Uninterpreted Functions: Their use in Data Transformations

Mary Hall

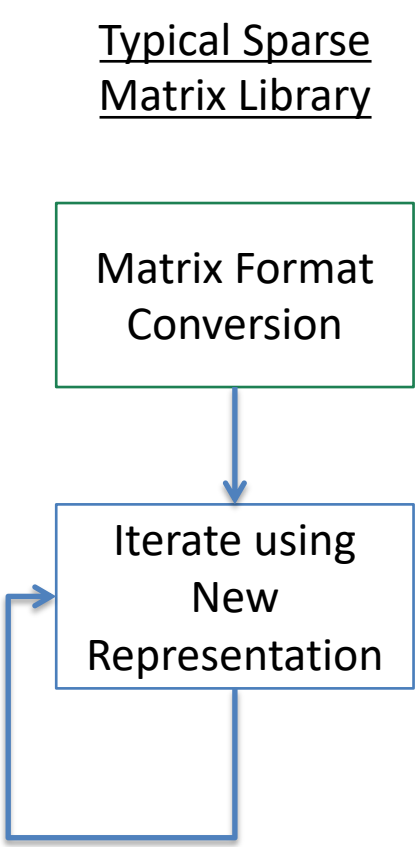
January 26, 2019

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

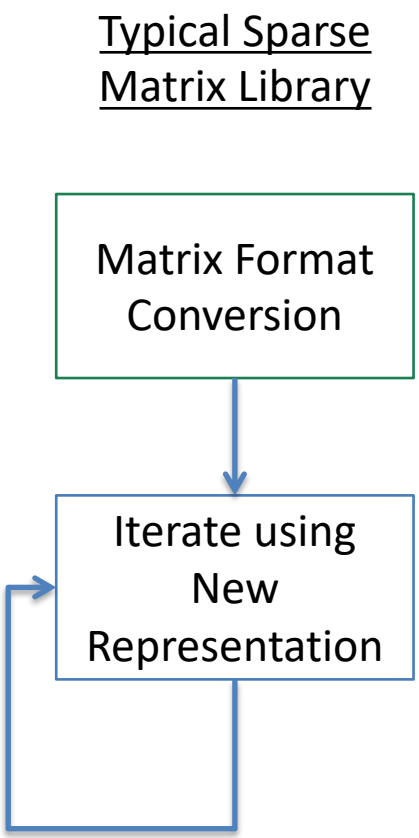
This work has been funded in part by DOE SciDAC and NSF projects.

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

A (in CSR): nonzeros only

[1 5 7 2 3 6 4]

A (in BCSR): 2x2 blocks

$$\left[\begin{array}{|c|c|}\hline 1 & 5 \\ \hline 7 & 2 \\ \hline \end{array}\ \begin{array}{|c|c|}\hline 3 & 6 \\ \hline 0 & 4 \\ \hline \end{array}\right]$$

Loop and Data Transformations

- ***make-dense*** : sparse → dense
  - Eliminate non-affine accesses
  - Introduces affine loops

- ***compact*** and ***compact-and-pad*** : dense → 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 $\begin{pmatrix} 1 & 5 & 7 & 2 & 3 & 6 & 4 \end{pmatrix}$

