On recovering multi-dimensional arrays in Polly

Tobias Grosser, Sebastian Pop, J. Ramanujam, P. Sadayappan

ETH Zürich, Samsung R&D Center Austin, Louisiana State University, Ohio State University

19. January 2015

IMPACT'15 at HiPEAC 2005, Amsterdam, NL

Arrays

```
for i:
  for j:
    for k:
      A[i + p][2 * j][k + i] = ...
```

- Data structure
  - Collection of elements
  - Elements identified *n*-dimensional index
  - Element addresses can be directly computed from index
- Widely used
  - Core component of polyhedral model
  - Used in **real** programs

What is the problem?

Arrays are trivial, each programming language has native support for them!

Right?

A common way to represent multi-dimensional arrays

```
struct Array2D {
  size_t size0;
  size_t size1;
  float *Base;
};

#define ACCESS_2D(A, x, y) *(A->Base + (y) * A->size1 + (x))
#define SIZE0_2D(A) A->size0
#define SIZE1_2D(A) A->size1

void gemm(struct Array2D *A, struct Array2D *B,
          struct Array2D *C) {
L1:  for (int i = 0; i < SIZE0_2D(C); i++)
L2:    for (int j = 0; j < SIZE1_2D(C); j++)
L3:      for (int k = 0; k < SIZE0_2D(A); ++k)
           ACCESS_2D(C, i, j) +=
             ACCESS_2D(A, i, k) * ACCESS_2D(B, k, j);
}
```

C99 - The solution?

```
void gemm(int n, int m, int p,
          float A[n][p], float B[p][m], float C[n][m]) {
L1:  for (int i = 0; i < n; i++)
L2:    for (int j = 0; j < m; j++)
L3:      for (int k = 0; k < p; ++k)
           C[i][j] +=
             A[i][k] * B[k][j];
}
```

C99 arrays lowered to LLVM-IR

```
define void @gemm(i32 %n, i32 %m, i32 %p,
    float* %A, float* %B, float* %C) {
;for i:
; for j:
;  for k:
    %A.idx = mul i32 %i, %p
    %A.idx2 = add i32 %A.idx, %k
    %A.idx3 = getelementptr float* %A, i32 %A.idx2
    %A.data = load float* %A.idx3
    %B.idx = mul i32 %k, %m
    %B.idx2 = add i32 %B.idx, %j
    %B.idx3 = getelementptr float* %B, i32 %B.idx2
    %B.data = load float* %B.idx3
    %C.idx = mul i32 %i, %m
    %C.idx2 = add i32 %C.idx, %j.0
    %C.idx3 = getelementptr float* %C, i32 %C.idx2
    %C.data = load float* %C.idx3
    %mul = fmul float %A.data, %B.data
    %add = fadd float %C.data, %mul
    store float %add, float* %C.idx3
;  endfor k
; endfor j
;endfor i
}
```

LLVM sees polynomial index expressions

```
void gemm(int n, int m, int p,
          float A[], float B[], float C[]) {
L1:  for (int i = 0; i < n; i++)
L2:    for (int j = 0; j < m; j++)
L3:      for (int k = 0; k < p; ++k)
           C[i * m + j] +=
             A[i * p + k] * B[k * M + j];
}
```

Polynomial index expressions cause trouble

- Can not be modeled with affine techniques
- Block clearly beneficial loop-interchange in icc 15.0
  - Parametric version, not interchanged $\rightarrow$ 15s

```
void oddEvenCopyLinearized(int N, float *Ptr) {

#define A(o0, o1) Ptr[(o0) * N + (o1)]
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      A_(2 * j, i) = A(2 * j + 1, i);
}
```

Polynomial index expressions cause trouble

- Can not be modeled with affine techniques
- Block clearly beneficial loop-interchange in icc 15.0
  - Parametric version, not interchanged → 15s
  - Fixed-size version, interchanged → 2s

```
void oddEvenCopyLinearized(int N, float *Ptr) {
  N = 20000;
#define A(o0, o1) Ptr[(o0) * N + (o1)]
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      A_(2 * j, i) = A(2 * j + 1, i);
}
```

The Problem

Given *a set of* **single dimensional memory accesses** *with index expressions that are multivariate polynomials and a set of iteration domains,* **derive a multi-dimensional view***:*

- A multi-dimensional array definition
- For each original array access, a corresponding multi-dimensional access.

