Loop Parallelism I

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2023 PMPH Lecture Slides

Course Organization

W	HARDWARE		SOFTWARE	LAB/CUDA
1	Trends Vector Machine	⟵	List HOM (Map-Reduce)	Intro & Simple Map Programming
2	In Order Processor	⟶ ⟵	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.

Problem Statement

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
```

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 = ...
```

Def. 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 = ...
```

Def. 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 = ...
```

Def. 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 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
```

Loop-Nest Dependencies

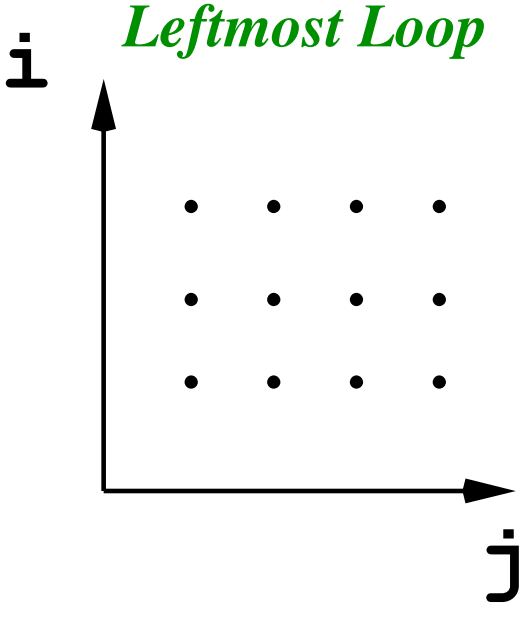
Lexicographic ordering, e.g., $\vec{k}$ =(i=2,j=4) < $\vec{l}$ =(i=3,j=3).

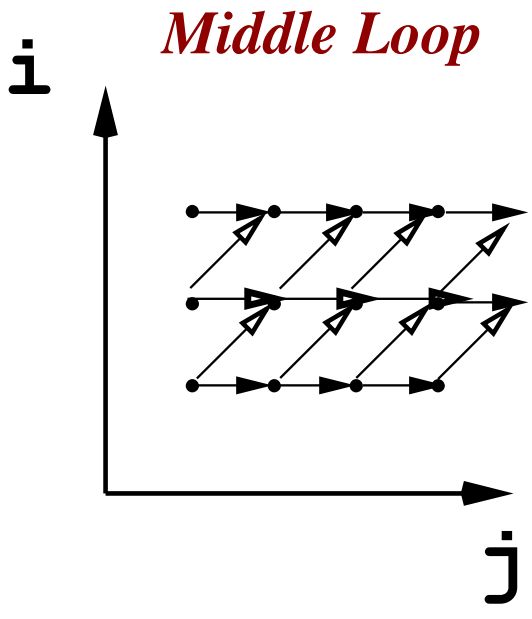
Three Loop Examples

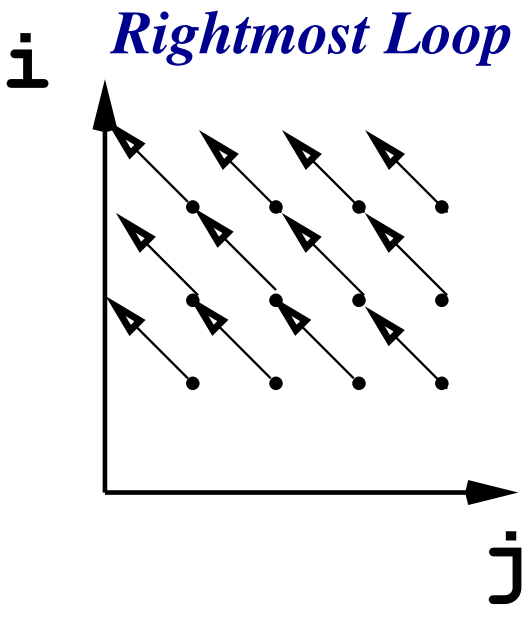
```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

```
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
```







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.

How to Compute the Direction Vectors?

