COMP 515: Advanced Compilation for Vector and Parallel Processors

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

COMP 515  Lecture 21  1 December, 2015
Compiling Array Assignments
(a.k.a. “Scalarization”)

Allen and Kennedy, Chapter 13
Fortran 90

• Fortran 90: successor to Fortran 77

• Slow to gain acceptance:
  — Need better/smarter compiler techniques to achieve same level of performance as Fortran 77 compilers

• This chapter focuses on a single new feature - the array assignment statement:  \( A(1:100) = 2.0 \)
  — Intended to provide direct mechanism to specify parallel/vector execution

• This statement must be implemented for the specific available hardware. In an uniprocessor, the statement must be converted to a scalar loop: Scalarization
  — “Scalarization” techniques are also useful for vectorization when array size is larger than vector width (common case)
Fortran 90

- Range of a vector operation in Fortran 90 denoted by a triplet:
  \(<\text{lower bound}: \text{upper bound}: \text{increment}>\)

  \[A(1:100:2) = B(2:51:1) + 3.0\]

- Semantics of Fortran 90 require that for vector statements, all inputs to the statement are fetched before any results are stored

- As with DO loops, the default value of the increment is 1, i.e., \(B(2:51)\) is equivalent to \(B(2:51:1)\)
Outline

• Simple scalarization
• Safe scalarization
• Techniques to improve on safe scalarization
  — Loop reversal
  — Input prefetching
  — Loop splitting
• Multidimensional scalarization
• A framework for analyzing multidimensional scalarization
Scalarization

• Replace each array assignment by a corresponding DO loop
• Is it really that easy?
• Two key issues:
  — Wish to avoid generating large array temporaries
  — Wish to optimize loops to exhibit good memory hierarchy performance
Simple Scalarization

• Consider the vector statement:
  \[ A(1:200) = 2.0 \times A(1:200) \]

• A scalar implementation:

  \[
  S_1 \quad \text{DO } I = 1, 200 \\
  S_2 \quad \text{A}(I) = 2.0 \times \text{A}(I) \\
  \text{ENDDO}
  \]

• However, some statements cause problems:
  \[ A(2:201) = 2.0 \times A(1:200) \]

• If we naively scalarize, we get incorrect code:
  \[
  \text{DO } i = 1, 200 \\
  \text{A}(i+1) = 2.0 \times \text{A}(i) \\
  \text{ENDDO}
  \]
Scalarization Faults

• Why do scalarization faults occur?

• Vector operation semantics: All values from the RHS of the assignment should be fetched before storing into the result

• If a scalar operation stores into a location fetched by a later operation, we get a scalarization fault

• **Principle 13.1**: A vector assignment generates a scalarization fault if and only if the scalarized loop carries a true dependence.

• These dependences are known as **scalarization dependences**

• To preserve correctness, compiler should never produce a scalarization dependence
Safe Scalarization

- Naive algorithm for safe scalarization: Use temporary storage to make sure scalarization dependences are not created

  - Consider:
    
    \[ A(2:201) = 2.0 \times A(1:200) \]

  - can be split up into:
    
    \[ T(1:200) = 2.0 \times A(1:200) \]
    
    \[ A(2:201) = T(1:200) \]

  - Then scalarize using SimpleScalarize
    
    \[
    \begin{align*}
    & \text{DO } I = 1, 200 \\
    & \quad T(I) = 2.0 \times A(I) \\
    & \text{ENDDO} \\
    & \text{DO } I = 2, 201 \\
    & \quad A(I) = T(I-1) \\
    & \text{ENDDO}
    \end{align*}
    \]
Safe Scalarization

• Procedure SafeScalarize implements this method of scalarization

• **Good news:**
  – Scalarization always possible by using temporaries

• **Bad News:**
  – Substantial increase in memory use due to temporaries
  – More memory operations per array element
  – Akin to overheads incurred in implementing functional languages

• **We shall look at a number of techniques to reduce the effects of these disadvantages**
Loop Reversal

\[ \mathbf{A}(2:256) = \mathbf{A}(1:255) + 1.0 \]

- A scalarization approach using loop reversal that avoids the need for a temporary:

\[
\text{DO } \text{I} = 256, 2, -1 \\
\quad \mathbf{A}(\text{I}) = \mathbf{A}(\text{I}-1) + 1.0 \\
\text{ENDDO}
\]
Loop Reversal