Conditions

- (**R1**) **Affine:**
  New access functions are affine
- (**R2**) **Equivalence:**
  Addresses computed by original and multi-dimensional view are identical
- (**R3**) **Within bounds:**
  Array subscripts for all but outermost dimension are within bounds

If (**R3**) not statically provable $\rightarrow$ derive run-time conditions.

An Optimistic Delinearization Algorithm

Guessing the shape of the array is `A[][P1][P2]` we:

1. Collect possible array size parameters
2. Derive dimensionality and array size
3. Compute multi-dimensional access functions
4. Derive validity conditions considering loop constraints

Example

- Initialize a multi-dimensional subarray
  - Size of the full array: $n_0 \times n_1 \times n_2$
  - Array to initialize starts at: $o_0 \times o_1 \times o_2$
  - Size of area to initialize: $s_0 \times s_1 \times s_2$

```
void set_subarray(float A[],
  unsigned o0, unsigned o1, unsigned o2,
  unsigned s0, unsigned s1, unsigned s2,
  unsigned n0, unsigned n1, unsigned n2) {

  for (unsigned i = 0; i < s0; i++)
    for (unsigned j = 0; j < s1; j++)
      for (unsigned k = 0; k < s2; k++)
S:        A[(n2 * (n1 * o0 + o1) + o2)
            + n1 * n2 * i + n2 * j + k] = 1;
}
```

Example

0) Start: $A[(n_2(n_1 o_0 + o_1) + o_2) + n_1 n_2 i + n_2 j + k]$

1) Expanded index expression:
$n_2 n_1 o_0 + n_2 o_1 + o_2 + n_1 n_2 i + n_2 j + k$

2) Terms with induction variables: $\{n_1 n_2 i, n_2 j, k\}$

3) Sorted parameter-only terms: $\{n_1 n_2, n_2\}$

4) Assumed size: `A[][n1][n2]`

Example

5) Inner dimension: divide by n_2
Quotient: $n_1 o_0 + o_1 + n_1 i + n_2 j$
Remainder: $o_2 + k$ $\rightarrow A[?][?][k + o_2]$

6) Second inner dimension: divide by n_1
Quotient: $o_0 + i$ $\rightarrow A[i + o_0][?][?]$
Remainder: $o_1 + j$ $\rightarrow A[?][j + o_1][?]$

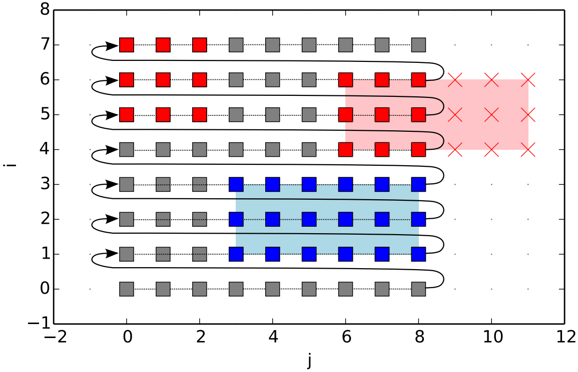
7) Full array access: $A[i + o_0][j + o_1][k + o_2]$

8) Validity conditions:

$$\begin{aligned}\forall i, j, k : \quad & 0 \leq i < s_0 \wedge 0 \leq j < s_1 \wedge 0 \leq k < s_2 : \\ & 0 \leq k + o_2 < n_2 \wedge 0 \leq j + o_1 < n_1 \wedge 0 \leq i + o_0 \\ & \Rightarrow o_1 \leq n_1 - s_1 \wedge o_2 \leq n_2 - s_2\end{aligned}$$

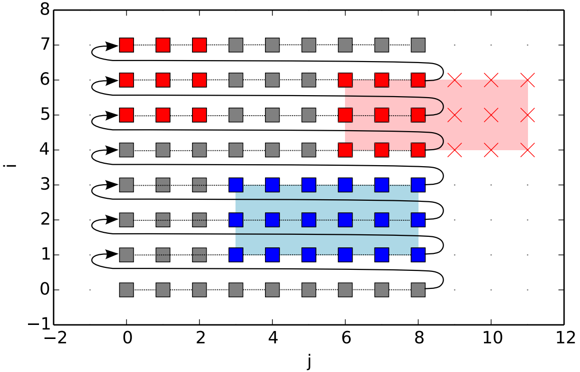
Why validity conditions?

