Uninterpreted Functions: Their use in Data Transformations

Mary Hall

January 26, 2019
Acknowledgments

- Anand Venkat, Utah PhD, now at Intel
- Payal Nandy, Utah PhD student
- University of Arizona collaborators:
  - Michelle Strout, Mahdi Mohammadi
- Boise State collaborators:
  - Cathie Olschanowsky, Eddie Davis
- Intel collaborators:
  - Jongsoo Park, Hongbo Rong, Raj Barik

This work has been funded in part by DOE SciDAC and NSF projects.
Background: Data Transformations in Sparse Matrix Libraries

- Manually-tuned libraries provide architecture-aware implementations
  - Reorganize matrix representation
  - Might modify data, e.g. pad with zeros

- **Disadvantages**
  - Must be manually ported as new architectures emerge
  - Difficult to compose optimizations
Data Transformations: Motivation

Elements of compiler approach

- Polyhedral transformations
  - Composability with parallelization and other code transformations

- Uninterpreted functions
  - Represent runtime loop bounds and array indices
  - Location for transformed data

- Inspector/Executor
  - Inspector, using original loop bounds, derives transformed data representation
  - Executor, using modified loop bounds, operates on transformed data that may be padded with zeros

- Novel loop and data transformations
  - Make-dense
  - Compact-and-pad
Inspector/Executor:  
CSR to BCSR Transformation

*Specialize matrix representation for nonzero structure*

- **Compressed Sparse Row (CSR)** is a general structure that is widely used
- **Blocked Compressed Sparse Row (BCSR)**
  - Uses fixed size dense blocks if any elements are nonzero
  - Only represents blocks with at least one nonzero
  - Pads with explicit zeros if not in CSR representation; 0 computation retains meaning
  - Code for dense block is very efficient; Profitable if padding is limited

\[
\begin{bmatrix}
1 & 5 & 7 & 2 & 3 & 6 & 4
\end{bmatrix}
\]

A (in CSR): nonzeros only

\[
\begin{bmatrix}
1 & 5 & 3 & 6 & 4
7 & 2 & 0 & 4
\end{bmatrix}
\]

A (in BCSR): 2x2 blocks
Loop and Data Transformations

- **make-dense**: sparse $\rightarrow$ dense
  - Eliminate non-affine accesses
  - Introduces affine loops

- **compact** and **compact-and-pad**: dense $\rightarrow$ sparse
  - Eliminate redundancy
  - Generate inspector-executor code
  - **compact-and-pad** additionally performs a data transformation
CSR to BCSR: Transformations

**Dense Matrix**
\[
\begin{pmatrix}
1 & 5 & 0 & 0 \\
7 & 2 & 0 & 0 \\
0 & 0 & 3 & 6 \\
0 & 0 & 0 & 4 \\
\end{pmatrix}
\]

**CSR Format**
- \( A = \begin{pmatrix} 1 & 5 & 7 & 2 & 3 & 6 \end{pmatrix} \)
- \( \text{index} = \{ 0, 2, 4, 6, 7 \} \)
- \( \text{col} = \{ 0, 1, 0, 1, 2, 3, 3 \} \)

**BCSR Format**
- \( A' = \begin{pmatrix} 1 & 5 & 3 & 6 & 7 & 2 & 4 & 0 \end{pmatrix} \)
- Block-row offsets: \( \begin{pmatrix} 0 & 1 \end{pmatrix} \)
- Block-col: \( \begin{pmatrix} 0 & 1 \end{pmatrix} \)

Example with rows = cols = 4, block size = 2x2

- Make dense
- Row & col tile
- Compact & pad
**make-dense**

*make-dense*:

Expose affine loop bounds and subscripts for further transformation

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

**Example with n=4**
### Compact-and-pad

<table>
<thead>
<tr>
<th>ii</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Eliminate entirely**

- **Pad with 0**

```
<table>
<thead>
<tr>
<th>i\k</th>
<th>0</th>
<th>1</th>
<th>i\k</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>•</td>
<td>•</td>
<td>0</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>1</td>
<td>•</td>
<td>•</td>
<td>1</td>
<td>0</td>
<td>•</td>
</tr>
</tbody>
</table>
```

**ii = 0 – rows/block-size**

**kk = block-row offsets**
CSR to BCSR

Original code:

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

```c
make-dense(s0,col[j])
```

```c
for (i=0; i<n; i++)
    for (k=0; k<n; k++)
        for (j=index[i]; j<index[i+1]; j++)
            if(k==col[j])
                s0: y[i]+=a[j]*x[col[j]];
```

```c
compact-and-pad(s0, kk, A)
```

```c
tile(0,2,c,counted)
```

```c
tile(0,2,r,counted)
```

```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])
                        s0: y[ii*r+i]+=a[j]*x[kk*c+k];
```

```c
for(ii=0; ii<n/r; ii++)
    for(kk=off_index[ii]; kk<off_index[ii+1]; kk++)
        for(i=0; i<r; i++)
            for(k=0; k<c; k++)
                s0: y[ii*r + i] += A_prime[kk][i][k] * x[explicit_index[kk]*c +k];
```
Generated Inspector Code

Listing 10: Optimized inspector for BCSR SpMV.
Loop and Data Transformations

**Inspector Code is 1.5x faster than OSKI**

**Executor Code within 1% of performance of OSKI**
Loop and Data Transformations

**DIA Inspector Speedup**

- Matrices
- Average

**ELL Inspector Speedup**

- Matrices
- Inspector + Transpose
- Inspector

**Performance of Compiler generated Inspectors and Executors competitive with CUSP**

**DIA Executor Performance**

- Matrices
- CUDA-CHiLL
- CUSP

**ELL Executor Performance**

- Matrices
- CUDA-CHiLL
- CUSP
/* SpMM from LOBCG on symmetric matrix */
for( i =0; i < n ; i ++) {
  for ( j = index [ i ]; j < index [ i +1]; j ++)
    for( k =0; k < m ; k ++);
    y [ i ][ k ]+= A [ j ]* x [ col [ j]][ k ];
/* transposed computation exploiting symmetry*/
for ( j = index [ i ]; j < index [ i +1]; j ++)
  for( k =0; k < m ; k ++)
    y [ col [ j]][ k ]+= A [ j ]* x [ i ][ k ];
}

**Code A:** Multiple SpMV computations (SpMM), 7 lines of code

**Data Transformation:** Convert Matrix Format
CSR ➔ CSB
11 different block sizes/implementation

**Parallelism:** Thread-level (OpenMP w/schedule)
**Parallelism:** SIMD (AVX2)

**Other:** Indexing simplification

**Code B:** Manually-optimized SpMM from LOBCG, 2109 lines of code

**Take-away message:** Compiler-generated Code A faster than manual Code B!