Integrating Data Layout Transformations with the Polyhedral Model

IMPACT 2019
January 23rd, 2019
Jun Shirako and Vivek Sarkar
Georgia Institute of Technology
Loop Transformations

• Change the statement order of program (i.e., loop structures)
  • Impact on temporal/spatial locality and parallelism
  • Use dependence analysis to identify legal transformations
  • Best loop transformation depend on hardware and data layout

• Large body of work since 1980’s, including
  • AST-based loop transformations
    • Loop fusion/distribution, permutation, skewing, tiling, and etc.
    • Sequence of individual transformations applied to AST
  • Polyhedral transformations
    • Linear algebraic framework to generalize loop transformations
    • Unified and formalized as affine scheduling problems
Loop Transformations

/// Input
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++)
S:  C[i][j] *= beta;
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j++)
        for (k = 0; k < nk; k++)
T:  C[i][j] += alpha * A[i][k]
    * B[k][j];

// Loop fusion
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++)
    for (k = 0; k < nk; k++)
      for (j = 0; j < nj; j++)
    T:  C[i][j] += alpha * A[i][k]
        * B[k][j];
}

// Loop permutation
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++)
S:  C[i][j] *= beta;
    for (i = 0; i < ni; i++)
      for (k = 0; k < nk; k++)
        for (j = 0; j < nj; j++)
          T:  C[i][j] += alpha * A[i][k]
            * B[k][j];

Polyhedral model: Unify arbitrary loop transformations as affine scheduling

\[ \Theta_S = \{ S(i, j) \to (0, i, 0, j) \} \]
\[ \Theta_T = \{ S(i, j, k) \to (0, i, 1, k, j) \} \]
Data Layout Transformations

- Change the memory layout of given (fragment of) program
  - Impact on spatial data locality of arrays/variables
  - Always legal transformations, as far as no over-write
  - Best layouts depend on program execution order and parallelism

- Various approaches proposed, including
  - Array dimensional permutations
    - Row-major vs. column-major selection for 2-D arrays
  - Data tiling combined with loop iteration tiling
    - Per-tile data elements are located closely in space
    - ~5.4x improvement on a 32-thread (4-socket) AMD Opteron [Reddy-ICS14]
  - Selection between Array-of-Struct and Struct-of-Array
    - Possibly different choices for different systems (e.g., CPUs vs. GPUs)
    - ~4.7x improvement on a 8-thread IBM POWER7 [Sharma-EuroPar15]
Data Layout Transformations

Goal: Unify arbitrary set of layout transformations via polyhedral model
Background: Polyhedral Compilation

• Polyhedral model
  • Algebraic framework for affine program representations & transformations
    • Unified view that captures arbitrary loop structures
    • Generalize loop transformations as form of affine transform

• Polyhedral representations (SCoPs)
  • Domain $D_{Si}$: set of statement instances for statement $Si$
  • Access $A_{Si}$: mapping a statement instance to array element(s) to be accessed
  • Schedule $\Theta_{Si}$: mapping a statement instance to lexicographical time stamp
    • Capture composition of loop transformations as a single affine mapping
Affine Representation of Data Layout Transformations

- Unification of various layout transformations as affine mapping
  - Affine scheduling problem to formalize layout transformations
    - As with schedule to generalize loop transformations
    - Additional legality constraints for valid data layout transformations
- Two types of layout representations
  - Array-based
    - Unit of mapping/transformation is an array element
    - Always legal as far as one-to-one mapping
  - Value-based
    - Unit of mapping/transformation is the value defined by a statement instance
    - Support broader range of data layout transformations, including storage expansion (i.e., privatization) and contraction
Array-based Data Layout Transformations

for (k = 0; k < nk; k++)
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
      S: C[i][j] += alpha * A[i][k] * B[k][j];

\[ D_C = \{ C(e_1, e_2) : 0 \leq e_1 < ni, 0 \leq e_2 < nj \} \]
\[ D_A = \{ A(e_1, e_2) : 0 \leq e_1 < ni, 0 \leq e_2 < nk \} \]
\[ D_B = \{ B(e_1, e_2) : 0 \leq e_1 < nk, 0 \leq e_2 < nj \} \]

- **Array domain** \( D_A \): set of elements for array A
  - \( A(e) \) to denote an element of array A
  - Lower and upper bounds of each dimension are affine combination of global parameters (constant value at beginning of runtime SCoP region)
Array-based Data Layout Transformations

- Layout $\Phi_A$: mapping array element $A(e)$ to memory space vector
  - To capture the relative position in the transformed memory space
  - Impose one-to-one mapping to avoid additional legality constraints
  - Data layout transformation = find a new layout mapping
Summary: Array-based Data Layout Transformations