• When can we use loop reversal?
  — Loop reversal maps true dependences into antidependences
  — But may also map antidependences into dependences
    \[ A(2:257) = \frac{(A(1:256) + A(3:258))}{2.0} \]

• After scalarization:
  
  \[
  \text{DO I = 2, 257}
  \quad A(I) = \frac{(A(I-1) + A(I+1))}{2.0}
  \quad \text{ENDDO}
  \]

• Loop Reversal gets us:
  
  \[
  \text{DO I = 257, 2}
  \quad A(I) = \frac{(A(I-1) + A(I+1))}{2.0}
  \quad \text{ENDDO}
  \]

• Thus, cannot use loop reversal in presence of antidependences

• Goal: ensure that scalarized loop has no loop-carried true dependences
Input Prefetching

\[ A(2:257) = \left( A(1:256) + A(3:258) \right) / 2.0 \]

- Causes a scalarization fault when naively scalarized to:
  
  ```
  DO I = 2, 257
      A(I) = ( A(I-1) + A(I+1) ) / 2.0
  ENDDO
  ```

- Problem: Stores into first element of the LHS in the previous iteration

- Input prefetching: Use scalar temporaries to store elements of input and output arrays
Input Prefetching

• A first-cut at using temporaries:

\[
\text{DO } I = 2, 257 \\
T1 = A(I-1) \\
T2 = (T1 + A(I+1)) / 2.0 \\
A(I) = T2 \\
\text{ENDDO}
\]

• T1 holds element of input array, T2 holds element of output array

• But this faces the same problem. Can correct by moving assignment to T1 into previous iteration...
Input Prefetching

\[ \text{T1} = A(1) \]
\[ \text{DO I = 2, 256} \]
\[ \quad \text{T2} = \left( \frac{\text{T1} + A(I+1)}{2} \right) / 2.0 \]
\[ \quad \text{T1} = A(I) \]
\[ \quad A(I) = \text{T2} \]
\[ \text{ENDDO} \]
\[ \text{T2} = \left( \frac{\text{T1} + A(257)}{2} \right) / 2.0 \]
\[ A(I) = \text{T2} \]

• Note: We are using scalar replacement, but the motivation for doing so is different than in Chapter 8
Input Prefetching

• Already seen in Chapter 8, we need as many temporaries as the dependence threshold + 1.

• Example:

```fortran
DO I = 2, 257
    A(I+2) = A(I) + 1.0
ENDDO
```

• Can be changed to:

```fortran
T1 = A(1)
T2 = A(2)
DO I = 2, 255
    T3 = T1 + 1.0
    T1 = T2
    T2 = A(I+2)
    A(I+2) = T3
ENDDO
T3 = T1 + 1.0
T1 = T2
A(258) = T3
T3 = T1 + 1.0
A(259) = T3
```
Input Prefetching

• Can also unroll the loop and eliminate register to register copies

• Principle 13.2: Any scalarization dependence with a threshold known at compile time can be corrected by input prefetching.
Input Prefetching

• Sometimes, even when a scalarization dependence does not have a constant threshold, input prefetching can be used effectively

\[ A(1:N) = \frac{A(1:N)}{A(1)} \]

• which can be naively scalarized as:

\[
\begin{align*}
\text{DO } & i = 1, N \\
& A(i) = \frac{A(i)}{A(1)} \\
\text{ENDDO}
\end{align*}
\]

• true dependence from first iteration to every other iteration

• antidependence from first iteration to itself

• Via input prefetching, we get:

\[
\begin{align*}
tA1 & = A(1) \\
\text{DO } & i = 1, N \\
& A(i) = \frac{A(i)}{tA1} \\
\text{ENDDO}
\end{align*}
\]
Multidimensional Scalarization

• Vector statements in Fortran 90 in more than 1 dimension:

\[ A(1:100, 1:100) = B(1:100, 1, 1:100) \]

• corresponds to:

\[
\begin{align*}
&\text{DO } J = 1, 100 \\
&A(1:100, J) = B(1:100, 1, J)
\end{align*}
\]

ENDDO

• Scalarization in multiple dimensions:

\[ A(1:100, 1:100) = 2.0 \times A(1:100, 1:100) \]

• Obvious Strategy: convert each vector iterator into a loop:

\[
\begin{align*}
&\text{DO } J = 1, 100, 1 \\
&\text{DO } I = 1, 100 \\
&A(I,J) = 2.0 \times A(I,J)
\end{align*}
\]

ENDDO
Multidimensional Scalarization

• What should the order of the loops be after scalarization?
  — Familiar question: We dealt with this issue in Loop Selection/Interchange in Chapter 5

