MLIR for Fortran

Vivek Sarkar * (Georgia Tech)
Nelson Amaral (U. Alberta)
Kit Barton (IBM)
Wang Chen (IBM)

* Disclaimer: opinions in this talk are those of the presenter, and do not reflect the official opinions of IBM!
Overview

• New research project planned for 2020 across Georgia Tech, U.Alberta, IBM

• Motivated by past work at IBM on ASTI optimizer for Fortran, and recent MLIR work at IBM with U.Alberta

• Goal is to leverage past experiences with ASTI’s High-level Intermediate Representation (HIR) to evaluate the design space for an MLIR dialect for optimization of Fortran codes
  – Special focus on array statements, loops, array accesses
  – Such an MLIR dialect could be useful for optimization of HPC codes in other languages as well

• Complementary to ECP Flang project

• All feedback and suggestions are most welcome!
Background: ASTI Optimizer
(Analyzer, Scalarizer, Transformer, Interprocedural optimizer)

- Automatic selection of high-order transformations in the IBM XL Fortran compilers
- Quantitative approach to program optimization using cost models
- High-order transformations selected for uniprocessor target include: loop distribution, fusion, interchange, reversal, skewing, tiling, unrolling, and scalar replacement of array references
- Design and initial product implementation completed during 1991–1993

Structure of XL Fortran Product Compiler (Version 4)

- **Fortran 90 front end**
- **ASTI Optimizer**
  - `-qhot`
  - **Intermediate Language**
  - **Transformed intermediate language**
  - **Interprocedural optimizer**
  - `-qipa`
  - **Optimized intermediate language**
  - `-o3`
  - **Optimizing back end**
  - Optimized RS/6000 executable

- **Translation to HIR**
  - **Input HIR**
  - **Analyzer**
  - **Scalarizer**
  - **Transformer**
  - **Transformed HIR**
  - **Code augmentor**
  - **Transformed & augmented HIR**
  - **Translation from HIR**
(a) Example loop containing an array assignment statement:
-----------------------------
do k = 1, n
 A(1:n,1:n) = A(1:n,1:n) + spread(B(1:n,k),2,n) * spread(C(k,1:n),1,n)
end do

(b) After unoptimized scalarization:
------------------------------------
do k = 1, n
 do i1 = 1, n ! parallelizable loop
  T1(i1) = B(i1,k)
 end do
end do

do i2 = 1, n ! parallelizable loop
 do i1 = 1, n ! parallelizable loop
  T2(i1,i2) = T1(i1)
 end do
end do

do i1 = 1, n ! parallelizable loop
 T3(i1) = C(k,i1)
end do

do i2 = 1, n ! parallelizable loop
 do i1 = 1, n ! parallelizable loop
  T4(i1,i2) = T3(i2)
 end do
end do

do i2 = 1, n ! parallelizable loop
 do i1 = 1, n ! parallelizable loop
  T5(i1,i2) = A(i1,i2) + T2(i1,i2) * T4(i1,i2)
 end do
end do
end do

(c) After optimized scalarization:
----------------------------------
do k = 1, n
 do i2 = 1, n ! parallelizable loop
   do i1 = 1, n ! parallelizable loop
       A(i1,i2) = A(i1,i2) + B(i1,k) * C(k,i2)
   end do
 end do
end do

(d) After collective transformation of loop nest (c):
-----------------------------------------------------
do bb$_i2=1,n,b$_i2 ! parallelizable loop
 do bb$_i1=1,n,b$_i1 ! parallelizable loop
   do bb$_k =1,n,b$_k
       do i2=max(1,bb$_i2),min(n,bb$_i2+b$_i2-1)
         do i1=max(1,bb$_i1),min(n,bb$_i1+b$_i1-1)
           do k=max(1,bb$_k),min(n,bb$_k+b$_k-1)
             A(i1,i2) = A(i1,i2) + B(i1,k) * C(k,i2)
           end do
         end do
       end do
     end do
   end do
 end do
end do
end do
end do
end do

---

Fig. 3. Matrix multiply example

---

Fig. 4. Matrix multiply example (contd.)

