Module Summary 131-135

Advanced Compiler Techniques

Source:
https://www.youtube.com/playlist?list=PLf3ZkSCyj1tf3rPAkOKY5hUzDrDoekAc7
Videos: 131 - 135

Summarizing

- Changing axes using Fourier Motzkin method (131)
- Affine array accesses (132)
- Data Reuse (133)
    - Data Reuse Category: Self Reuse (134)
    - Data Reuse Category: Self Spatial Reuse (135)

Module 131: Changing axes using Fourier Motzkin Method

Changing axes/order of loop iteration indices

- Fourier Motzkin: To project a polyhedron on a smaller dimension
  - For n-Dimension poly -> Remove n^{th} Dimension & project Poly on remaining n-1 Dimensions
  - Manipulate inequalities that represent polyhedron.
- Problem: Given polyhedron iteration space S, generate a loop nest with new order $(x_1, x_2 \ldots ,x_n)$ of loop iteration indices.

```
for ( x2 : 0 .. )
  for (x4 : 0 .. )
    ...
    for (xn   : 0..)
     for (xn-1 : 0 ..)
```

```
for ( x1 : 0 .. )
  for (x2 : 0 .. )
    ...
    for (xn : 0 ..)
```

- Change indices order: Respect data dependencies
- For the new order: **What is LB and UB for each of iteration indices** ?
  - Fourier Motzkin

Loop Bounds Generation for loop iterations indices

- Algorithm
  - Start with innermost loop index x_n
  - Bring inequations to form: (for c_1, c_2 >= 0)
    - $L \le c1 * x_n$
    - $c_2 * x_n \le U$
  - Use Fourier Motzkin recursively to project on remaining indices x_1, x_2,.. X_{n-1}
  - Output: Upper and Lower bound for x_1, which is constant.
    - Use above inequations to get lower upper bound of x_2, … till x_n

```
for ( x1 : 0 .. )
  for (x2 : 0 .. )
    ...
    for (xn : 0 ..)
```

Loop Bounds Generation

```
for (i=0; i<9; i++)
   for (j=i; j<7 && j<i+4; j++)
         A[i,j]=0;
```

```
for (j= 0; j <= 6; j++)
  for (i = min (8,j); i <= max(j-3,0); i++)
```

$0 \le i$
$i \le 8$
$i \le j$
$i \geq j-3$

$j \geq 0$
$j \le 6$

- Desired order: j[outermost], i[innermost]
- Start with inner-most loop index i.
- Bring inequations to expected form
- Remove i and project S on remaining indices i.e. j
- Once projected: use inequation to identify UB & LB of inner index i.e. i (based on fixed value of outer index)

Changing Axes

- Can iterate horizontally, vertically and **diagonally (j-i, j).**
  - LB and UB on new axes: **new variable** `k = j-i` . Desired order `(k [inner index], j [outer index])`
  - Use same algo to find LB and UB for k and j, which uses Fourier Motzkin algo

$i \geq j-3$
$i \geq 0$
$i \leq j$
$j \geq 0$
$j \leq 7$
(j, i)

$j-k \geq j-3$
$j-k \geq 0$
$j-k \geq j$
$j \geq 0$
$j \leq 7$

$k \leq 3, j$
$k \geq 0$
$j \geq 0$
$j \leq 7$

```
j >= k
j <= 7
j >= 0
k >= 0
k <= 3
```

```
for (k= 0; k <= 3; j++)
  for (j =0; j <= 7; j++)
    A[j-k, j] =0;
```

- Traverse order : (k [outermost] , j [innermost]])
- Project on k, **eliminate j**
- Get LB/UB of outer loop on k

Module 132: Affine array Accesses

Data Dependencies

- Reordered iteration indices: Switching order of iteration space, Changing axes
- Memory access within each iteration & Data Dependencies as result of changing axes
  - Do 2 iterations access same memory location ? **Data Dependent. No reordering.**

```
for (i=0; i<10; i++)
  for (j=0; j<20; j++)
    A[(i+j)%20] = i*j;
```

$$\{(i,j,i',j') \mid \underbrace{(i+j)\%20 = (i'+j')\%20}_{\text{non affine}}\}$$

