Combining Polyhedral and AST Transformations in CHiLL

Huihui Zhang, Anand Venkat, Protonu Basu, Mary Hall
University of Utah
January 19, 2016
Outline

• Introduction
  • Problem
    • Limitations of polyhedral transformation
  • CHiLL Compiler Abstractions
    • Combining polyhedral and AST transformations

• Case Studies
  • Inspector/executor transformation for sparse matrix computation
  • Partial sum transformation for stencil optimization
  • Parallel code generation
    • CUDA
    • OpenMP

• Related Work

• Conclusion
Introduction

• Limitation of typical polyhedral transformation
  • Limited to affine domain
  • Transform iteration spaces
  • Array indices of statements updated

• Complicated optimizations
  • AST transformation as a post-pass outside of polyhedral framework

• Challenges
  • Leverage the power of composability of polyhedral framework
CHiLL Compiler Abstractions

- **Input code**: for(i=0; i < n; i++)
  - s0: a[i+1] = a[i] + 5;

- **CHiLL Abstractions**:
  - **Statement**: s0: a[i+1] = a[i] + 5;
  - **IS**: \{i : 0 <= i < n\}
  - **xform**: \{i\}->[0,i+4,0]
  - **Polyhedral code**:
    - a[i+1] = a[i] + 5;

- **Generated code**:
  - for(i=4; i < n+ 4; i++)
    - s0: a[i-3] = a[i-4] + 5;

**CHiLL Compiler**
- **Loop transformation framework**
- **Code generation**

**Statement remains unmodified, other than array indices update**

**xform_inv = \{i\}->[i-4]\}

- Introduction
- Problem
- CHiLL Compiler Abstractions
- Case Studies
- Related Work
- Conclusion
CHiLL Compiler Abstractions

Input code:
for(i=0; i < n; i++)
s0: a[i+1]=a[i] + 5;

CHiLL Abstractions:
Statement: s0: a[i+1]=a[i] + 5;
IS: {[i] : 0 <= i < n}
xform: {[i] -> [0,1,0]}
code: Polyhedral

AST
Modified AST
Dep: <+1>

CHiLL Compiler
Loop transformation framework
Code generation

Generated code

Input code

• Introduction
  • Problem
  • CHiLL Compiler Abstractions
• Case Studies
• Related Work
• Conclusion
Non-Affine Extension – Coalesce Transformation

- Sparse matrix computation
  - Non-affine indirection through index arrays
    - Subscript expressions
      - $x[col[j]]$
    - Upper/lower loop bounds
      - $\text{index}[i], \text{index}[i+1]$
- Uninterpreted function symbol abstraction
  - Model functions or mappings (non-affine)
- Inspector/Executor mechanism
  - Inspector collects information at runtime used by optimized executor

CSR:
```
for(i=0; i < n; i++)
  for(j=\text{index}[i]; j < \text{index}[i+1]; j++)
    y[i]+=a[j]*x[col[j]]
```
Inspector Construction - Coalesce Transformation

Input code:
for(i=0; i < n; i++)
    for(j=index[i]; j<index[i+1]; j++)
        y[i]+=a[j]*x[col[j]]

AST & Iteration Space Manipulation

T_coalesce =\{[i,j]\rightarrow[k] | k=c(i,j) \land 0 \leq k < NNZ\}

Executor code:
for (k = 0; k < NNZ; k++)
    code

Statement update

y[c_inv[k][0]] += a[c_inv[k][1]]*x[col[c_inv[k][1]]];

Inspector code:
for(i = 0; i < n; i++)
    for(j = index[i]; j < index[i+1]; j++)
        code
c.create_mapping(i,j);

AST

Composability preserved; Dependence graph incrementally updated

The input code’s AST is modified to setup the data structures to represent the functionality of the inspector

- Introduction
- Case Studies
  - Inspector/Executor
  - Partial Sum
  - Parallel Code Generation
- Related Work
- Conclusion
More Complicated I/E Transformations - BCSR

**Input code:**

```c
for(i = 0; i < n; i++)
    for(j = index[i]; j < index[i+1]; j++)
        y[i] += a[j]*x[col[j]];
```

**Inspector Code:**

```c
for(ii=0; ii < n/r; ii++)
    for(i=0; i < r; i++)
        for(j=index[i*r+i]; j < index[i*r+i+1]; j++) {
            //reset marked to false (code not shown)
            code
            kk = col[j]/c; k=col[j]/c – kk*c;
            if(marked[kk] == false){
                marked[kk] = true;
                explicit_index[kk] = count;
                //initialize a'[count][0-r][0-c] to 0
                count++;
            }
            a'[count][i][k] = a[j];
            offset_index[ii+1] = count;
        }
```