• Profitability of a particular configuration depends on target architecture
  — For simplicity, we shall assume shorter strides through memory are better
  — Thus, optimal choice for innermost loop is the leftmost vector iterator
Loop Interchange

• Sometimes, there is a tradeoff between scalarization and optimal memory hierarchy usage

  \[ A(2:100, 3:101) = A(3:101, 1:201:2) \]

• If we scalarize this using the prescribed order:

  \[
  \text{DO } I = 3, 101 \\
  \text{DO 100 } J = 2, 100 \\
  \quad A(J,I) = A(J+1,2*I-5) \\
  \text{ENDDO} \\
  \text{ENDDO}
  \]

• Direction vectors for true dependences:

  \[ (<, >) \text{ (for } I = 3, 4) \text{ and } (> , >) \text{ (for } I = 6, 7) \]

• Cannot use loop reversal, input prefetching

• Can use temporaries
Loop Interchange

• However, we can use loop interchange to get:

    DO J = 2, 100
        DO I = 3, 101
            A(J, I) = A(J+1, 2*I-5)
        ENDDO
    ENDDO

• Not optimal memory hierarchy usage, but reduction of temporary storage

• Loop interchange is useful to reduce size of temporaries
• It can also eliminate scalarization dependences
General Multidimensional Scalarization

- **Goal:** To vectorize a single statement which has m vector dimensions
  - Given an ideal order of scalarization \((l_1, l_2, \ldots, l_m)\)
  - \((d_1, d_2, \ldots, d_n)\) be direction vectors for all plausible and implausible true dependences of the statement upon itself
  - The scalarization matrix is a \(n \cdot m\) matrix of these direction vectors

- For instance:

\[
A(1:N, 1:N, 1:N) = A(2:N+1, 1:N, 0:N-1) + A(0:N-1, 2:N+1, 1:N)
\]

\[
\begin{array}{ccc}
> & = & < \\
< & > & = \\
\end{array}
\]
If we examine any column of the direction matrix, we can immediately see if the corresponding loop can be safely scalarized as the outermost loop of the nest:

- If all entries of the column are $= \text{or} >$, it can be safely scalarized as the outermost loop without loop reversal.
- If all entries are $= \text{or} <$, it can be safely scalarized with loop reversal.
- If it contains a mixture of $<$ and $>$, it cannot be scalarized by simple means.
  - Loop skewing could work
Once a loop has been selected for scalarization, the dependences carried by that loop, any dependence whose direction vector does not contain a $=$ in the position corresponding to the selected loop may be eliminated from further consideration.

In our example, if we move the second column to the outside, we get:

$\begin{pmatrix} > & = & < \\ < & > & = \end{pmatrix} \rightarrow \begin{pmatrix} = & > & < \\ > & < & = \end{pmatrix}$

Scalarization in this way will reduce the matrix to:

$\begin{pmatrix} > & < \end{pmatrix}$
Scalarization Example

\[
\text{DO } J = 2, N-1 \\
\text{ENDDO}
\]

- Loop carried true dependence, antidependence
- Naive compiler could generate:

\[
\text{DO } J = 2, N-1 \\
\text{DO } i = 2, N-1 \\
\quad T(i-1) = (A(i-1,J) + A(i+1,J) + A(i,J-1) + A(i,J+1))/4 \\
\text{ENDDO} \\
\text{DO } i = 2, N-1 \\
\quad A(i,J) = T(i-1) \quad \text{ENDDO} \\
\text{ENDDO}
\]

- \(2 \cdot (N-2)^2\) accesses to memory due to array T
Scalarization Example

• However, can use input prefetching to get:

```fortran
DO J = 2, N-1
   tA0 = A(1, J)
   DO i = 2, N-2
      tA1 = (tA0+A(i+1,J)+A(i,J-1)+A(i,J+1))/4
      tA0 = A(i-1, J)
   A(i,J) = tA1
   ENDDO
   tA1 = (tA0+A(N,J)+A(N-1,J-1)+A(N-1,J+1))/4
   A(N-1,J) = tA1
ENDDO
```

• If temporaries are allocated to registers, no more memory accesses than original Fortran 90 program
Post Scalarization Issues

• Issues due to scalarization:
  – Generates many individual loops
  – These loops carry no dependences. So reuse of quantities in registers is not common

• Solution: Use loop interchange, loop fusion, unroll-and-jam, and scalar replacement