index $\begin{pmatrix} 0 & 2 & 4 & 6 & 7 \end{pmatrix}$

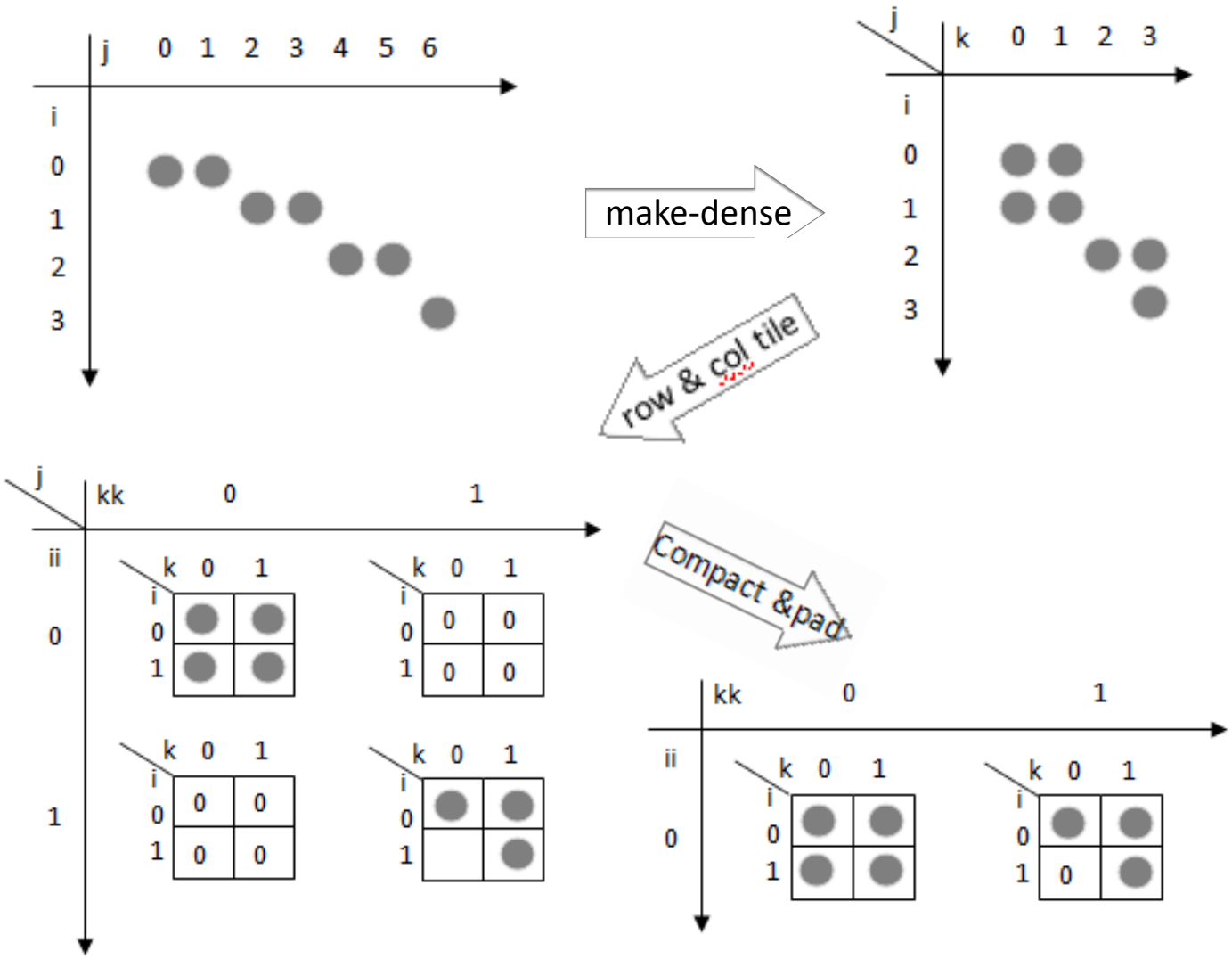
col $\begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 3 & 3 \end{pmatrix}$

BCSR Format

A' $\begin{pmatrix} \boxed{\begin{matrix} 1 & 5 \\ 7 & 2 \end{matrix}} & \boxed{\begin{matrix} 3 & 6 \\ 4 & 0 \end{matrix}} \end{pmatrix}$

Block-row offsets $\begin{pmatrix} 0 & 1 & 2 \end{pmatrix}$

Block-col $\begin{pmatrix} 0 & 1 \end{pmatrix}$



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

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]];
```

make-dense(0,j)