**Inspector/Executor**

```c
for(k = 0; k < n; k++)
    for(i=0; i < n; i++)
        for(j = index[i]; j < index[i+1]; j++)
            if(k == col[j])
                y[i] += a[j]*x[k];
```

**Parallel Code Generation**

```c
for(ii=0; ii < n/r; ii++)
    for(kk=0; kk < n/c; kk++)
        for(i=0; i < r; i++)
            for(k=0; k < c; k++)
                for(j=index[ii*r+i]; j < index[ii*r+i+1]; j++)
                    if(kk*c+k == col[j])
                        y[ii*r+i] += a[j]*x[kk*c+k];
```

**Related Work**

**Conclusion**

**Composable with other transformations**
Partial Sum Transformation – Stencil Optimization

- Constant-coefficient Stencils
  - Weighted sum

- High-order Stencils

- $p = 2$
- $p = 4$
- $p = 6$
- $p = 10$
Partial Sum Transformation - Reuse

```
for (j=0; j<N; j++)
for (i=0; i<N; i++) {
    out[j][i] =
        w1*( in[j-1][i] + in[j+1][i] +
             in[j][i-1] + in[j][i+1] ) +
        w2*( in[j-1][i-1] + in[j+1][i-1] +
             in[j-1][i+1] + in[j+1][i+1] ) +
        w3*( in[j][i] );
}
```

• Composable with communication-avoiding optimizations
  • Overlapped tiling
  • Loop fusion
  • Wavefront

**Still affine**

```
r1 = in[j][i+1];
r2 = in[j+1][i+1] + in[j-1][i+1];
R[i] = w1 * r1 + w2 * r2; ---- 1
C[i+1] = w3 * r1 + w1 * r2; ---- 2
L[i+2] = R[i]; ------------------ 3
out[j][i] = L[i] + C[i] + R[i];
```

AST structure reparsed; Dependence graph rebuilt

- Introduction
- Case Studies
  - Inspector/Executor
  - Partial Sum
  - Parallel Code Generation
- Related Work
- Conclusion
Parallel Code Generation

• Introduces
  • Parallel threads
  • Synchronization
  • Scaffolding code

• Approach
  • Apply transformations to set up for parallelization
    • E.g., tiling, datacopy
  • Annotate AST with aspects of parallel code generation
  • AST and polyhedral abstractions preserved until code generation, to facilitate composing transformations
  • Code generation emits specialized code
Parallel Code Generation - CUDA

```c
void MM(int c[N][N], int a[N][N], int b[N][N]) {
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                c[j][i] = c[j][i] + a[k][i] * b[j][k]; }
```

- **Impact to AST**
  - AST annotation of block/thread loops
  - Loops are marked for elimination
  - Polyhedral and AST abstractions remain until code generation

```c
for(t2 = 0; t2 <= 7; t2++) // loop ii, block dimension x{
    for(t4 = 0; t4 <= 15; t4++) // loop jj, block dimension y{
        for(t6 = 128*t2; t6 <= 128*t2+127; t6++) // loop i {
            for(t8 = 64*t4; t8 <= 64*t4+63; t8++) // loop j {
                for(t10 = 0; t10 <= 1023; t10++) // loop k {
                    s0(t2,t4,t6,t8,t10); }}}}}}
```

```c
cudaize(0,"mm_GPU",{}, {block="ii","jj"},thread="i","j"},{});
```
void MM(int c[N][N], int a[N][N], int b[N][N]) {
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                c[j][i] = c[j][i] + a[k][i] * b[j][k];
}

• Impact to AST
  • AST annotation of block/thread loops
  • Loops are mark for elimination
  • Polyhedral and AST abstractions remain until code generation
  • Loop iterators are replaced with block/thread index
    • Eg, ii, jj replaced with blockIdx.x, blockIdx.y

for(t2 = 0; t2 <= 7; t2++) // loop ii, block dimension x{
    for(t4 = 0; t4 <= 15; t4++) // loop jj, block dimension y{
        for(t6 = 128*; t6 <= 128*+127; t6++) // loop i {
            for(t8 = 64*; t8 <= 64*+63; t8++) // loop j {
                for(t10 = 0; t10 <= 1023; t10++) // loop k {
                    s0(,t2,t4,t6,t8,t10); }}}}

