Non-affine Extensions to Polyhedral Code Generation

Anand Venkat, Manu Shantharam, Mary Hall and Michelle Strout

2014 International Symposium on Code Generation and Optimization (CGO)
Polyhedral Transformations & Code Generation

**Stage 1:**
*Loop Bounds Extraction & Iteration Space Construction*

- **Input Code:**
  
  ```c
  for(i=0; i < n; i++)
  s0: a[i]=b[i];
  ```

- **Iteration Space (IS):**
  
  ```
  s0 =[[i] : 0 <= i <= n]
  ```

**Stage 2:**
*Transformation (T) Application (Eg. Loop shifting)*

- **Input IS:**
  
  ```
  [[i] : 0 <= i <= n]
  ```

- **Transformation (T):**
  
  ```
  T = {[i]->[i+4]}
  ```

**Stage 3:**
*Original Loop Iterators obtained as functions of new iterators*

- **Update statement macro with T_inv. Apply Polyhedra Scanning**

- **Output IS:**
  
  ```
  [[i] : 4 <= i < n + 4]
  ```

- **Output Code:**
  
  ```c
  for(i=4; i < n+ 4; i++)
  s0: a[i-4]=b[i-4];
  ```
Motivation

• Limitation of the Polyhedral Model
  • Loop bounds, array access expressions and transformations must be affine, i.e. of the form: \( a_0 + a_1 x_1 + a_2 x_2 + ... + a_n x_n \)

• Important non-affine construct:
  Indirection through index arrays such as \( B[i] \) in \( A[B[i]] \)
  • Common in sparse matrix and molecular dynamics computations
  • Compiler cannot determine memory access patterns statically

• Key observations:
  Non-affine iteration spaces/accesses can sometimes be tolerated
  Run-time inspection reveals mapping of iterations to array indices
  • Enables locality and parallelizing run-time transformations
Sparse Matrix-Vector Multiply (SpMV)

- Sparse matrix computations
  - Avoid redundant computation and space for zero-valued elements
  - Results in non-affine index arrays to derive column and row
  - SpMV libraries support multiple matrix formats and parallelization strategies to exploit matrix structure (e.g., CUSP for GPUs)

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

Non-affine loop bounds

Compressed Sparse Row (CSR) format

Non-affine subscript
Related Work

*Run-time Approaches for Sparse Matrix Vector Multiply (SpMV)*

- Basumallik and Eigenmann (PPoPP’06)
  - Use a loop restructuring run-time transformation on irregular loops

- Ravishankar et al. (SC’12)
  - Generate run-time I/E code for partitioning irregular loops on a distributed memory system

*Non-affine Polyhedral Abstractions*

- Pugh and Wonnacott (‘94)
  - Represent non-affine accesses for array dependence analysis

- Strout et al. (LCPC’12)
  - Represent run-time Inspector/Executor (I/E) transformations as non-affine transformations
Contributions

1. Represent iteration spaces for non-affine loop bounds
   • Enables further iteration space transformations

2. Support non-affine transformations using run-time inspection

3. Simplify array access expressions resulting from non-affine mappings

4. Demonstrate high-performance compiler-generated code on GPU
   • Performance of Sparse Matrix Vector Multiply (SpMV) kernel comparable to manually-tuned CUSP library
Non-affine Loop Bounds

Loop Bounds
Extraction & Iteration
Space Construction

a) Without Extension

SpMV Code:
for(i=0; i < n; i++)
s0: for(j=index[i];j<index[i+1];j++)
y[i]+=a[j]*x[col[j]]

b) With Extension

SpMV Code:
for(i=0; i < n; i++)
for(j=index[i];j<index[i+1];j++)
s0: y[i]+=a[j]*x[col[j]]

Inner j-loop bounds abstracted as
index(i) & index(i+1)

Iteration Space (IS):
{[i,j] : 0 <= i < n && index(i) <= j &&
 j < index(i+1)}
Non-affine Loop Bounds

- Un-interpreted Function Symbols
  - “Un-interpreted” as exact function mapping is not known
- Use to represent non-affine loop bounds in the iteration space
  - Enables other iteration space transformations (e.g., tiling)

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

```c
for (i = 0; i <= n; i ++)
    for (jj = index[i];jj<index[i+1];jj+=4)
        for (j = jj; j <min(index[i+1],jj + 4); j += 1)
            y[i] += (a[j] * x[col[j]]);
```
Non-affine Transformations