- Pair: Iteration 1: (i, j), Iteration 2: (i’, j’)
- 2nd part: Constraint for memory access
  - Affine constraint, affine expression ($C_i x_i + C_0$)
- Not affine : % is not covered by affine expression

```
for (i=0; i<10; i++)
  for (j=0; j<20; j++)
    A[2j+10-i] = i*j;
```

$$\{(i,j,i',j') \mid 2(j-j') = i-i'\}$$

Affine array Accesses

- Most programs have affine access to memory w.r.t. surrounding loop indices
- Array access in loop is affine, iff
  - Loop bounds are expressible as affine expressions of
    - *Surrounding loop indices & symbolic constants* (regular constants, loop invariant)
  - Index for each dimension of array (1D- A[i] or 2D- A[i,j]) access, is also affine expression on
    - *Surrounding loop indices & symbolic constants.*
    - Example: `2j+10-i`: Affine expression. `i+j %20`: Not affine expression
  - Examples: Indices i, j, k have affine bounds, n is loop invariant.

```
X [2 * i + j] : Affine

Y [i, j + n] : Affine

Z [ i+3*j+2, j] : Affine

X [i, j, k] : Affine
```

```
X [ i *j] : Not Affine

Y [ i* n + j] : Not Affine
```

- Not affine: Multiply with symbolic constant (which is not regular constant
- Affine: Addition to symbolic constant

Affine array Accesses

- Representation for 1 Array access in a loop nest: **4 tuple < F, f, B, b>**
  - **B and b** : Represent space of iteration space of polyhedron
  - **F and f** : Represent affine expression w.r.t loop iteration indices which specify multi-dimension address of memory access
  - If loop nest uses: a vector of index variable i, then `Bi + b >= 0` [Iteration space]
  - Accessed array element : `Fi + f` [Memory address]
    - F : Coefficient matrix: represents coefficient of each of loop indices (CiXi + C0). F-Ci, f-C0
- Example: Surrounding loop iteration indices vector $\begin{pmatrix} i \\ j \end{pmatrix}$

A[i-1]

$F = \begin{pmatrix} 1 & 0 \end{pmatrix} \quad f = \begin{pmatrix} -1 \end{pmatrix}$

$\begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \end{pmatrix}$

Examples:

- 2D access B[i, j] , Y[j, j+1] , Y[1,2]
- 3D access: Z[1, i, 2* i+j]

Affine array Accesses

- Linearized forms for multi-dimensional arrays : may be non-affine

$X[i,j] \longleftrightarrow X[i*n+j]$

affine — non-affine

- For polyhyderal analysis: prefer non-linearized affine representation
  - Common in image processing and neural networks
  - Easier to analyze and optimize
- Affine array access
  - Used to reason about data dependencies and resue characteristics.

Module 133: Data Reuse

Data reuse

- Reason about memory access. To identify memory footprint of each access
    - Find if 2 iterations are related, like if data dependency between them.
- Data reuse property: Identify sets of iteration that access same data or same cache line.
    - Can optimize for locality. Can bring those iteration close in execution time.
    - Useful for locality optimizations
    -
- Data dependence property
    - Identify access that refer to same memory location & at least one of them is a write.
    - For given 2 accesses: RAW, WAR, WAW.
        - Don't reorder iterations when these exist, as it will give different results.

Data reuse categories

- **Self reuse**
    - Multiple iterations of **same statement** access same data
- **Group reuse**
    - Two same iterations of **different statements** access same data
    - If different statements access same data in same iteration
- **Data Reuse (Temporal) is**
    - Temporal:
        - If **exact same data** is accessed multiple times across iterations.
        - Useful in cacheline or general locality.
    - Spatial:
        - If **different data** in same cacheline is accessed.
        - Useful in cacheline locality only.