- For any two statements $S1$ and $S2$ that may access the same array A (and one of the accesses is a write),
- in two symbolic iterations $I^1 \equiv (i_1^1, \ldots i_m^1)$ and $I^2 = (i_1^2, \ldots i_m^2)$ (such that $I^1 < I^2$)
- on indices $A[e_1^1, \ldots, e_n^1]$ and $A[e_1^2, \ldots, e_n^2]$, respectivelly,
- then *the direction vectors may be derived* from the equations

$$\begin{cases} e_1^1 = e_1^2 \\ \ldots \\ e_n^1 = e_n^2 \end{cases}$$

(The system of equations models the definition of a dependency: both accesses need to refer to the same memory location!)

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
```

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
```

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:
```

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) = (i2-1,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., $\begin{matrix}[=,<] \\ [<,<]\end{matrix} \rightarrow \begin{matrix}[<,=] \\ [<,<]\end{matrix}$ (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., $\begin{matrix}[=,<] \\ [<,<]\end{matrix} \rightarrow \begin{matrix}[<,=] \\ [<,<]\end{matrix}$ (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. A loop with no dependency cycle is parallel!

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

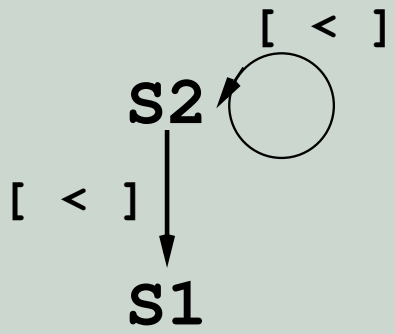
Def. Dependency Graph: edges from the source of the dependency, i.e., early iteration, to the sink, i.e., later iteration.

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. A loop with no dependency cycle is parallel!

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

Loop Distribution May Require Array Expansion

```
float tmp;
for(int 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 perform loop distribution.

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

Loop Distribution May Require Array Expansion

```
float tmp;
for(int 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 perform loop distribution.

If tmp is declared outside the loop then it 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 $\Rightarrow$ 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 a scalar or array location is covered by a **previous same-iteration write** $\Rightarrow$ 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.

Example: Eliminating WAR Dependencies

Anti Dependency (WAR) and Rewritten with Parallel Loops

```
  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
```

Example: Eliminating WAR Dependencies

Anti Dependency (WAR) and Rewritten with Parallel Loops

```
  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(int 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;
```

For OpenMP code, run the g++ compiler with the `-fopenmp` flag. You may play with the degree of parallelism by setting the `OMP_NUM_THREADS` enviornment variable, e.g., to run on 4 cores use: `$ export OMP_NUM_THREADS=4`.

Eliminating WAW Dependencies by Privatization

Assumption: array A is not used after the target loop.

```
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]%M];
}
```

Eliminating WAW Dependencies by Privatization

Assumption: array A is not used after the target loop.

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

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]%M];
}
```

Eliminating WAW Dependencies by Privatization

Assumption: array A is not used after the target loop.

```
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]%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]%M];
  }
}
```

The declaration of `int A[M]` inside the OMP parallel region, will create an array A for every thread (and visible/accessible inside the thread context). Alternatively you may expand A with a new dimension of number-of-processor size, and use explicit indexing.

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 (and commutative) then the cross-iteration RAWs on x can be resolved by:

- performing the updates x $\oplus$ = *exp* atomically, OR 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.

A loop in which all depepndencies are of the form above is called a generalized reduction; all properties of parallel loops are preserved.

```
// 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 ^ 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 ^ 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 = scan (+) 0 B

2. let A = scan (^) 0 (iota N)

3. let A = scan (\ a b -> map2 (+) a b)
                (replicate M 0.0) B
        ≡
   let A = transpose <|
           map (scan (+) 0.0) <|
           transpose B
```

1. Direction-Vector Analysis

2. Optimizing Temporal Locality by Block Tiling
   - Brute-Force Nearest Neighbor Case Study
   - Matrix Multiplication Case Study

3. Coalesced Accesses: Matrix Transposition Case Study

Brute-Force Nearest Neighbor

For simplicity we assume a 1-dimensional space:

- N reference `points`;
- M `queries`, i.e., for each query we need to find its nearest neighbor w.r.t. the reference points.

```
DOALL i = 0, M-1, 1 // In parallel
  float query = queries[i];
  float nn_dist = Infinity;
  float nn_ind  = -1;
  // Sequential over ref. points
  DO j = 0, N-1, 1
    ref_pt = points[j];
    float dist = fabs(query-ref_pt);
    IF dist < nn_dist
      nn_ind  = j;
      nn_dist = dist;
    ENDIF
  ENDDO
  nns[i] = nn_ind;
ENDDO
```

Strategy:

- We execute the queries in parallel;
- We sequentialize the computation of each query; in principle it is a map-reduce, but we do not exploit it.
- Why?