- Generalized loop coalescing transformation
  - Flatten a multi-dimensional loop nest into a single loop

\[ T_{\text{coalesce}} = \{ [i,j] \rightarrow [k] \mid k = c(i,j) \land 0 \leq k < \text{NNZ} \} \]

- Benefit
  - Enables other transformations (e.g., longer vectors, more tiling)
Non-affine Transformations

- Mapping from input loop iterators to output loop iterator determined at run-time

- An **Inspector** records this mapping
  - Code with updated references is termed the **Executor**

- Code Generation utilizes the run-time map constructed for the un-interpreted function to “fill-in” the inverse mapping
  - Eg. \( i = \text{c\_inv}[k][0] \) & \( j = \text{c\_inv}[k][1] \)

**Inspector Data Structure:**

```c
struct access_relation {
    // array to track old iterators
    int c_inv[][2];
    // variable to keep track of k
    int k;
    void create_mapping(int i, int j) {
        c_inv[k][0] = i;
        c_inv[k][1] = j;
        k++;
    }
}
```

**Inspector Code:**

```c
struct access_relation c;
for (i=0; i<=n-1; i++)
    for (j=index[i]; j<=index[i+1]-1; j++)
        c.create_mapping(i,j);
```

**Executor Code:**

```c
for (k = 0; k < NNZ; k++)
y[\text{c\_inv}[k][0]]
+= A[\text{c\_inv}[k][1]]*x[\text{col}[\text{c\_inv}[k][1]]];
```
Non-affine Transformations

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

Copy Input Loop IS to Inspector’s IS

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

Set Coalesced Loop as Executor’s IS

Copy Input Loop statement code to executor

Executor Code :
for (k = 0; k < NNZ; k++)
  y[c_inv[k][0]] += A[c_inv[k][1]]*x[col[c_inv[k][1]]]
Optimizations

- Un-simplified Output Loop:
  
  \[
  \text{for (}k = 0; \; k < \text{NNZ}; \; k++\text{)} \\
  y[\text{c}^{-1}[k][0]] \; +\!\!\!\!\!\; A[\text{c}^{-1}[k][1]] \!\times \! x[\text{col}[\text{c}^{-1}[k][1]]];
  \]

- Array access Indirection incurs extra memory load instruction overheads

- Inspector provides additional information that iterator \(j\) in input loop is equal to iterator \(k\) in output loop

  \[
  T_{\text{coalesce}} = \{[i,j] \rightarrow [k] \mid k = c(i,j) \; \wedge \; 0 \leq k < \text{NNZ} \} \; \wedge \; j=k
  \]

- Inverse mapping simplification results in optimized Output Loop:

  \[
  \text{for (}k = 0; \; k < \text{NNZ}; \; k++\text{)} \\
  y[\text{c}^{-1}[k][0]] \; +\!\!\!\!\!\; A[k] \!\times \! x[\text{col}[k]];
  \]
CUDA-CHiLL: Transformations for GPU

CHiLL: Loop Transformations

Omega+: Linear Constraint Solver

CodeGen+: Polyhedra Scanning

Iteration Spaces+ Mappings

IeGenLib: Non-affine Constraint Solver

Lua Script: tile... cudaize...

Input Code: for...
GPU Optimization Strategies

- GPUs are massively multithreaded
  - Compiler should expose as many parallel computations as possible

- Optimize for memory coalescing => Adjacent threads accessing contiguous memory locations increase effective memory bandwidth

- Compiler interfaces transformed code with architecture-specific reduction library routines
  - Tiling transformations allow controlling the granularity of the reduction

- Parallel reductions desired as they
  - Increase degree of parallelism
  - Improve memory coalescing
Case Study : SpMV

• Highly optimized parallel derivations for the SpMV kernel targeting GPUs
• Parallelization Strategies as in Bell and Garland (SC’09)
SpMV CSR Scalar

*a. CSR Scalar Script*

```plaintext
tile_by_index(0, \{"i"\}, \{Ti\}, \{l1_control="ii"\}, \{"ii", "i", "j"\})

cudaize(0,"spmv_GPU, \{a=NNZ, x=N, y=N, col=NNZ, index=NNZ\}, \{block=\{"ii\}, thread=\{"i\}\},\})
```