- 2D array `A[n0][n1]` with $n_0 = 8 \wedge n_1 = 9$
- Access set blue
  - Parameters: $o_0 = 1 \wedge o_1 = 3 \wedge s_0 = 3 \wedge s_1 = 6$
  - Run-time condition: $o_1 \leq n_1 - s_1 \rightarrow 3 \leq 9 - 6 \rightarrow \top$



Why validity conditions?

- 2D array `A[n0][n1]` with $n_0 = 8 \wedge n_1 = 9$
- Access set red
  - Parameters: $o_0 = 4 \wedge o_1 = 6 \wedge s_0 = 3 \wedge s_1 = 6$
  - Run-time condition: $o_1 \leq n_1 - s_1 \Rightarrow 6 \leq 9 - 6 \Rightarrow \bot$
  - `A[6][9]` and `A[7][0]` alias ↯



Array shapes targeted with optimistic delinearization

- A[*][P_2][P_3] and A[*][P][P] $\Leftarrow$ Just presented
  - Multiple accesses
  - Array size parameters in subscript expressions
- A[*][$\beta_2 P_2$][$\beta_3 P_3$]
- A[*][$P_2 + \alpha_2$][$P_3 + \alpha_3$]

Size parameters in subscripts

```
float A[][N][M];
for (i = 0; i < L; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < M; k++)
S1:     A[i][j][k] = ...;

S2: A[1][1][1] = ...;
S3: A[0][0][M - 1] = ...;
S4: A[0][N - 1][0] = ...;
S5: A[0][N - 1][M - 1] = ...;
```

Size parameters in subscript - Offset expressions

```
float A[];
for (i = 0; i < L; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < M; k++)
S1:   A[i * N * M  + j * M + k] = ...;

S2: A[N * M + M + 1] = ...;
S3: A[M - 1] = ...;
S4: A[N * M - M] = ...;
S5: A[N * M - 1] = ...;
```

Size parameters in subscripts - Recovered array view

```
float A[][N][M];
for (i = 0; i < L; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < M; k++)
S1:   A[i][j][k] = ...;

S2: A[1][1][1] = ...;
S3: A[0][1][-1] = ...;
S4: A[1][-1][0] = ...;
S5: A[1][][-1] = ...;
```

Equivalent delinearizations

1) Equivalent delinearizations

$$\begin{aligned} & A[f_0][f_1] && \text{with } A[\;][s_1] \\ = \; & A[f_0 s_1 + f_1] && \text{with } A[\;] \\ = \; & A[(f_0 - k)s_1 + (ks_1 + f_1)] && \text{with } A[\;] \\ = \; & A[f_0 - k][ks_1 + f_1] && \text{with } A[\;][s_1] \end{aligned}$$

Equivalent delinearizations

1) Equivalent delinearizations

$$\begin{aligned} & A[f_0][f_1] && \text{with } A[\ ][s_1] \\ = \;& A[f_0 s_1 + f_1] && \text{with } A[\ ] \\ = \;& A[(f_0 - k)s_1 + (ks_1 + f_1)] && \text{with } A[\ ] \\ = \;& A[f_0 - k][ks_1 + f_1] && \text{with } A[\ ][s_1] \end{aligned}$$

2) How to model: `A[N * i + N + p]`

`A[i + 1][p]` valid only if $0 \leq p < N$

or

`A[i][N + p]` valid only if $-N \leq p < 0$

Equivalent delinearizations

1) Equivalent delinearizations

$$\begin{aligned} & A[f_0][f_1] && \text{with } A[\,][s_1] \\ = \; & A[f_0 s_1 + f_1] && \text{with } A[\,] \\ = \; & A[(f_0 - k)s_1 + (ks_1 + f_1)] && \text{with } A[\,] \\ = \; & A[f_0 - k][ks_1 + f_1] && \text{with } A[\,][s_1] \end{aligned}$$

2) How to model: `A[N * i + N + p]`

`A[i + 1][p]` valid only if $0 \leq p < N$

or

`A[i][N + p]` valid only if $-N \leq p < 0$

3) Apply a piecewise mapping:

$(f_0, f_1) \rightarrow (f_0 + k, -ks_1 + f_1) \mid \exists k : ks_1 \leq f_1 < (k+1)s_1$