Brute-Force Nearest Neighbor

For simplicity we assume a 1-dimensional space:

- N reference `points`;
- M `queries`, i.e., for each query we need to find its nearest neighbor w.r.t. the reference points.

```
DOALL i = 0, M-1, 1 // In parallel
  float query = queries[i];
  float nn_dist = Infinity;
  float nn_ind  = -1;
  // Sequential over ref. points
  DO j = 0, N-1, 1
    ref_pt = points[j];
    float dist = fabs(query-ref_pt);
    IF dist < nn_dist
      nn_ind  = j;
      nn_dist = dist;
    ENDIF
  ENDDO
  nns[i] = nn_ind;
ENDDO
```

Strategy:

- We execute the queries in parallel;
- We sequentialize the computation of each query; in principle it is a map-reduce, but we do not exploit it.
- Why?

Please note that the queries access the ref points in the same order, so there is potential for exploiting temporal locality.

Loop Strip Mining: A Simple Transformation

Strip mining breaks a loop into two-nested loops, as shown below:

```
// Original Code
DO i = 0, N, 1  // stride 1
  loop_body(i)
ENDDO
```

```
// After Strip Mining
DO ii = 0, N, T          // stride T
  DO i = ii, MIN(ii+T-1,N), 1
    loop_body(i)
  ENDDO
ENDDO
```

When is it safe to strip-mine a loop?

Strip Mining is always safe because the observant student will surely notice that the transformed loop nest executes the same instructions the original loop and in the same order.

Strip Mining: Forming the CUDA Grid and Block

```
DOALL ii = 0, M-1, B // CUDA Grid
  DOALL i = ii, MIN(ii+B-1,M-1), 1 //Block
    // local thread id within Block
    int   loc_tid = i - ii;
    float query = queries[i];
    float nn_dist = Infinity;
    float nn_ind  = -1;
    // Sequential over ref. points
    DO j = 0, N-1, 1
      ref_pt = points[j];
      float dist = fabs(query-ref_pt);
      IF dist < nn_dist
        nn_ind  = j;
        nn_dist = dist;
      ENDIF
    ENDDO
    nns[i] = nn_ind;
  ENDDO // END CUDA BLOCK
ENDDO // END CUDA GRID
```

- Semantically, `blockIdx.x` would be `ii/B`,
- Semantically, `threadIdx.x` would be `i - ii`;
- Harware caching will exploit some temporal locality across threads, but we can do better by using shared memory as a staging buffer!

Strip Mining: Blocking the Sequential Loop

```
DOALL ii = 0, M-1, B // CUDA Grid
  DOALL i = ii, MIN(ii+B-1,M-1), 1 // CUDA Block
    // local thread id within Block
    int   loc_tid = i - ii;
    float query = queries[i];
    float nn_dist = Infinity;
    float nn_ind  = -1;
    // Sequential over ref. points
    DO jj = 0, N-1, B
      // What slice of the points array is
      // used by the loop below? What to do?
      DO j = jj, MIN(jj+B-1, N-1), 1
        ref_pt = points[j];
        float dist = fabs(query-ref_pt);
        IF dist < nn_dist
          nn_ind  = j;
          nn_dist = dist;
        ENDIF
    ENDDO ENDDO
    nns[i] = nn_ind;
  ENDDO
ENDDO
```

Strip Mining: Blocking the Sequential Loop

```
DOALL ii = 0, M-1, B // CUDA Grid
  DOALL i = ii, MIN(ii+B-1,M-1), 1 // CUDA Block
    // local thread id within Block
    int   loc_tid = i - ii;
    float query = queries[i];
    float nn_dist = Infinity;
    float nn_ind  = -1;
    // Sequential over ref. points
    DO jj = 0, N-1, B
      // What slice of the points array is
      // used by the loop below? What to do?
      DO j = jj, MIN(jj+B-1, N-1), 1
        ref_pt = points[j];
        float dist = fabs(query-ref_pt);
        IF dist < nn_dist
          nn_ind  = j;
          nn_dist = dist;
        ENDIF
    ENDDO ENDDO
    nns[i] = nn_ind;
  ENDDO
ENDDO
```