*b. CSR Scalar Code*

```c
__global__ void spmv_GPU (float *y, float *a, float *x, int *col, int *index){
    if(tx <= NROWS – Ti*bx – 1)
        for(j=index(Ti*bx + tx);
            j <= index__(Ti*bx + tx) – 1; j+=1)
            y[Ti*bx + tx] += (a[j]*x[col[j]]);
}
```
SpMV CSR Vector

- Tiling for parallel row computations
- Second tiling for intra warp parallelization within row

c. CSR Vector Script

```cpp
tile_by_index(0,{"i"},{Ti}, {l1_control="ii"},{"ii","i","j"})CU=1
tile_by_index(0,{"j"},{Tj},{l1_control="jj",l1_tile="j"}, {"ii","i","j","jj"},strided)CU=1
scalar_expand_by_index(0,{"i","j"},"RHS", CP_TO_SHARED, NO_PAD,ACCUMULATE_THEN_ASSIGN)
cudaize(0,"spmv_GPU",{ a=NNZ,x=N,y=N, col=NNZ,index=NNZ},
{block="ii", thread=('j', "i")},{});
reduce_by_index(0,{"jj"}, "reduce_warp",{}, tx)```

d. CSR Vector Code

```cpp
#define index_(i) index[i]
#define index__(i) index[i + 1]
__global__ void spmv_GPU(float *y, float *a, float *x, int *col, int *index) {
...
__device__ __shared__ float _P1[TILESZ*WARPSZ];
if (ty <= NROWS - TILESZ* bx - 1) {
    if (tx <= index__(ty + TILESZ* bx) - index_(ty + TILESZ* bx) - 1) {
        _P1[tx + ty * WARPSZ] = 0;
        if (tx <= index__(ty + TILESZ* bx) - index_(ty + TILESZ* bx) - 1) {
            for (jj = index_(ty + TILESZ* bx); jj <= -tx + index__(ty + TILESZ* bx) - 1; jj += WARPSZ)
                p1[tx + ty * WARPSZ] += (a[tx + jj] * x[col[tx + jj]]);
            reduce_warp(&y[ty + TILESZ* bx],&_P1[tx + ty * WARPSZ],
                _Lt(31,index__(ty + TILESZ* bx) - index_(ty + TILESZ* bx) - 1));
        }
    }
}}
```
SpMV CSR Vector

for (i=0; i < n; i++)
    for (j=index[i]; j<index[i+1]; j++){
        $T[j] = a[j] * x[col[j]]$;
        y[i] += $T[j]$
    }

Product Expression is scalar expanded

Threads:


Shared Memory Reduction
SpMV COO

• Input loop is coalesced and then tiled
  • Each block consists of multiple warps
  • Each warp reduces a partition of non-zeros
  • Rows may span across warp boundaries

• Non-zeros on warp boundaries defer write to global memory to avoid data races
  • Peeling and distribution utilized to separate the update across boundaries from interior points
  • Second level reduction accounts for boundary updates
SpMV COO

Block₀

Warp₀

TID : 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
VAL : 0 9 1 3 5 9 5 8 2 3 6 8 2 4 6 8 1 2 3 4 5 6 7 8 1 2 4 5 6 7 8 9

Defer write to second level

spmv_first_level_gpu

Block₁

Warp₁

Warp₂

Warp₃

TID : 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
VAL : 0 9 1 3 5 9 5 8 2 3 6 8 2 4 6 8 1 2 3 4 5 6 7 8 1 2 4 5 6 7 8 9

Write to global memory

Write to global memory

Write to global memory

TID : 0 1 2 3
VAL : 8 8 16

spmv_second_level_gpu
Experiments

• Experiments conducted on Nvidia Tesla C2050 Fermi
  • 14 Streaming Multiprocessors, 32 cores per SM.
  • 1 GB of global memory, 64KB register file per SM.
  • We compare performance of generated code to the corresponding CUSP implementation

• Matrices chosen were from the UFL Sparse Matrix Collection
Methodology

• CSR Scalar
  • Each CUDA thread processes 1 row
  • Auto-tuned for different configurations of threads per block

• CSR Vector
  • 2-dimensional blocks
    • 1st dimension for threads per block
    • 2nd dimension for No. of non-zeros within a row being reduced
  • 2nd dimension tuned based on input matrix row length

• COO
  • Optimizations
    • Indirection elimination
    • Padding to eliminate control flow
Results

CSR Scalar

- Speedup compared to CUSP
- No. of Threads per Block: 1024, 256, 64

Graph showing performance comparison between CSR Scalar and CUSP for various matrices.

Table 1: A suite of unstructured test matrices.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>No. of Rows</th>
<th>Nonzeros</th>
<th>Average Nonzeros per Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac_econ_fwd500</td>
<td>78,134</td>
<td>55,4</td>
<td>50.6</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>217,918</td>
<td>53.3</td>
<td>64.1</td>
</tr>
<tr>
<td>conph</td>
<td>140,874</td>
<td>55.4</td>
<td>50.6</td>
</tr>
<tr>
<td>fwd500</td>
<td>170,998</td>
<td>53.3</td>
<td>64.1</td>
</tr>
<tr>
<td>econ</td>
<td>206,500</td>
<td>55.4</td>
<td>50.6</td>
</tr>
<tr>
<td>scircuit</td>
<td>46,835</td>
<td>5.6</td>
<td>5.6</td>
</tr>
<tr>
<td>shipsec1</td>
<td>64,100</td>
<td>5.6</td>
<td>5.6</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>53.3</td>
<td>64.1</td>
</tr>
</tbody>
</table>

