COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515
COMP 515 Projects

• Yuhan Peng, Maggie Tang
  —DFGL transformations and OpenCL generation

• Prasanth Chatarasi
  —Polyhedral extensions for data race detection

• Lucas Martinelli, Jonathan Sharman,
  —Exploration of dependences and transformations in higher level OO languages, with a focus on C++ language and libraries (RAJA, Kokkos)

• Jack Feser
  —Exploration of ILP solvers for dependence analysis

• Pete Curry, Lung Li
  —OpenCL transformations for Digital Signal Processors

• Zhipeng Wang
  —Memory Hierarchy Management for iterative graph structures.
procedure vectorize (L, D)
// L is the maximal loop nest containing the statement.
// D is the dependence graph for statements in L.
1. find the set \{S_1, S_2, \ldots, S_m\} of maximal strongly-connected regions in the
   dependence graph D restricted to L (Tarjan);

2. construct \(L_p\) from L by reducing each \(S_i\) to a single node and compute \(D_p\), the
   dependence graph naturally induced on \(L_p\) by \(D\);

3. let \(\{p_1, p_2, \ldots, p_m\}\) be the m nodes of \(L_p\) numbered in an order consistent with \(D_p\) (use
   topological sort);

4. for \(i = 1\) to \(m\) do begin
   if \(p_i\) is a dependence cycle then
     generate a DO-loop nest around the statements in \(p_i\);
   else
     directly rewrite \(p_i\) in Fortran 90, vectorizing it with respect to every loop
     containing it;
   end
end vectorize
### Problems With Simple Vectorization

```plaintext
DO I = 1, N
    DO J = 1, M
        S1
        A(I+1,J) = A(I,J) + B
    ENDDO
ENDDO
```

- **Dependence from** $S_1$ **to itself with** $d(i, j) = (1,0)$
- **Key observation:** Since dependence is at level 1, we can **vectorize** the inner loop!
- **Can be converted to:**
  ```plaintext
  DO I = 1, N
    S1
    A(I+1,1:M) = A(I,1:M) + B
  ENDDO
  ```
- **The simple algorithm does not capitalize on such opportunities**
Advanced Vectorization Algorithm
(Recursive “codegen” procedure)

procedure codegen(R, k, D);
// R is the region for which we must generate code.
// k is the minimum nesting level of possible parallel loops.
// D is the dependence graph among statements in R..
1. find the set \{S_1, S_2, \ldots, S_m\} of maximal strongly-connected regions in the dependence graph D restricted to R;
2. construct \(R_p\) from R by reducing each \(S_i\) to a single node and compute \(D_p\), the dependence graph naturally induced on \(R_p\) by D;
3. let \(\{p_1, p_2, \ldots, p_m\}\) be the m nodes of \(R_p\) numbered in an order consistent with \(D_p\) (topological sort);
4. for i = 1 to m do begin
   if \(p_i\) is cyclic then begin
      generate a level-k DO statement;
      let \(D_i\) be the dependence graph consisting of all dependence edges in D that are at level \(k+1\) or greater and are internal to \(p_i\);
      codegen \(p_i, k+1, D_i\);
      generate the level-k ENDDO statement;
   end
   else
      generate a vector statement for \(p_i\) in \(r(p_i)-k+1\) dimensions, where \(r(p_i)\) is the number of loops containing \(p_i\);
end

codegen(L, 1, D); // Root call for recursive “codegen” procedure
Advanced Vectorization Algorithm

```plaintext
DO I = 1, 100
  X(I) = Y(I) + 10
  DO J = 1, 100
    S_1
    B(J) = A(J, N)
    DO K = 1, 100
      S_2
      A(J+1, K) = B(J) + C(J, K)
      ENDDO
    S_3
    Y(I+J) = A(J+1, N)
  ENDDO
ENDDO

• codegen called at the outermost level
• S_1 will be vectorized, and moved later due to topological sort

DO I = 1, 100
  codegen({S_2, S_3, S_4}, 2, D)
ENDDO
X(1:100) = Y(1:100) + 10
```
Advanced Vectorization Algorithm