- Strip mine the sequential loop by block size `B`;
- Notice that all `B` threads of the inner block, access in the innermost loop the slice `points[jj: jj+B-1: 1]`, which has at most `B` consecutive elements.
- **Solution:** the `B` threads of the CUDA block collectively copy the `B` elements into a shared-memory `buffer`, and then read from there in the innermost loop (instead of acessing the `points` array which is held in global memory).
- **Essentially we use shared memory as an explicitly programmable cache to reduce the number of global-memory accesses by a factor of `B`.**

Shared Memory as an Explicitly-Programmed Cache

```
DOALL ii = 0, M-1, B // CUDA Grid
  DOALL i = ii, MIN(ii+B-1,M-1), 1 // CUDA Block
    // local thread id within Block
    int   loc_tid = i - ii;
    float query = queries[i];
    float nn_dist = Infinity;
    float nn_ind  = -1;
    // Sequential over ref. points
    DO jj = 0, N-1, B
      float buffer[B];
      collectiveCopy(buffer, points, jj, B);
      // buffer[loc_tid] = points[jj+loc_tid];
      __syncthreads();
      DO j = jj, MIN(jj+B-1, N-1), 1
        ref_pt = buffer[j-jj]; //points[j]
        float dist = fabs(query-ref_pt);
        IF dist < nn_dist
          nn_ind  = j;
          nn_dist = dist;
        ENDIF
    ENDDO __syncthreads(); ENDDO
    nns[i] = nn_ind;
  ENDDO
ENDDO
```

- **Essentially we use shared memory as an explicitly programmable cache to reduce the number of global-memory accesses by a factor of `B`.**
- Each of the `B` threads copies one element from global to shared memory to fill in the `buffer`. Please note that the access to global memory is coalesced.
- Each thread than reads in the innermost loop `B` elements from the shared-memory `buffer`. The number of accesses to global memory is thus reduced with a factor of B!

Approximate CUDA Code

```
// CPU code
int  B = 256;
int dimx = (M + B - 1) / B;
//  dimx = ceil( ((float)M) / B );
dim3 block(B,1,1), grid(dimx,1,1);

unsigned long int elapsed;
struct timeval t_start,t_end,t_diff;
gettimeofday(&t_start, NULL);
  nearestNeighb<B><<<grid, block>>>
       (queries, points, nns, 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 = 3.0 * M * N;
double gigaFlops=(flops*1.0e-3f) /
                 elapsed;
```

```
template <int B> // KERNEL
__global__ void nearestNeighb( ... ) {
  __shared__ float shmem[B];
  int ii = blockIdx.x * B; //blockDim.x==B
  int tidx = threadIdx.x, i = ii+tidx;
  float nn_dist = INFINITY;
  int   nn_ind = -1;
  float query = (i<M)? queries[i] : 0.0;
  for(int jj=0; jj<N; jj+=B) {
    shmem[tidx] = (jj+tidx<N) ?
                  points[jj+tidx] : INFINITY;
    __syncthreads();
    for(int j=0; j<B; j++) {
    //if(jj+j < N && i < M) {
        float ref_pt = shmem[j];
        float dist = fabs(querry - ref_pt);
        if(dist < nn_dist) {
            nn_ind = jj + j;
            nn_dist = dist;
        }
    //}
    } __syncthreads();
  } if ( i < M ) nns[i] = nn_dist;
}
```

Matrix Multiplication: Loop Strip Mining

```
DOALL i = 0, M-1, 1     // Parallel
  DOALL j = 0, N-1, 1   // Parallel
    float tmp = 0.0
    DO k = 0, U-1, 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 = 0, M-1, 1     // Parallel
  DOALL j = 0, N-1, 1   // Parallel
    float tmp = 0.0
    DO k = 0, U-1, 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 = 0, N-1, 1  // stride 1
  loop_body(i)
ENDDO
```

```
DO ii = 0, N-1, T          // stride T
  DO i = ii, MIN(ii+T-1,N-1), 1
    loop_body(i)
  ENDDO
ENDDO
```