The table lists matrices and their properties, such as the number of rows, nonzeros, and average nonzeros per row.
Results

Figure 6: Speedup of CSR Scalar generated code compared to CUSP.

Figure 7: Speedup of CSR Vector generated code compared to CUSP.

Table 1: A suite of unstructured test matrices.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>N</th>
<th>NNZ</th>
<th>Base (32, 16)</th>
<th>Autotuned</th>
</tr>
</thead>
<tbody>
<tr>
<td>cant</td>
<td>128</td>
<td>1400</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>consph</td>
<td>1024</td>
<td>9701</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mac_econ_fwd500</td>
<td>500</td>
<td>2500</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mc2depi</td>
<td>64</td>
<td>1000</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>32</td>
<td>700</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pwtk</td>
<td>16</td>
<td>20</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rma10</td>
<td>4</td>
<td>20</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>scircuit</td>
<td>16</td>
<td>50</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>shipsec1</td>
<td>2</td>
<td>10</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
No indirection and padded bar. With these optimizations represent the modest cost of abstraction; i.e., using a reduc-
examples that show slightly lower performance than CUSP tuned CUSP library, targeting GPUs. Improvements over comparable and sometimes exceeds that of the manually-
the compiler-generated SpMV, the resulting performance is integrate seamlessly with the code generation extensions. For plex sequences of new and existing transformations that in-
dex arrays in loop bounds and subscripts. We demon-
sions to polyhedral code generation for supporting non-a
9. CONCLUSIONS
the systematic compiler-based derivation.
core kernel; that is, a small performance loss results from been abstracted so as to achieve a clean separation from the in our implementation the reduction implementation has coupling of reduction and the remainder of SpMV, whereas slightly improved performance in CUSP is due to the tight
the automatically-generated COO implementation. This CUSP implementation.
CUSP still achieves marginally better performance than the generated COO code achieves 92% of the corresponding
performance by an additional 10% on average as indicated by the as discussed in Section 4 further improves the overall perfor-
ture with zeroed entries (array accesses are within bounds by padding the data struc-
tion and padded", eliminates IF conditions that check if the compiler-generated version. The first optimization, "Indirec-
demonstrated the robustness of this approach in applying com-
prove significantly so that the average speedup over CUSP

Figure 8: Speedup of COO generated code with re-

Results

COO

Speedup compared to CUSP

<table>
<thead>
<tr>
<th>Test Case</th>
<th>No indirection and padded</th>
<th>Indirection and padded</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>cant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consph</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mac_econ_fwd500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mc2depi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pdb1HYS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwtk</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rma10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scircuit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shipsec1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discussion

• Average Performance Improvement of 1.14x over CUSP for CSR Scalar and Vector
  • Auto-tuning to pick optimal block size
  • Significant performance improvement for matrices with exceptionally small row lengths
  • CUSP uses a fixed block size of 256 for CSR Scalar

• COO performs within 8% of CUSP version
  • Performance was traded off for a systematic compiler based reduction implementation derivation
Summary

• Non-affine extensions
  • Support for representing non-affine bounds in iteration space
  • Generalized loop coalescing as non-affine transformation

• Updated code generation
  • Extended statement macro interface for non-affine mappings
  • Simplify multiple indirections in array accesses

• Compiler transformation recipes for high performing SpMV variants on GPU

• Compiler-generated code that performs comparably with manually tuned library, CUSP