseudocode; , C is the number of cooperating threads

```
__shared__ REAL sh_hist[ 16 * L ]; // L is the size of the block.
int tid = threadIdx.x;
// init local histogram(s)
for( t = tid; t < 16*L; t += L )
    sh_hist[ t ] = 0.0;
__syncthreads();

// 2) each thread executes U iterations sequentially.
u = blockIdx.x * U * L + tid;
for( t = u; t < u+(U*L); t += L ) {
    (ind, val) = fun(image,i);
    // we work with the transposed sh_hist
    ind = ( ind * L / C ) + ( tid / C );
    if ( C > 1 )
         raw_cas_update( sh_hist, ind, val );
    else sh_hist[ ind ] += val;
}  __syncthreads();

// 3) reduce locally across histograms
segm_scan_reg_block( sh_hist, B * L / C, L / C );
// 4) commit the resulted local histogram to global storage  ...
```

# CAS-like Synchronization

### Function raw_cas_update

```
void raw_cas_update(volatile REAL* sh_hist, ulong ind, REAL val) {
    REAL acc, id  = (REAL)threadIdx.x;
    if (val == 0.0) return;
    bool repeat = true;
    while(repeat) {
        acc = sh_hist[ind];
        sh_hist[ind] = id;
        if( sh_hist[ ind ] == id ) {
            sh_hist[ ind ] = acc + val;
            repeat = false;
        }
    }
}
```

Bug in NVIDIA implem of OpenCL: if barriers are placed inside `while` loop the execution does not deadlocks but incorrect result!

How Do Results Look Like?

1. Direction-Vector Analysis
2. Block Tiling: Matrix Multiplication Case Study
3. Coalesced Accesses: Matrix Transposition Case Study
4. Non-Trivial Synchronization: Histogram Case Study
5. Known Recurrences: Tridiagonal Solver Case Study
6. Imperative Context: Summarization of Array Indexes

# TRIDAG Motivation

*Stochastic Volatility Calibration*: given a set of observed prices of contracts, identify a model of such prices, as a function of volatility (unknown), time and strikes (known), and some other (unobserved) parameters.

Volatility is modeled as a system of continuous differential equations and solved via Crank-Nicolson finite-difference method:

$$\frac{\partial f}{\partial t}(x,t) + \mu(x,t)\frac{\partial f}{\partial x}(x,t) + \frac{1}{2}\sigma(x,t)^2\frac{\partial^2 f}{\partial x^2}(x,t) - r(x,t)f(x,t) = 0 \tag{1}$$

with terminal condition $f(x,T) = F(x), x \in \mathcal{S}$, where $F$ is known.

It comes down to solving this equation for many instances of $\mu, \sigma, r$.

# Explicit Finite Difference Approach

- $\Delta x = (x_J - x_1)/J$, $\Delta t = (t_N - t_1)/N$
- Uses Parametrizations:
  - $D_x f_{j,n} = \frac{f_{j+1,n} - f_{j-1,n}}{2\Delta x}$, $D_x^2 f_{j,n} = \frac{f_{j+1,n} - 2f_{j,n} + f_{j-1,n}}{(\Delta x)^2}$
  - $D_t^- f_{j,n} = \frac{f_{j,n} - f_{j,n-1}}{\Delta t}$
- Using this discretization in the differential equation yields: $f_{j,n-1} = \alpha_{j,n} f_{j-1,n} + \beta_{j,n} f_{j,n} + \gamma_{j,n} f_{j+1,n}$
- where $f_{j,N}$ are known $\forall j \in \{1 \ldots J\}$ and we aim to find $f_{j,1}, \forall j$.
- Trivial Parallel Algorithm Depth $O(T)$, Work $O(JT)$
- However, mathematical reasoning requires $N >> J$, i.e., a very fine-grained time discretization $\Rightarrow$ deep depth!