Matrix Multiplication: Loop Interchange

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

```
DOALL ii = 0, M-1, T
  DOALL i = ii, MIN(ii+T-1,M-1), 1    // loop
    DOALL jj = 1, N, T                // interchange. Why Safe?
      DOALL j = jj, MIN(jj+T-1,N-1), 1
        float tmp = 0.0
        DO kk = 0, U-1, T
          DO k = kk, MIN(kk+T-1,U-1), 1
            tmp += A[i,k]*B[k,j]
        ENDDO ENDDO
        C[i,j] = tmp;
ENDDO ENDDO ENDDO ENDDO
```

Matrix Multiplication: Loop Interchange

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

```
DOALL ii = 0, M-1, T
  DOALL i = ii, MIN(ii+T-1,M-1), 1    // loop
    DOALL jj = 1, N, T                // interchange. Why Safe?
      DOALL j = jj, MIN(jj+T-1,N-1), 1
        float tmp = 0.0
        DO kk = 0, U-1, T
          DO k = kk, MIN(kk+T-1,U-1), 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 = 0, M-1, T                          // grid.y
  DOALL jj = 0, N-1, T                        // grid.x
    DOALL i = ii, MIN(ii+T-1,M-1), 1  // block.y
      DOALL j = jj, MIN(jj+T-1,N-1), 1// block.x
        float tmp = 0.0
        DO kk = 0, U-1, T
          DO k = kk, MIN(kk+T-1,U-1), 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]`):

Matrix Multiplication: Summarizing Read Subscripts

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

```
DOALL ii = 0, M-1, T                          // grid.y
  DOALL jj = 0, N-1, T                        // grid.x
    DOALL i = ii, MIN(ii+T-1,M-1), 1  // block.y
      DOALL j = jj, MIN(jj+T-1,N-1), 1// block.x
        float tmp = 0.0
        DO kk = 0, U-1, T
          DO k = kk, MIN(kk+T-1,U-1), 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` $\Rightarrow$ 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 = 0, M-1, T   // grid.y
  DOALL jj = 0, N-1, T // grid.x
    DOALL i = ii, MIN(ii+T-1,M-1), 1
      DOALL j = jj, MIN(jj+T-1,N-1), 1
        float tmp = 0.0
        DO kk = 0, U-1, T
          //we would like to copy
          //to shared memory here
          //& use it inside loop k
          DO k = kk, MIN(kk+T-1,U-1), 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 = 0, M-1, T   // grid.y
  DOALL jj = 0, N-1, T // grid.x
    DOALL i = ii, MIN(ii+T-1,M-1), 1
      DOALL j = jj, MIN(jj+T-1,N-1), 1
        float tmp = 0.0
        DO kk = 0, U-1, T
          //we would like to copy
          //to shared memory here
          //& use it inside loop k
          DO k = kk, MIN(kk+T-1,U-1), 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*U + (kk+tidx)] : 0.0;
    Bsh[tidy][tidx] = (j<N && kk_tidy<U) ?
                      B[(kk+tidy)*N + 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*N + 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*U + (kk+tidx)] : 0.0;
    Bsh[tidy][tidx] = (j<N && kk_tidy<U) ?
                      B[(kk+tidy)*N + 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*N + j] = tmp;
}
```

1. Direction-Vector Analysis

2. Optimizing Temporal Locality by Block Tiling
   - Brute-Force Nearest Neighbor Case Study
   - Matrix Multiplication Case Study

3. Coalesced Accesses: Matrix Transposition Case Study

Matrix Transposition: Motivation

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

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

The transformed program performs about three-times more accesses to global memory than the original.

But exhibits only coalesced accesses!

Coalesced Access: a (half) warp accesses in a SIMD instruction consecutive memory (word) locations.

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];
} } } }
```

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+1];
  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+1];
  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 about 2× faster than the naive version,
- but the motivating example runs much faster than that when transposition coalesces accesses to arrays `A` and `B`. Why?

Lessons Learned So Far

- Tiled transposition is about 2× faster than the naive version,
- but the motivating example runs much faster than that 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!

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 $\Rightarrow$ 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
```