- **Array element** \( A(e) \) as unit of representation/transformation
  - Array domain \( D_A \): define upper/lower bounds of dimensions
  - Layout \( \Phi_A \): map element \( A(e) \) to arbitrary transformed data layout
    - Individual array element \( A(e) \) has unique location specified by \( \Phi_A(A(e)) \)

- **Strength**
  - No additional legality constraints, assuming one-to-one mapping
  - Cover layout transformations to improve spatial locality
    - Array permutation, SoA/AoS conversion, data skewing, and data tiling

- **Weakness**
  - Not amenable to support many-to-one (contraction of memory space) and one-to-many (expansion/privatization for parallelism) transformations
  - Best layout \( \Phi_A \) can differ across statements that access \( A \)
    - Need data re-distribution with additional data transfer overhead
Value-based Data Layout Transformations

• Total data expansion [Feautrier-IJPP91]
  • Convert the input program into single-assignment form
  • Value: Unit of transformation
    • Producer: An statement instance \( S(i) \) defines the value
    • Consumers: One or more statement instances \( T_1(j_1), \ldots, T_n(j_n) \) use the value

• Dataflow
  • Relations between producer \( S(i) \) and consumers \( T_1(j_1), \ldots, T_n(j_n) \) are captured by dataflow analysis (i.e., \( j_1 = f_1(i_1), \ldots, j_n = f_n(i_n) \) )
  • Let \( flow_k \) denote k-th dataflow:

\[
flow_k = \{ S_k(\vec{i}) \rightarrow T_{k,1}(\vec{j}_1), \ldots, T_{k,n_k}(\vec{j}_{n_k}) \}
\]
Value-based Data Layout Transformations

• Loop transformations
  • Schedule: \( \Theta_S = \{ S(i) \rightarrow \text{time\_stamp\_vector} \} \)
    • \( S(i) \) is a statement instance
    • Capture sequential execution order of a program, i.e., loop structure
      • Loop transformations = find a new schedule map \( \Theta \)

• Data layout transformations
  • Layout: \( \Phi_S = \{ S(i) \rightarrow \text{memory\_space\_vector} \} \)
    • \( S(i) \) define a unique value to be used by consumers
      • Single-assignment form via total data expansion
    • Capture relative position in the transformed memory space, i.e., data layout
      • Layout transformations = find a new layout map \( \Phi \)
Legality of Value-based Data Layout Transformations

- **Value (k-th dataflow)**
  - Relations between producer $S_k(i)$ and consumers $T_{k,1}(j_1), \ldots, T_{k,n}(j_n)$
    \[
    flow_k = \{ S_k(i) \rightarrow T_{k,1}(j_1), \ldots, T_{k,n}(j_n) \}\n    \]

- **Legality**
  - Order of instructions: The producer of a value must precede any consumers of the value (producer-consumer requirement)
    \[
    \Theta(S_k(i)) < \text{lex}_{\min} (\Theta(T_{k,1}(j_1)), \ldots, \Theta(T_{k,n}(j_n)))\n    \]
  - Liveness of value: The memory location of a value must not be overwritten until the last use of the value (liveness requirement)
    - Given two values whose dataflows are $flow_k$ and $flow_l$:
      \[
      \text{lex}_{\max} (\Theta(T_{k,1}(j_1)), \ldots, \Theta(T_{k,n}(j_n))) \leq \Theta(S_l(i))
      \]
      \[
      \lor \text{lex}_{\max} (\Theta(T_{l,1}(j_1)), \ldots, \Theta(T_{l,n}(j_n))) \leq \Theta(S_k(i))
      \]
      \[
      \lor \Phi(S_k(i)) \neq \Phi(S_l(i))\n      \]
Summary: Value-based Data Layout Transformations

- **Value defined by** $S(\mathbf{i})$ **as unit of representation/transformation**
  - Total data expansion to convert into single-assignment form
  - Layout $\Phi_A$: map value to arbitrary transformed data layout

- **Strength**
  - Enable many-to-one (contraction) and one-to-many (expansion) transform
  - Cover layout transformations to improve spatial locality
    - Array permutation, SoA/AoS conversion, data skewing and tiling

- **Weakness**
  - Impose additional legality constraints to drastically increase complexity
  - (Currently) lack of efficient cost models and algorithms to co-optimize schedule and layout considering memory contraction/expansion
Code Generation via Schedule Tree

Layout map:

\( \Phi_C = \{ \ C(e_1, e_2) \rightarrow (0, e_1, e_2) \} \)
\( \Phi_A = \{ \ A(e_1, e_2) \rightarrow (1, e_2, 0, e_1) \} \)
\( \Phi_B = \{ \ B(e_1, e_2) \rightarrow (1, e_1, 1, e_2) \} \)