# Implicit Finite Difference Approach

- $\Delta x = (x_J - x_1)/J$, $\Delta t = (t_N - t_1)/N$
- Uses Parametrizations:
  - $D_x f_{j,n} = \frac{f_{j+1,n} - f_{j-1,n}}{2\Delta x}$, $D_x^2 f_{j,n} = \frac{f_{j+1,n} - 2f_{j,n} + f_{j-1,n}}{(\Delta x)^2}$
  - $D_t^+ f_{j,n} = \frac{f_{j,n+1} - f_{j,n}}{\Delta t}$
- Using this discretization in the differential equation yields: $f_{j,n+1} = a_{j,n} f_{j-1,n} + b_{j,n} f_{j,n} + c_{j,n} f_{j+1,n}$
- where $f_{j,N}$ are known $\forall j \in \{1 \ldots J\}$ and we aim to find $f_{j,1}, \forall j$,
- meaning that we know $f_{j,n+1}$ $\forall j$ and want to compute $f_{j,n}$ $\forall j$
- Requires solving a tridiagonal system (TRIDAG) at every time step; non-trivial to parallelize!
- However, mathematical reasoning requires $N \sim J$, i.e., the depth is reduced and degree of parallelism increased.

# TRIDAG Problem Statement

**Needed: Parallel Algorithm for Computing $X$:**

Given $A$ and $D$, find $X$ such that $A * X = D$,
where $A \in \mathbb{M}^{n \times n}$ is tridiagonal, and $X$ and $D$ vectors of size $n$.

$$\begin{array}{lcccc}
A & * & X & = & D \\
|\; b_0 \;\; c_0 \;\; 0 \;\; 0 \;\; \ldots\ldots\ldots\ldots\ldots \;\; 0 \;| & & |\; x_0 \;| & & |\; d_0 \;| \\
|\; a_0 \;\; b_1 \;\; c_1 \;\; 0 \;\; \ldots\ldots\ldots\ldots\ldots \;\; 0 \;| & & |\; x_1 \;| & & |\; d_1 \;| \\
|\; 0 \;\; a_1 \;\; b_2 \;\; c_2 \;\; 0 \;\; \ldots\ldots\ldots\ldots \;\; 0 \;| & & |\; x_2 \;| & & |\; d_2 \;| \\
|\; \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \;| & * & |\; \ldots\ldots \;| & = & |\; \ldots\ldots \;| \\
|\; 0 \;\; 0 \;\; \ldots \;\; 0 \;\; a_{n-3} \;\; b_{n-2} \;\; c_{n-2} \;| & & |\; x_{n-2} \;| & & |\; d_{n-2} \;| \\
|\; 0 \;\; 0 \;\; \ldots \;\; 0 \;\; 0 \;\; a_{n-2} \;\; b_{n-1} \;| & & |\; x_{n-1} \;| & & |\; d_{n-1} \;|
\end{array}$$

At every step in the time series/discretization (sequential), we know matrix `A` and vector `D` and need to compute in parallel vector `X`.

# High-Level Solution (Sequential and Parallel)

**Problem is to Find $X = [x_0, ..., x_{n-1}]^T$ such that:**

$$
\begin{array}{ccccc}
A & * & X & = & D \\
\left|\begin{array}{lllllll}
b_0 & c_0 & 0 & 0 & \multicolumn{2}{l}{\ldots\ldots\ldots\ldots\ldots\ldots} & 0 \\
a_0 & b_1 & c_1 & 0 & \multicolumn{2}{l}{\ldots\ldots\ldots\ldots\ldots\ldots} & 0 \\
0 & a_1 & b_2 & c_2 & 0 & \ldots\ldots\ldots\ldots\ldots & 0 \\
\multicolumn{7}{l}{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots} \\
0 & 0 & \ldots & 0 & a_{n-3} & b_{n-2} & c_{n-2} \\
0 & 0 & \ldots & 0 & 0 & a_{n-2} & b_{n-1}
\end{array}\right|
& * &
\left|\begin{array}{l} x_0 \\ x_1 \\ x_2 \\ \ldots\ldots \\ x_{n-2} \\ x_{n-1} \end{array}\right|
& = &
\left|\begin{array}{l} d_0 \\ d_1 \\ d_2 \\ \ldots\ldots \\ d_{n-2} \\ d_{n-1} \end{array}\right|
\end{array}
$$

- STEP 1: Compute the LU decomposition of $A$, i.e., $A = L * U$, where $L$ and $U$ are lower and upper-diagonal matrices $\in \mathbb{M}^{n \times n}$.
- STEP 2: Solve $L * Y = D$, i.e., compute partial solution $Y$.
- STEP 3: Solve $U * X = Y$, i.e., compute problem solution $X$.
- Each of the three steps needs to be parallel (of depth $O(logn)$)!