for(t12 = 0, t2 <= 7, t2++) // loop ii, block dimension x{
for(t4 = 0, t4 <= 15, t4++) // loop jj, block dimension y{
for(t6 = 128*, t6 <= 128*+127, t6++) // loop i {
for(t8 = 64*, t8 <= 64*+63, t8++) // loop j {
for(t10 = 0, t10 <= 1023, t10++) // loop k {
    s0(,t2,t4,t6,t8,t10); }}}}

cudaize(0,"mm_GPU",{}, {block={"ii","jj"},thread={"i","j"}})
Parallel Code Generation - CUDA

for (kk = 0; kk <= 63; kk += 1) {
    for (tmp_tx = 0; tmp_tx <= 7; tmp_tx += 1)
        for (k = 16 * kk; k <= 16 * kk + 15; k += 1)
            c[...][...] = c[...][...] + a[...][...] * b[...][...];
    __syncthreads();
}

for (kk = 0; kk <= 63; kk += 1) {
    for (tmp_tx = 0; tmp_tx <= 7; tmp_tx += 1)
        for (k = 16 * kk; k <= 16 * kk + 15; k += 1)
            c[...][...] = c[...][...] + P1[...][...] * b[...][...];
    __syncthreads();
}

... mm_GPU <<<dimGrid0, dimBlock0 >>>(...);
... __global__ void mm_GPU(...) {
    ...
}

- Data Copy Transformation
- Synchronization
  - AST annotation
- Scaffolding code

- Introduction
- Case Studies
  - Inspector/Executor
  - Partial Sum
  - Parallel Code Generation
    CUDA
    OpenMP
- Related Work
- Conclusion
Parallel Code Generation - OpenMP

```c
#pragma omp parallel private (...) num_threads(6) {
    tid=omp_get_thread_num();
    for (k=-3; k<=66; k++) {
        loop jj
        for (t=0; t<=min(3,intFloor(t+3,2)); t++) {
            for (j=t-3; j<=t+66; j++) {
                for (i=t-3+intMod(-k-color-j-(t-3),2); i<=t+66; i+2) {
                    S0(t,k-t,j,i); /* Laplacian */
                    S1(t,k-t,j,i); /* Helmholtz */
                    S2(t,k-t,j,i); /* GSRB */
                }
            }
        }
    }
}
```

**AST Manipulation**
- Tile, then control loop marked for elimination
- Loop bound and statements update
- OpenMP directives
- Additional code
  - Synchronization and thread index

```c
// Explicit Spin Lock
zplanes[tid] = t2;
if (left != tid)
{ while(zplanes[left] < t2) {
    _mm_pause();
} } else{}
if (right != tid)
{ while(zplanes[right] < t2) {
    _mm_pause();
} }//end k 
```

**Case Studies**

**Inspector/Executor**
- Partial Sum

**Parallel Code Generation**
- CUDA
- OpenMP

**Related Work**

**Conclusion**
Related Work

• *J. Shirako SC’14*: *Oil and water can mix: An integration of polyhedral and ast-based transformations*
  - Decoupled framework
  - Need to extract dependence information between stages
  - Polyhedral stage limited to affine domain

• *T. Grosser TOPLAS’15*: *Polyhedral ast generation is more than scanning polyhedra*
  - User supplied AST expressions
  - Elegant for CUDA code generation
  - Expressing more complicated optimizations and data structures such as I/E transformation?
## Conclusion

- A broader class of optimizations supported by combining polyhedral and AST transformations

<table>
<thead>
<tr>
<th>Optimization techniques</th>
<th>AST transformations</th>
<th>Polyhedral transformations</th>
<th>Composable with other optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspector/executor for sparse codes</td>
<td>• Linked list struct in AST</td>
<td>• Encode sparse iteration space of executor</td>
<td>• Datacopy, scalar expansion</td>
</tr>
<tr>
<td></td>
<td>• Parse if condition in AST and convert to relation</td>
<td>• Derive closed form</td>
<td>• Tiling and unrolling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partial sums for high-order stencils</td>
<td>• Create partial sum buffers</td>
<td>• Create iteration spaces</td>
<td>• Fusion, distribution</td>
</tr>
<tr>
<td></td>
<td>• Create new statements</td>
<td>• Lexicographical ordering</td>
<td>• Skewing</td>
</tr>
<tr>
<td></td>
<td>• Delete existing statements</td>
<td>• New dependence graph</td>
<td>• Permutation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel code generation</td>
<td>• Eliminate certain loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Update statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Synchronizations</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Kernel launch/OMP clause</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Introduction
- Case Studies
- Related Work
- Conclusion