- codegen ($\{S_2, S_3, S_4\}$, 2, D)
- level-1 dependences are stripped off

DO $I = 1, 100$
  DO $J = 1, 100$
    codegen($\{S_2, S_3\}$, 3, D)
  ENDDO
$S_4$  Y($I+1:I+100$) = A($2:101,N$)
ENDDO

$X(1:100) = Y(1:100) + 10$
Advanced Vectorization Algorithm

- codegen ({S\textsubscript{2}, S\textsubscript{3}}, 3, D)
- level-2 dependences are stripped off

DO I = 1, 100
  DO J = 1, 100
    B(J) = A(J,N)
    A(J+1,1:100) = B(J) + C(J,1:100)
  ENDDO
  Y(I+1:I+100) = A(2:101,N)
ENDDO
X(1:100) = Y(1:100) + 10
Advanced Vectorization Algorithm
(shown as distributed parallel loops)

DO I = 1, 100
  DO J = 1, 100
    S2: B(J) = A(J,N)
    DOALL K = 1, 100
    S3: A(J+1,K) = B(J) + C(J,K)
    ENDDO
  ENDDO
DOALL J = 1, 100
S4: Y(I+J) = A(J+1,N)
END DO
ENDDO
DOALL I = 1, 100
S1: X(I) = Y(I) + 10
END DO
Enhancing Fine-Grained Parallelism

Chapter 5 of Allen and Kennedy
Fine-Grained Parallelism

Techniques to enhance fine-grained (vector) parallelism:

• Loop Interchange
• Scalar Expansion
• Scalar Renaming
• Array Renaming
Loop Shifting (Permutation)

• Motivation: Identify loops which can be moved and interchange them to “optimal” nesting levels

• Theorem 5.3 In a perfect loop nest, if loops at level \( i, i+1, \ldots, i+n \) carry no dependence, it is always legal to shift these loops inside of loop \( i+n+1 \). Furthermore, these loops will not carry any dependences in their new position.
Loop Shifting

\[
\begin{align*}
\text{DO } & I = 1, N \\
\text{DO } & J = 1, N \\
\text{DO } & K = 1, N \\
S & \quad A(I, J) = A(I, J) + B(I, K) \times C(K, J) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

-  $S$ has true, anti and output dependences on itself, hence codegen will fail as recurrence exists at innermost level.
- Use loop shifting to shift loops $I$ and $J$ inside loop $K$:

\[
\begin{align*}
\text{DO } & K = 1, N \\
\text{DO } & I = 1, N \\
\text{DO } & J = 1, N \\
S & \quad A(I, J) = A(I, J) + B(I, K) \times C(K, J) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
Loop Shifting

\[
\begin{align*}
& \text{DO } K = 1, N \\
& \quad \text{DO } I = 1, N \\
& \quad \quad \text{DO } J = 1, N \\
& \quad \quad S \quad A(I,J) = A(I,J) + B(I,K) \times C(K,J) \\
& \quad \text{ENDDO} \\
& \text{ENDDO} \\
& \text{ENDDO}
\end{align*}
\]

\textit{codegen} vectorizes to:

\[
\begin{align*}
& \text{DO } K = 1, N \\
& \quad \quad A(1:N,1:N) = A(1:N,1:N) + \text{SPREAD}(B(1:N,K),2) \times \text{SPREAD}(C(K,1:N),1)
\end{align*}
\]

ENDDO
Loop Selection

- **Loop Shifting** doesn’t always find the best loop to move. Consider:

```plaintext
DO I = 1, N
    DO J = 1, M
        S
        A(I+1,J+1) = A(I,J) + A(I+1,J)
    ENDDO
ENDDO
```

- **Direction matrix:**

```
<  <  
=  <
```

- Loop shifting algorithm will fail to uncover vector loops; however, interchanging the loops can lead to:

```plaintext
DO J = 1, M
    A(2:N+1,J+1) = A(1:N,J) + A(2:N+1,J)
ENDDO
```

- Need a more general algorithm

```
<  <  
<  =
```
Loop Selection

- Loop selection:
  - Select a loop at nesting level $p \geq k$ that can be safely moved outward to level $k$ and shift the loops at level $k$, $k+1$, ..., $p-1$ inside it.

```
... K P ...
```

```
P K ...
```
Fully Permutable Loop Nest

• A contiguous set of $k \geq 1$ loops, $i_j, \ldots, i_{j+k-1}$ is fully permutable if all permutations of $i_j, \ldots, i_{j+k-1}$ are legal.