# High-Level Solution (Sequential and Parallel)

## Problem is to Find $X = [x_0, ..., x_{n-1}]^T$ such that:

$A \quad * \quad X \quad = \quad D$

$$
\left|\begin{array}{ccccccc}
b_0 & c_0 & 0 & 0 & \ldots\ldots\ldots & \ldots\ldots\ldots & 0 \\
a_0 & b_1 & c_1 & 0 & \ldots\ldots\ldots & \ldots\ldots\ldots & 0 \\
0 & a_1 & b_2 & c_2 & 0 & \ldots\ldots\ldots & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & \ldots & 0 & a_{n-3} & b_{n-2} & c_{n-2} \\
0 & 0 & \ldots & 0 & 0 & a_{n-2} & b_{n-1}
\end{array}\right|
*
\left|\begin{array}{c}
x_0 \\ x_1 \\ x_2 \\ \ldots\ldots \\ x_{n-2} \\ x_{n-1}
\end{array}\right|
=
\left|\begin{array}{c}
d_0 \\ d_1 \\ d_2 \\ \ldots\ldots \\ d_{n-2} \\ d_{n-1}
\end{array}\right|
$$

```
void tridag( REAL* a, REAL* b, REAL* c, REAL* d, int n, REAL* y, REAL* u ){
    double beta;
    y[0] = d[0];   u[0] = b[0];
    for(int i=1; i<n; i++) {     // distribute with
        beta = a[i] / u[i-1];     // direction vcts!
        u[i] = b[i] - beta*c[i-1];
        y[i] = d[i] - beta*y[i-1];
    }
    y[n-1] = y[n-1]/u[n-1];
    for(int i=n-2; i>=0; i--) {
        y[i] = (y[i] - c[i]*y[i+1]) / u[i];
}  }
```

# LU-Decomposition Recurrences

## Writing the LU decomposition: $A = L * U$, where $L, U \in \mathbb{M}^{n \times n}$

$$\begin{array}{ccccc}
A & = & L & * & U \\
\left| \begin{array}{cccccc} b_0 & c_0 & 0 & 0 & \ldots & 0 \\ a_0 & b_1 & c_1 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & c_{n-2} \\ 0 & 0 & \ldots & 0 & a_{n-2} & b_{n-1} \end{array} \right|
& = &
\left| \begin{array}{ccccc} 1 & 0 & 0 & \ldots & 0 \\ m_0 & 1 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & 0 & m_{n-2} & 1 \end{array} \right|
& * &
\left| \begin{array}{ccccc} u_0 & c_0 & 0 & \ldots & 0 \\ 0 & u_1 & c_1 & \ldots & 0 \\ \ldots & \ldots & \ldots & u_{n-2} & c_{n-2} \\ 0 & \ldots & \ldots & 0 & u_{n-1} \end{array} \right|
\end{array}$$

## LU-decomposition Recurrences:

I. First, compute $u_i$, $i \in \{0 .. n-1\}$:

I.a) $u_0 = b_0$

I.b) $u_i = b_i - ( a_{i-1} * c_{i-1} ) / u_{i-1}$, $i \in \{1 .. n-1\}$

II. Knowing the u's, we can easily compute the m's:

$m_i = a_i / u_i$, $i \in \{0 .. n-2\}$

# The Difficult Recurrence of LU-Decomposition I.b)

## Parallel Algorithm: **scan** with $\mathbb{M}^{2\times 2}$ multiplication operator

Recurrence:

I.a) $u_0 = b_0$

I.b) $u_i = b_i - ( a_{i-1} * c_{i-1} ) / u_{i-1}, \quad i \in \{1 .. n-1\}$

1) Substitute in I.b) $u_i \leftarrow q_{i+1} / q_i$,

where $q_1 = b_0$ and $q_0 = 1.0$, i.e., $u_0 == b_0$ still holds! We obtain:

I.c)
$$\begin{vmatrix} q_{i+1} \\ q_i \end{vmatrix} = \begin{vmatrix} b_i & - a_{i-1} * c_{i-1} \\ 1.0 & 0.0 \end{vmatrix} * \begin{vmatrix} q_i \\ q_{i-1} \end{vmatrix} \quad i \in \{1 .. n-1\}$$

2) Denoting the above $2 \times 2$ matrix by $S_i \in \mathbb{M}^{2\times 2}$ we obtain by induction:

I.d) $| q_{i+1} \;\; q_i |^T = ( S_i * S_{i-1} * S_1 ) * | b_0 \;\; 1.0 |^T$

3) We can compute all such matrices, i.e., $S_i * S_{i-1} * S_1$ with $i \in \{1..n-1\}$,
by applying **scanInc** with the (associative) matrix-multiplication operator (*)

I.e) matrices = scanInc (*) [$S_{n-1}$, $S_{n-2}$, ..., $S_1$]

4) Finally, we compute each $[q_{i+1}, q_i]^T$ by multiplying its corresponding
matrix with $[b_0, 1.0]^T$, and compute $u_i = q_{i+1} / q_i$. Both are PARALLEL:

I.f) q_pairs = map (matVectMult ($b_0$, 1.0)) matrices

I.g) u = map (\ (x,y) → x/y) q_pairs

# STEP 2 & 3: Solving For/Backward Recurrences

## STEP 2: Parallel Algorithm for $L * Y = D$ (forward recurrence)

1) By doing the arithmetics, it comes down to solving the recurrence:

   II.a) $y_0 = d_0$

   II.b) $y_i = d_i - m_{i-1} * y_{i-1}$, $i \in 1 .. n-1$

2) We shall use **scanInc** with linear-function-composition operator:

   i) Consider $f_1(x) = a_1 + b_1*x$ and $f_2(x) = a_2 + b_2*x$

   ii) $(f_1 \diamond f_2)(x) = (a_2+b_2*a_1) + (b_1*b_2)*x$

   iii) We represent such a function f as a pair (a, b), and implement function application via operator: apply x (a, b) = a + b*x

3) Representing $f_i = (d_i, -m_{i-1})$, $i \in 1 .. n-1$

   i) It follows by induction that $y_i = (f_i \diamond f_{i-1} \diamond \ldots \diamond f_1)(d_0)$

   ii) We do a **scanInc** with $\diamond$ and neutral element (0.0, 1.0):

   f_out = scanInc ($\diamond$) (0.0, 1.0) [$f_1$, ..., $f_{n-1}$]

   iii) Finally, apply the f_out's to $d_0$ to compute all $y_i$, $i \in \{1 .. n-1\}$

   y = map (apply $d_0$) f_out

STEP 3: Solving $U * X = Y$ is similar to STEP 2 (but *backwards*):

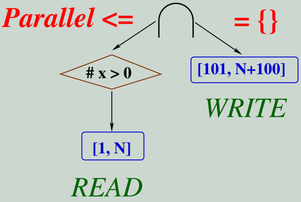
III.a) $x_{n-1} = y_{n-1}/u_{n-1}$  b) $x_i = y_i/u_i - c_i*x_{i+1}/u_i$, $i \in \{n-2..0\}$

1. Direction-Vector Analysis
2. Block Tiling: Matrix Multiplication Case Study
3. Coalesced Accesses: Matrix Transposition Case Study
4. Non-Trivial Synchronization: Histogram Case Study
5. Known Recurrences: Tridiagonal Solver Case Study
6. Imperative Context: Summarization of Array Indexes

# Interprocedural Summarization of Array Indexes

## Independence-Summary Simple Example

```
DO i = 1, N
  A(i+100) = ...
  IF (x > 0) THEN
    ... = A(i)
  ENDIF
ENDDO
```



- Techniques that analyze read-write pairs of accesses become very conservative on larger loops with non-trivial control flow.
- *Alternative:* inter-procedural summarization + model loop independence via an equation on summaries of shape $S = \emptyset$
- Decrease overhead by extracting lightweight predicates that prove independence at runtime, e.g., $\texttt{x} \leq 0 \quad \vee \quad \texttt{N} < 100$.

Show Calculix from SPEC2006!

# Building RO, RW, WF Summaries Interprocedurally

Summaries (RO, RW, WF) are

- constructed via a bottom-up parse of the CALL and CD graphs,
- structural data-flow equations dictate how to compose consecutive regions, aggregate/translate across loops/callsites, ...

# Building RO, RW, WF Summaries Interprocedurally

Summaries (RO, RW, WF) are

- constructed via a bottom-up parse of the CALL and CD graphs,
- structural data-flow equations dictate how to compose consecutive regions, aggregate/translate across loops/callsites, ...