Cover only a finite number of cases

- Covering all values of k requires polynomial constraints
- We can explicitly enumerate a fixed number of cases $[k_l, k_u]$
- Two cases are often enough: No parameter / One parameter

$$
(f_0, f_1) \rightarrow \begin{cases}
(f_0 + k_l, -k_l s_1 + f_2) & f_1 < k_l s_1 \\
\vdots & \\
(f_0 + (\text{-}1), \text{-}(\text{-}1)s_1 + f_2) & (\text{-}1)s_1 \leq f_1 < 0 \\
(f_0, f_1) & 0 \leq f_1 < 1s_1 \\
(f_0 + 1, \text{-}(1)s_1 + f_2) & 1s_1 \leq f_1 < 2s_1 \\
\vdots & \\
(f_0 + k_u, -k_u s_1 + f_2) & k_u s_1 \leq f_1
\end{cases}
$$

Delinearizing A[*]$[P_2 + \alpha_2][P_3 + \alpha_3]$

Original access: $A[f_0(\vec{i})][f_1(\vec{i})][f_2(\vec{i})]$

Original shape: $A[\ ][P_1 + \alpha_1][P_2 + \alpha_2]$

Linearized and expanded:

$$f_0(\vec{i})P_1P_2 + f_0(\vec{i})P_1\alpha_2 + f_0(\vec{i})P_2\alpha_1 + f_0(\vec{i})\alpha_1\alpha_2 + \\ f_1(\vec{i})P_2 + f_1(\vec{i})\alpha_2 + f_2(\vec{i})$$

Corresponding polynomial expression (grouped by parameters):

$$g_{\{1,2\}}(\vec{i})P_1P_2 + g_{\{1\}}(\vec{i})P_1 + g_{\{2\}}(\vec{i})P_2 + g_{\emptyset}(\vec{i})$$

Delinearizing A[*][$P_2 + \alpha_2$][$P_3 + \alpha_3$] - Match terms

- Assuming a parameter order, we can match terms.

2D

$$
\begin{aligned}
f_0(\vec{i}) &= g_{\{1\}}(\vec{i}) \\
f_1(\vec{i}) &= g_{\emptyset}(\vec{i}) - g_{\{1\}}(\vec{i})\alpha_1
\end{aligned}
$$

3D

$$
\begin{aligned}
f_0(\vec{i}) &= g_{\{1,2\}}(\vec{i}) \\
\alpha_2 &= g_{\{1\}}(\vec{i})/g_{\{1,2\}}(\vec{i}) \\
f_1(\vec{i}) &= g_{\{2\}}(\vec{i}) - g_{\{1,2\}}(\vec{i})\alpha_1 \\
f_2(\vec{i}) &= g_{\emptyset}(\vec{i}) - g_{\{2\}}(\vec{i})\alpha_2
\end{aligned}
$$

The general algorithm

1. Collect possible parameters
2. For each permutation of parameters
    2.1 Derive f_0
    2.2 Derive α-values
    2.3 Derive $f_i, i > 0$ expressions
    2.4 Derive run-time condition

Experimental Evaluation

Tested with our LLVM/Polly based implementation.

polybench

- 27 out of 29 kernels correctly delinearized
- run-time checks created for 5 benchmarks

Julia

- Delinearization* of a 2D gemm kernel

boost::ublas

- Delinearization* of a 2D gemm kernel

*Some loop invariant code motion needed.

Performance

dgemm implemented with boost::ublas

Compilers	linear	delin.	Speedup
icc	2.2	-	-
gcc	2.2	-	-
clang	2.2	-	-
clang + Polly	2.2	**1.2**	**1.8x**

Different Jula gemm kernels

Type	linear	delin.	Speedup
single float	13	**3**	**4.3x**
double float	14	**3**	**4.6x**
i16	7	**2**	**3.5x**
i32	13	**3**	**4.3x**
i64	15	**3**	**5x**
i128	22	**5**	**4.4x**

Conclusion

- Derived multi-dimensional array view from polynomial index expression
- Different shapes
  - A[*][P_2][P_3] and A[*][P][P]
    - Multiple accesses
    - Array size parameters in subscript expressions
  - A[*][$\beta_2 P_2$][$\beta_3 P_3$]
  - A[*][$P_2 + \alpha_2$][$P_3 + \alpha_3$]
- Optimistic approach handling insufficient static information