---

Fig. 5. Performance measurements for matrix multiply example
Structure of ASTI Transformer


ASTI transformations include:

- Loop transformations: distribution, fusion unimodular, tiling, unrolling, coalescing, parallelization, array language scalarization
- Data transformations: scalar replacement, alignment, padding

Traditional Optimizer Structure

Traditional Optimizer Structure

Loop Structure Graph and Dependence Analysis foundations derived from prior work in PTRAN project

Demand-driven incremental analyses

ASTI transformations include:

- Loop transformations: distribution, fusion unimodular, tiling, unrolling, coalescing, parallelization, array language scalarization
- Data transformations: scalar replacement, alignment, padding
Loop Structure Tree

LCFGs, input/output transformer has to enumerate its control and data

V. SARKAR

Loop Structure analyzer initialized

VOL. 41

Loop-Level Control Flow Graphs

LCFG(I.3):

START → S.6 → S.7 → STOP

LCFG(I.2):

START → S.4 → S.5 → S.3 → True

START → S.3 → I.1 → S.9 → STOP

START → S.1 → S.2 → S.10 → STOP

Figure 4.
Structure of a Single Optimization Pass

Optimization = Analysis + Transformation

Examples:

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value numbering</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Liveness analysis</td>
<td>Dead store elimination</td>
</tr>
<tr>
<td>Dependence analysis</td>
<td>Instruction scheduling</td>
</tr>
</tbody>
</table>
Issues

• **Heuristics**: sequence of optimization passes for a given optimization level is usually hardwired

• **Phase ordering**: a later optimization can create new opportunities for a previous optimization

• **Compile time/space overheads**: all optimization passes are performed on all instructions in all procedures

• **Unpredictability**: hard to predict how much performance improvement will be delivered by compiler optimizations

• **Pessimization**: optimizing transformations can sometimes degrade performance
Rethinking compilers …a quantitative approach

\[ \text{Optimization} = \text{Cost function} + \text{Analysis} + \text{Transformation} \]

**Examples:**

<table>
<thead>
<tr>
<th>Cost Function</th>
<th>Analysis</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store freqs</td>
<td>Live variable analysis</td>
<td>Dead store elimination</td>
</tr>
<tr>
<td>Basic block freqs</td>
<td>Bounds analysis</td>
<td>Bounds check elimination</td>
</tr>
<tr>
<td>Cache misses</td>
<td>Dependence vectors</td>
<td>Loop interchange and tiling</td>
</tr>
</tbody>
</table>
Transformations performed in ASTI for Locality Optimizations

1. Initialization
2. Loop distribution
3. Identification of perfect loop nests
4. Reduction recognition
5. Locality optimization
6. Loop fusion
7. Loop–invariant scalar replacement
8. Loop unrolling and interleaving
9. Local scalar replacement
10. Transcription — generate transformed HIR

Other transformations performed for Vector/SMP/HPF parallelization
Challenges

• Cost function should be efficient to compute but sufficiently accurate
  – Use lower and upper bounds as approximations

• Well-tuned code should not incur large compilation overhead
  – Perform analysis and transformation incrementally and on demand, only when cost function indicates potential for performance improvement
  – Use algorithms with low-polynomial-time complexity

• Phase ordering should be driven by cost functions
  – Use classical optimization theory heuristics in driver for optimizing compiler e.g., sort potential transformations in decreasing order of benefit
Summary

• Time to rethink optimizing compilers
  – Reduce compilation overhead
  – Increase optimization effectiveness

• Quantitative approach provides a promising foundation

• *Future goal*: build an optimizer in which all optimization selection and phase ordering decisions are driven by cost functions rather than hardwired heuristics
  – MLIR offers a promising opportunity for such an approach
  – Fortran is an important domain for demonstrating such an approach
BACKUP SLIDES START
HERE
Structure of Optimizing Compilers

Source code

Front end

HIR optimizations

HIR = High-level Intermediate Representation

Middle end

Optimized HIR

Lowering of IR

LIR = Low-level Intermediate Representation

Optimized LIR

LIR optimizations

Instruction Selection

Optimized MCR

MCR = Machine Code Representation

MCR optimizations

Final assembly

Optimized Binary code