### Simplified solvh_do20 from dyfesm

```
DO i = 1, N
    CALL geteu (XE(IA(i)), NP, SYM)
    CALL matmul(XE(IA(i)), NS)
ENDDO

SUBROUTINE matmul(XE, NS)
  INTEGER NS, XE(*)
  DO j = 1, NS
    ...    = XE(j) ...
    XE(j) = ...
  ENDDO
END
```

```
SUBROUTINE geteu(XE, NP, SYM)
  INTEGER NP, SYM, XE(16, *)

  IF (SYM .NE. 1) THEN
    DO i = 1, NP
      DO j = 1, 16
        XE(j, i) = ...
      ENDDO
    ENDDO
  ENDIF
END
```

# Summarizing Subroutine geteu

## WF summary for geteu; $\mathrm{RO}_{geteu} = \mathrm{RW}_{geteu} = \emptyset$

```
S_geteu  SUBROUTINE geteu(XE, NP, SYM)
           INTEGER NP, SYM, XE(16, *)
S_IF        IF (SYM .NE. 1) THEN
S_Li          DO i = 1, NP
S_Lj            DO j = 1, 16
S_WF              XE(j, i) = ...
                ENDDO
              ENDDO
           ENDIF
         END
```

$WF^{XE}_{S_{WF}} = \{16 * i + j - 1\}$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop $i$: $\{16 * i + j - 1 \;\mid j \in \{1..\,16\}\} \rightarrow 16 * i + [0, 15]$
- Loop $j$: $\{16 * i + [0, 15] \mid i \in \{1..NP\}\} \rightarrow [0, 16 * NP - 1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

# Summarizing Subroutine geteu

## WF summary for geteu; $\text{RO}_{geteu} = \text{RW}_{geteu} = \emptyset$

```
S_geteu   SUBROUTINE geteu(XE, NP, SYM)
            INTEGER NP, SYM, XE(16, *)
S_IF         IF (SYM .NE. 1) THEN
S_Li           DO i = 1, NP
S_Lj             DO j = 1, 16
S_WF                XE(j, i) = ...
                 ENDDO
               ENDDO
            ENDIF
          END
```

$$WF^{XE}_{S_{Lj}} = 16 * i + [0, 15]$$

$$WF^{XE}_{S_{WF}} = \{16 * i + j - 1\}$$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop $i$: $\{16 * i + j - 1 \ \mid j \in \{1..\ 16\}\} \rightarrow 16 * i + [0, 15]$
- Loop $j$: $\{16 * i + [0, 15] \mid i \in \{1..NP\}\} \rightarrow [0, 16 * NP - 1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

# Summarizing Subroutine `geteu`

## WF summary for geteu; $\mathrm{RO}_{geteu} = \mathrm{RW}_{geteu} = \emptyset$

```
$S_{geteu}$  SUBROUTINE geteu(XE, NP, SYM)
               INTEGER NP, SYM, XE(16, *)
$S_{IF}$         IF (SYM .NE. 1) THEN
$S_{Li}$           DO i = 1, NP
$S_{Lj}$             DO j = 1, 16
$S_{WF}$                 XE(j, i) = ...
                     ENDDO
                   ENDDO
                 ENDIF
              END
```

$$WF^{XE}_{S_{Li}} = [0, 16*NP-1]$$

$$WF^{XE}_{S_{Lj}} = 16*i \; + \; [0, 15]$$

$$WF^{XE}_{S_{WF}} = \{16*i+j-1\}$$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop $i$: $\{16*i+j-1 \;\; | \; j \in \{1..\,16\}\} \rightarrow 16*i+[0,15]$
- Loop $j$: $\{16*i+[0,15] \;|\; i \in \{1..NP\}\} \rightarrow [0,16*NP-1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