- **Schedule tree representation**
  - Straightforward to capture nested structures of data layout
  - Capable to compute total data size and relative offset to array element
    - Sequence node: \( size(sequence_k) = \sum_{i=0}^{\#children} size(child_node_{k,i}) \)
    - Band node: \( size(band_k) = length_k \times size(child_node_{k,(0)}) \)
    \( length_k = \max(range(band_k)) + pad_k \)
    - Leaf node: \( size(leaf_k) = 1 \) * impose same type for all arrays
#pragma scop
{
    for (i = 0; i < NI; i++)
        for (j = 0; j < NJ; j++)
            C[i][j] *= beta;

    for (k = 0; k < NK; k++)
        for (i = 0; i < NI; i++)
            for (j = 0; j < NJ; j++)
                C[i][j] += alpha * A[i][k] * B[k][j];
}

Layout transformation by:
\[
\Phi_C = \{ C(e_1, e_2) \rightarrow (0, e_1, e_2) \} \\
\Phi_A = \{ A(e_1, e_2) \rightarrow (1, e_2, 0, e_1) \} \\
\Phi_B = \{ B(e_1, e_2) \rightarrow (1, e_1, 1, e_2) \}
\]

// Dimension length
int len_0_0 = nj + pad;
int len_0 = ni;
int len_1_0 = ni + pad;
int len_1_1 = nj + pad;
int len_1 = max(nk, nk);

// Tree node size
int band_0_0 = len_0_0 * 1;
int band_0 = len_0 * band_0_0;
int band_1_0 = len_1_0 * 1;
int band_1_1 = len_1_1 * 1;
int seq_1 = band_1_0 + band_1_1;
int band_1 = len_1 + seq_1;
int seq_root = band0 + band_1;

// Allocation for new layout
double *field = malloc(seq_root * sizeof(double));

// Macro to access new layout
#define _C(e1, e2) field[(e1)*band_0_0 + (e2)]
#define _A(e1, e2) field[band_0 + (e2)*seq_1 + (e1)]
#define _B(e1, e2) field[band_0 + (e1)*seq_1 + \band_1_0 + (e2)]

// Data transfer (copy-in)
for (e1 = 0; e1 < ni; e1++)
    for (e2 = 0; e2 < nj; e2++)
        _C(e1, e2) = C[e1][e2];

// Original scop region
for (i = 0; i < NI; i++)
    for (j = 0; j < NJ; j++)
        _C(i, j) *= beta;

for (k = 0; k < NK; k++)
    for (i = 0; i < NI; i++)
        for (j = 0; j < NJ; j++)
            _C(i, j) += alpha * _A(i, k) * _B(k, j);

// Data transfer (copy-out)
...
Preliminary Results for Loop and Data Layout Co-optimizations

• Platforms
  • 12-core 2.8GHz Intel Xeon (Westmere) with Intel C/C++ compiler v15.0
  • 24-core 3.0GHz IBM POWER8 with XL C/C++ compiler 13.1

• Benchmarks: PolyBench 4.2
  • 22 benchmarks (total 29 benchmarks) whose kernels are n-dimensional loops working on m-dimensional arrays (n > m)
  • Data copy-in / copy-out were part of measured execution time

• Experimental variants
  • Minimum distance schedule (PLUTO algorithm) + best layout
    • Compute schedule for original layout; and then manually search best layout
  • PolyAST [Shirako-SC14] + best layout
    • Same as first variant, with different scheduler
  • Iterative search (co-optimization)
    • Iterates through different layouts and apply PolyAST loop transformation in each case; and find the globally best solution.
Performance on 12-core Intel Xeon Westmere

Geometric mean improvement: 1.21x over PolyAST + best layout
Performance on 24-core IBM POWER8

Geometric mean improvement: 1.24x over PolyAST + best layout
Conclusions

• Affine representation of data layout transformations
  • Array-based layout transformations
    • No additional legality constraints to be imposed
  • Value-based layout transformations
    • Support many-to-one (contraction) / one-to-many (expansion) transformations

• Preliminary integration of loop and data layout transformations
  • Iterates candidate layouts and compute best loop transformation in each
  • Select the globally best solution based on memory and computational cost
    • 1.21x / 1.24x geometric mean speedup on 12-core Xeon / 24-core POWER8

• Future work
  • Continue the work on cost-driven integration for array-based layout transformations
    • Comparison with the optimal solution by runtime exhaust search
    • Extensions and evaluations on GPU architectures
  • Develop heuristic to co-optimize schedule and value-based layout transformations