Data Reuse

- ```
  float Z[n];
  for (i=0; i<n; i++) {
      for (j=0; j<n; j++) {
          Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/2;
  ```

- Self **Spatial** reuse
  - Each of Z[j], Z[j+1], Z[j+2] have self spatial reuse across different iterations.
  - In isolation, Z[j] has self spatial reuse in different iterations. High spatial locality.
  - 4 Different accesses considered as separate statements.
  - Z[j] likely to hit in same cache line
- Self **Temporal** reuse
  - Exact **same element** is accessed repeatedly once for **each iteration of outer loop**.
- **Group** spatial reuse
  - Z[j], Z[j+1] access same cache line
- **Group** temporal reuse
  - Across different iterations. Access by Z[j] would be accessed by Z[j+1] in next iteration.
  ```

Data Reuse

- ```
  float Z[n];
  for (i=0; i<n; i++) {
      for (j=0; j<n; j++) {
          Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/2;
  ```

1. No. of memory access = $4n^2$. For each iteration n^2 access. 4 accesses for each.
2. Memory footprint = n/c cache lines . c = cache line size. Distinct memory location order n.
   From 1 & 2. Pigeon-hole principle . Data reuse.

Reuse factors

- Factor of n: Due to self temporal reuse. $4n^2$ & n/c.
- Factor of c: due to self spatial reuse. Cache line access by same statement.
- Factor of 4: due to group temporal reuse.
  ```

Module 134: Self Reuse (Temporal)

Self Reuse (Temporal)

- Self reuse: Same element accessed for all iterations. F=(0), f=(0)

```
for (i=0; i<n; i++) {
      A[0]=0;
 }
```

- **Find relation between: F, f and reuse**

```
 -- No Self reuse --
for (i = 0; i < n; i++) {
  A[i] = 0; // Different element accessed.
}

 -- Self reuse --
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    A[i] = 0; // Same element for j.
  }
}

F = (1 0)  f= (0)
```

```
 -- Self reuse --
 -- 2D nest, 2D array access.
 -- Iteration sapce: 2D
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    A[i, 2*i] = 0; // Same loc for inner loop iter.
  }
}

F = (1 0)     f = (0)
    (2 0)         (0)

 -- No Self reuse --
 -- 2D nest, 2D array access.
 -- Iteration sapce: 2D
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    A[i, 2*i + j] = 0; // diff values across diff iter.
  }
}

F = (1 0)     f = (0)
    (2 1)         (0)
```

Self Reuse (Temporal)

```
-- Self reuse --
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    A[i + j, 2*i + 2* j] = 0;
  }
}
```

$$F = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} \quad f = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

- Self reuse reason:
    - For A(10, 20)
    - Order n indices: (0,10), (1,9)...
    - Multiple points in iteration space access same array location
- If data referenced by access has
    - k dimensions: Dimensionality of access. Example: A[i+j, 2i+2j] 1D space.
    - Access is nested in d-depth loop nest, where d > k, (loop nest depth=2, dim of access= 1)
    - **Then same data can be reused : n^{d-k} times**

Self Reuse (Temporal)

- Dimensionality of a reference ~ **Rank** of the coefficient matrix (F)
  - Self Reuse: Rank of coefficient matrix < dimensionality of loop iteration space
    - k < d : Reuse.
    - k !< d : No reuse.
- Find self reuse or not
  - Identify to find iterations i and i' (number of points in i and i') where
    - **Fi + f = Fi' + f** . Or **F(i - i') = 0**
  - If F is f**ull rank matrix**.
    - Only 1 trivial solution: **i = i'** . (same iteration and thus reuse). No non-trivial solution.
  - If F is **not full rank matrix** (rank of F < total dimension of matrix).
    - Other non-zero solution: Null space of F.