• Data dependence test: Loops $i_j, \ldots, i_{j+k-1}$ are fully permutable if for each dependence vector $(d_1, \ldots, d_n)$ carried at levels $j \ldots j+k-1$, each of $d_j, \ldots, d_{j+k-1}$ is non-negative.

• Fundamental result (to be discussed later in course): a set of $k$ fully permutable loops can be transformed using only Interchange, Reversal and Skewing transformations into an equivalent set of $k$ loops where $k-1$ of the loops are parallel.
Scalar Expansion and its use in Removing Anti and Output Dependences

DO $ I = 1, N$
$ S_1 \quad T = A(I) \quad \swarrow$
$ S_2 \quad A(I) = B(I) \quad \nearrow$
$ S_3 \quad B(I) = T \quad \searrow$
ENDDO

- **Scalar Expansion:**

DO $ I = 1, N$
$ S_1 \quad T$(I) = A(I) \quad \swarrow$
$ S_2 \quad A(I) = B(I) \quad \nearrow$
$ S_3 \quad B(I) = T$(I) \quad \searrow$
ENDDO
$ T = T$(N)

- **leads to:**

$ S_1 \quad T$(1:N) = A(1:N)$
$ S_2 \quad A(1:N) = B(1:N)$
$ S_3 \quad B(1:N) = T$(1:N)$ \quad T = T$(N)$
Scalar Expansion

- However, scalar expansion (or any other form of storage duplication) is not useful in removing true dependences. Consider:

  ```
  DO I = 1, N
      T = T + A(I) + A(I+1)
      A(I) = T
  ENDDO
  ```

- **Scalar expansion gives us:**

  ```
  T$(0) = T
  DO I = 1, N
      S$_1$  T$(I) = T$(I-1) + A(I) + A(I+1)
      S$_2$  A(I) = T$(I)
  ENDDO
  T = T$(N)
  ```
Scalar Expansion: Safety

• Scalar expansion is always safe

• When is it useful?
  — Brute force approach: Expand all scalars, vectorize, shrink all unnecessary expansions.
  — However, we want to predict when expansion is useful i.e., when scalar expansion can enable a dependence edge to be deleted

• Dependences due to reuse of memory location vs. reuse of values
  — Dependences due to reuse of values must be preserved (true dependences)
  — Dependences due to reuse of memory location can be deleted by expansion (anti & output dependences)
    - This is also why functional languages are easier to parallelize, at the cost of increased memory overhead
Scalar Renaming

```
DO I = 1, 100
S_1 T = A(I) + B(I)
S_2 C(I) = T + T
S_3 T = D(I) - B(I)
S_4 A(I+1) = T * T
ENDDO
```

- **Renaming scalar T:**

```
DO I = 1, 100
S_1 T_1 = A(I) + B(I)
S_2 C(I) = T_1 + T_1
S_3 T_2 = D(I) - B(I)
S_4 A(I+1) = T_2 * T_2
ENDDO
```
Scalar Renaming

• will lead to:

S_3 \quad T2^$(1:100) = D(1:100) - B(1:100)
S_4 \quad A(2:101) = T2^$(1:100) * T2^$(1:100)
S_1 \quad T1^$(1:100) = A(1:100) + B(1:100)
S_2 \quad C(1:100) = T1^$(1:100) + T1^$(1:100)
T = T2^$(100)
1. Solve exercise 5.6 in book
   — Your solution should be legal for all values of K (note that the value of K is invariant in loop I)

Exercise 5.6: What vector code should be generated for the following loop?

DO I = 1, 100
   A(I) = B(K) + C(I)
   B(I+1) = A(I) + D(I)
END DO

• Due in class on Thursday, Oct 8th