# Summarizing Subroutine `geteu`

### WF summary for geteu; $RO_{geteu} = RW_{geteu} = \emptyset$

```
S_geteu  SUBROUTINE geteu(XE, NP, SYM)
           INTEGER NP, SYM, XE(16, *)
S_IF        IF (SYM .NE. 1) THEN
S_Li          DO i = 1, NP
S_Lj            DO j = 1, 16
S_WF               XE(j, i) = ...
                ENDDO
              ENDDO
           ENDIF
         END
```

$$WF^{XE}_{S_{IF}} = \begin{array}{c} (SYM \neq 1) \\ \downarrow \\ [0, 16*NP-1] \end{array}$$

$$WF^{XE}_{S_{Li}} = [0, 16*NP-1]$$

$$WF^{XE}_{S_{Lj}} = 16*i + [0,15]$$

$$WF^{XE}_{S_{WF}} = \{16*i+j-1\}$$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop $i$: $\{16*i+j-1 \mid j \in \{1..16\}\} \rightarrow 16*i+[0,15]$
- Loop $j$: $\{16*i+[0,15] \mid i \in \{1..NP\}\} \rightarrow [0,16*NP-1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

# Summarizing Subroutine `matmult`

## RW summary for `matmul`; $\mathrm{RO}_{matmul} = \mathrm{WF}_{matmul} = \emptyset$

| | |
|---|---|
| $S_{matmul}$ | `SUBROUTINE matmul(XE, NS)` |
| | `INTEGER NS, XE(*)` |
| $S_{loop}$ | `DO j = 1, NS` |
| $S_{RO}$ | `...    = XE(j) ...` |
| $S_{WF}$ | `XE(j) = ...` |
| | `ENDDO` |
| | `END` |

$$RO^{XE}_{S_{RO}} = \{j-1\}$$
$$WF^{XE}_{S_{WF}} = \{j-1\}$$

- Composing read-only $RO_{S_1}$ and write-first $WF_{S_2}$ regions:
- $RO = RO_{S_1} - WF_{S_2}$, $WF = WF_{S_2} - RO_{S_1}$, $RW = RO_{S_1} \cap WF_{S_2}$
- In our case $RO = \emptyset$, $WF = \emptyset$, $RW = \{j-1\}$
- Over loop `DO j`: $RO_{loop} = \emptyset$, $WF_{loop} = \emptyset$, $RW_{loop} = [0, NS-1]$

# Summarizing Subroutine `matmult`

## RW summary for `matmul`; $\mathrm{RO}_{matmul} = \mathrm{WF}_{matmul} = \emptyset$

| | |
|---|---|
| $S_{matmul}$ | `SUBROUTINE matmul(XE, NS)` |
| | `  INTEGER NS, XE(*)` |
| $S_{loop}$ | `   DO j = 1, NS` |
| $S_{RO}$ | `     ...    = XE(j) ...` |
| $S_{WF}$ | `     XE(j) = ...` |
| | `  ENDDO` |
| | `END` |

$$S_{RO} \;\diamond\; S_{WF} \;=\; \{\emptyset,\; \emptyset,\; RW = \{j-1\}\}$$

$$RO^{XE}_{S_{RO}} = \{j-1\}$$

$$WF^{XE}_{S_{WF}} = \{j-1\}$$

- Composing read-only $RO_{S_1}$ and write-first $WF_{S_2}$ regions:
- $RO = RO_{S_1} - WF_{S_2}$, $WF = WF_{S_2} - RO_{S_1}$, $RW = RO_{S_1} \cap WF_{S_2}$
- In our case $RO = \emptyset$, $WF = \emptyset$, $RW = \{j-1\}$
- Over loop `DO j`: $RO_{loop} = \emptyset$, $WF_{loop} = \emptyset$, $RW_{loop} = [0, NS-1]$

# Summarizing Subroutine `matmult`

RW summary for `matmul`; $\mathrm{RO}_{matmul} = \mathrm{WF}_{matmul} = \emptyset$

```
S_matmul   SUBROUTINE matmul(XE, NS)
             INTEGER NS, XE(*)
S_loop        DO j = 1, NS
S_RO             ...    = XE(j) ...
S_WF             XE(j) = ...
             ENDDO
           END
```

$RW^{XE}_{S_{loop}} = [0, NS-1]$

$S_{RO} \diamond S_{WF} = \{\emptyset,\ \emptyset,\ RW = \{j-1\}\}$

$RO^{XE}_{S_{RO}} = \{j-1\}$

$WF^{XE}_{S_{WF}} = \{j-1\}$

- Composing read-only $RO_{S_1}$ and write-first $WF_{S_2}$ regions:
- $RO = RO_{S_1} - WF_{S_2},\ WF = WF_{S_2} - RO_{S_1},\ RW = RO_{S_1} \cap WF_{S_2}$
- In our case $RO = \emptyset,\ WF = \emptyset,\ RW = \{j-1\}$
- Over loop `DO j`: $RO_{loop} = \emptyset,\ WF_{loop} = \emptyset,\ RW_{loop} = [0, NS-1]$