Self Reuse (Temporal)

- Full rank matrix. Dim of matrix = 2x2 and Rank of matrix= 2

$$F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

null space :

$i = 0$

$j = 0$ (empty)

No self reuse

- Rank of matrix = 1. Null space (non-empty): i =j. FX = 0 . Points of order n.

$$F = \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix}$$

null space:

$i - j = 0$

Has self reuse

Module 135: Self Spatial Reuse

Self Spatial Reuse

- Self spatial reuse: Different elements accessed in each iteration.

```
for (i=0; i<n; i++)
    A[i]=0;
```

- **Self reuse, temporal: No.** Same element not accessed in different iterations. F=(1). R=1, Dim=1.
- **Self spatial reuse: Yes.** A[0] in cache. A[1].. will hit same cache.

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        A[j,i]=0;
```

- **Self reuse, temporal: No.** Rank=2. Dim=2 $F = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
- **Self spatial reuse: Yes.**
    - If cache lines are accessed multiple times across different points in iteration space
    - Access (1,0) and (1, 1) are adjacent elements
    - Reuse distance for spatial locality large but spatial reuse.

Self Spatial Reuse

- To reason Self spatial reuse, need to know Size of cache line.
  - Approximation: Consider 2 array elements access. They share same cacheline iff
    - They differ only in last dimension of a d-dimension array.
    - Assuming: all elements in last dimension fit in a single cache line.

```
for (i = 0; i < n; i++) {
  A[i] = 0;
}
```

Last dimension: i. Belongs to 1 cache line. i removed, 0 dim access.

```
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    A[j,i] = 0;
  }
}
```

Remove i^{th} index. j fits in 1 cache line.

- Dimensionality of access = 1 (j index)
- Dimensionality of loop nest = 2
- 1 < 2. So self spatial reuse.

- For last dimension, accesses are cheaper, so approximation meaningful.

Self Spatial Reuse

- For self spatial reuse:
  - Truncate F by removing / Drop last row of coefficient matrix F. (New step in self spatial reuse)
  - Resulting coeff matrix is effective coefficient matrix.
  - If Rank of truncated matrix < depth of loop nest then self-spatial reuse.
- Significance of identifying self spatial reuse
  - It may be **possible to reorder computation** such that (we exploit spatial locality)
    - Innermost loop varies only the last coordinate of array
  - If self spatial reuse: Is it possible to reorder computation such that
    - **Reuse distance** between multiple accesses to same cacheline becomes close in exec order

Self Spatial Reuse and Spatial locality

- Innermost loop index: i.

Example

A[3, 2i, 7i+j] for iteration indices $\begin{pmatrix} i \\ j \end{pmatrix}$

Removing the last dimension yields

A [3, 2i]

which has rank = 1

Because rank < d (2), this access

```
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    A[3, 2i, 7i+j]
  }
}
```

Has Self spatial reuse. Not spatial locality

Iteration vector (0) (0) (.) (1)

- Innermost loop index: j .
  - Apart from self spatial reuse. Will also exhibit spatial locality. (innermost loop iterates on innermost dimension and it is in cache line.)
- Iteration vector should belong to nullspace of truncated F to obtain spatial locality
  - Null space should be non-trivial. Need more points than 0.

Self Spatial Reuse and Spatial locality

$$F = \begin{pmatrix}0 & 0\\ 2 & 0\end{pmatrix}$$

$$F = \begin{pmatrix}0 & 0\\ 0 & 2\end{pmatrix} \begin{pmatrix}0\\ 1\end{pmatrix}$$

- Innermost loop index: i. `A [3, 2i,` $\begin{pmatrix}0\\ 1\end{pmatrix}$ `+ j]`
  - Doesn’t satisfy the requirement
  - If use (i, j). j inner. In null space. Has spatial locality. Reuse
  - If use (j, i). i inner. Not in null space. No spatial locality. Reuse
  - For locality order of iteration space matters, if i inner or j. Not for reuse.

- A[3, 2i, i+7j] . Remove i+7j. For A[3, 2i], Rank=1. Spatial Reuse.
- A[2i, i+7j, 3] . Remove 3. For A[2i, i+7j]. Rank=2. Dimensionality=2. No spatial reuse.
- A[j, j, i] . Rank =1 < 2. Has spatial reuse. If i innermost, would have spatial locality.

Thank you

- Pankaj Gode