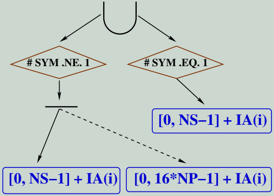
# Summarizing Accesses for the Target Loop

### RW summary for loop `DO i`: $RW^i = ?$

```
          INTEGER NS, NP, IA(*), XE(*)
S_loop    DO i = 1, N
S_WF        CALL geteu (XE(IA(i)),NP,SYM)
S_RW        CALL matmul(XE(IA(i)),NS)
          ENDDO
```

$S_{WF} \diamond S_{RW} = \{\emptyset,\ WF^i = WF_{geteu},\ RW^i\}$

$\mathbf{RW^i =}$



$\cup$

# SYM .NE. 1

# SYM .EQ. 1

[0, NS−1] + IA(i)

[0, NS−1] + IA(i)

[0, 16*NP−1] + IA(i)

In our case, a sufficient condition for `XE` independence is:

$$\cup_{i=1}^{N}(RW_i \ \cap\ (\cup_{k=1}^{i-1} RW_k)) = \emptyset \quad \Leftrightarrow \quad RW_i = \emptyset \quad \Leftrightarrow$$

$$\Leftrightarrow \quad \mathtt{SYM} \neq 1\ \wedge \mathtt{NS} \leq 16 * \mathtt{NP}$$

# Summary-Based Independence Equations

Flow and Anti Independence Equation for loop of index i:

$$\begin{aligned} S_{find} = & \{(\cup_{i=1}^{N} WF_i) \ \cap \ (\cup_{i=1}^{N} RO_i)\} \ \cup \\ & \{(\cup_{i=1}^{N} WF_i) \ \cap \ (\cup_{i=1}^{N} RW_i)\} \ \cup \\ & \{(\cup_{i=1}^{N} RO_i) \ \cap \ (\cup_{i=1}^{N} RW_i)\} \ \cup \\ & \{\cup_{i=1}^{N} (RW_i \ \cap \ (\cup_{k=1}^{i-1} RW_k))\} \quad = \emptyset \end{aligned} \tag{2}$$

Output Independence Equation for loop of index i:

$$S_{oind} = \{\cup_{i=1}^{N} (WF_i \ \cap \ (\cup_{k=1}^{i-1} WF_k))\} \quad = \emptyset \tag{3}$$

Computing $S_{find}$ and $S_{oind}$ solves a more difficult problem than we need, i.e., computes the indexes involved in cross-iteration deps.

*Loop Independence: when are $S_{find}$ and $S_{oind}$ are empty?*

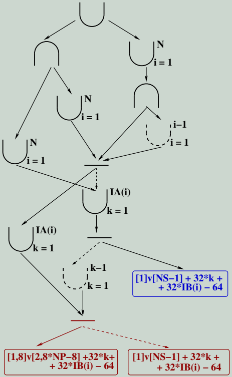
# Key Idea: Predicate-Centric Approach

*Approach centered on extracting arbitrarily-shaped predicates.*

## Key Idea

- *Source of inaccuracy:* summary representation not closed under composition w.r.t. set operations.
- *Language* representation for summaries ... precise but expensive to compute at runtime
- “Let’s *reason* about it!”
  $8 * NP < NS + 6 \Rightarrow A - B = \emptyset \Rightarrow S = \emptyset!$

*Flow Independence USR Example:*



//Show Calculix from SPEC2006!

# A Not So Trivial Loop

## Key Idea

```
k = k0        Ind.  k = k0                      DO i = 0, N-1
DO i=0,N-1    Var.  DO i = 0, N-1                IF(cond(b(i)))THEN
  k = k+2      ⇒     a(k0+2*i)=..                   civ = civ + 1 ⇒ ?
  a(k)=..     Sub.  ENDDO                           a(civ) = ...
ENDDO              k=k0+MAX(2N-2,0)             ENDIF ENDDO
   (a)                 (b)                            (c)
```

- After induction variable substitution, loop (b) is easily found parallel: $\texttt{k0 + 2*i}_1 = \texttt{k0 + 2*i}_2 \Rightarrow \texttt{i}_1 = \texttt{i}_2$

# A Not So Trivial Loop

## Key Idea

```
k = k0        Ind.  k = k0                       DO i = 0, N-1
DO i=0,N-1    Var.  DO i = 0, N-1                 IF(cond(b(i)))THEN
  k = k+2      ⇒     a(k0+2*i)=..                    civ = civ + 1 ⇒ ?
  a(k)=..     Sub.  ENDDO                             a(civ) = ...
ENDDO              k=k0+MAX(2N-2,0)              ENDIF ENDDO
   (a)                  (b)                            (c)
```

- After induction variable substitution, loop (b) is easily found parallel: $\texttt{k0 + 2*i}_1 = \texttt{k0 + 2*i}_2 \Rightarrow \texttt{i}_1 = \texttt{i}_2$
- Loop (c) is similar to (a) but it is more difficult to reason about because `civ` cannot be expressed in terms of the loop index.
- Denote by $\texttt{civ}_0$, $\texttt{civ}^i_\mu$, $\texttt{civ}^i_\gamma$ the values of `civ` at the beginning of the loop, and at the start and end of iteration `i`.
- Express the iteration summary on each path:
  - THEN branch: $\{\texttt{civ}^i_\mu\texttt{+1}\} = [\texttt{civ}^i_\mu\texttt{+1},\texttt{civ}^i_\gamma]$
  - ELSE branch: $\emptyset = [\texttt{civ}^i_\mu\texttt{+1},\texttt{civ}^i_\gamma]$, (an empty interval has its lower bounds $<$ its upper bound.)

# A Not So Trivial Loop

## Key Idea

```
k = k0      Ind.  k = k0                        DO i = 0, N-1
DO i=0,N-1  Var.  DO i = 0, N-1                  IF(cond(b(i)))THEN
  k = k+2    ⇒     a(k0+2*i)=..                     civ = civ + 1 ⇒ ?
  a(k)=..   Sub.  ENDDO                             a(civ) = ...
ENDDO            k=k0+MAX(2N-2,0)               ENDIF ENDDO
   (a)               (b)                               (c)
```

- We have deduced that $W_i = [\texttt{civ}^i_\mu\texttt{+1},\texttt{civ}^i_\gamma]$
- Summarize the accesses of the first $i-1$ iterations:
  - $\texttt{civ}^{i+1}_\mu = \texttt{civ}^{i-1}_\gamma$ since the value of `civ` at the start of an iteration is the same as the one at the end of previous iteration.
  - $\cup_{k=0}^{i-1} W_k = [\texttt{civ}^0_\mu\texttt{+1},\texttt{civ}^0_\gamma] \cup \ldots \cup [\texttt{civ}^{i-1}_\mu\texttt{+1},\texttt{civ}^{i-1}_\gamma] =$
    $= [\texttt{civ}^0_\mu\texttt{+1},\texttt{civ}^{i-1}_\gamma] = [\texttt{civ}^0_\mu\texttt{+1},\texttt{civ}^i_\mu]$
- $(\cup_{k=0}^{i-1} W_k) \cap W_i = [\texttt{civ}^0_\mu\texttt{+1},\texttt{civ}^i_\mu] \cap [\texttt{civ}^i_\mu\texttt{+1},\texttt{civ}^i_\gamma] = \emptyset$
- We have just proved NO output dependencies occur on array a!

# A Not So Trivial Loop

## Solving the True-Dependence (RAW) on civ

```
DO i = 0, N-1
 IF(cond(b(i)))THEN
    civ = civ + 1 ⇒ ?
    a(civ) = ...
ENDIF ENDDO
```

- Extract the loop slice that computes the CIV value. If all accesses to civ are in reduction statement then compute partial contributions of each iteration.
- exclusive scan on the per-iteration contributions give the value of civ at the beginning of each iteration, and
- is plugged in the original loop.

```
civ0 = civ
DOALL i = 0, N-1
  civ = 0
  IF(cond(b(i))) THEN
    civ = civ + 1
  ENDIF
  civs[i] = civ
ENDDO

civs' = scanExc (+) 0 civs

DOALL i = 0, N-1
  civ = civ0 + civs'[i]
  IF(cond(b(i))) THEN
    civ = civ + 1
    a(civ) = ...
  ENDIF
ENDDO
```