```
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])
    y[i]+=a[j]*x[k]
```

Affine loop bounds

Guard

Eliminate non-affine subscript

i	*j*	*col [j]*
0	0	0
	1	1
1	2	0
	3	1
2	4	2
	5	3
3	6	3

	l	*k*
•	0	0
•		1
×		2
×		3
•	1	0
•		1
×		2
×		3
×	2	0
×		1
•		2
•		3
×	3	0
×		1
×		2
•		3

Example with n=4

Compact-and-pad

0								**1**								***ii***
0				1				0				1				***kk***
0		1		0		1		0		1		0		1		***i***
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	***k***
•	•	•	•	X	X	X	X	X	X	X	X	•	•	X	•	

Eliminate entirely

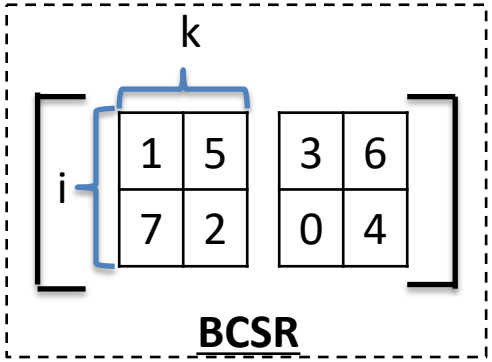
Pad with 0

i\k	0	1		i\k	0	1	
0	•	•		0	•	•	
1	•	•		1	0	•	
0				1			
0				1			

ii = 0 – rows/block-size

kk = block-row offsets

CSR to BCSR



Original code:

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

make-dense(s0,col[j])

```
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]];
```

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

```
for (ii=0; ii<n/r; i++)
  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];
```

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

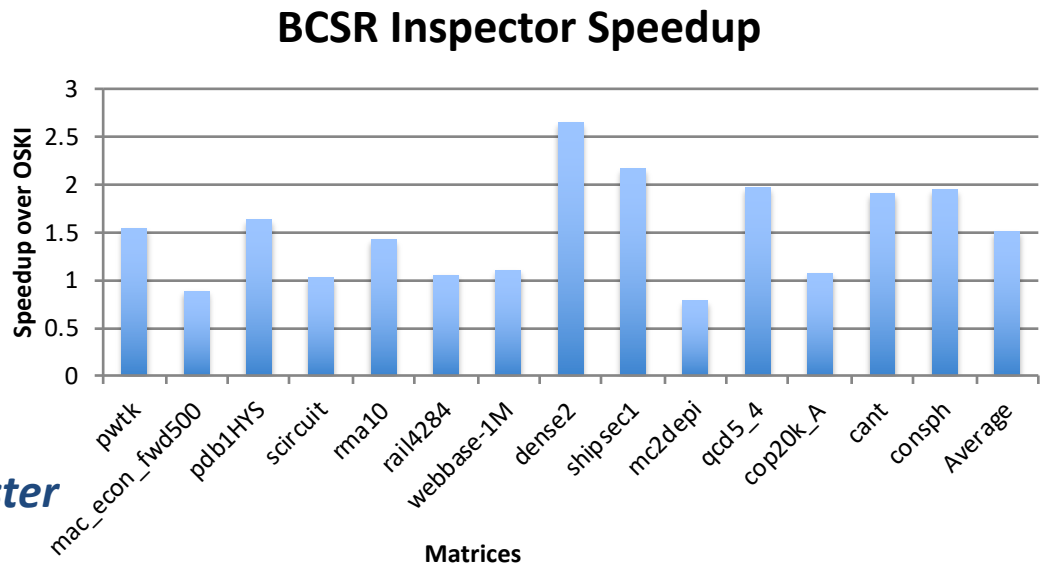
```
for(ii=0; ii < n/r; ii++)
  for(kk=off_index[ii]; kk<off_index[ii+1]; kk++)
    for(i=0; i < r; i++)
      for(k=0; k < c; k++)
        s0: y[ii*r + i] += A_prime[kk][i][k]
                  *x[explicit_index[kk]*c +k];
```

Generated Inspector Code

```
1  struct list_item{
2    float data[R][C];
3    int col;
4    struct list_item *next;
5  };
6  struct mk{
7    struct list_item *list_ptr;
8  };
9  offset_index[0] = 0;
10 count = 0;
11 struct mk marked[];
12 struct list_item *list=NULL;
13 for(ii = 0; ii < N/R; ii++){
14  for(i = 0; i < R; i++)
15   for(j = index[ii*R + i]; j < index[ii*R+i+1] ; j++){
16    kk = col[j]/C;
17    marked[kk].list_ptr = NULL;
18   }
19  for(i = 0; i < R; i++)
20   for(j = index[ii*R + i]; j < index[ii*R+i+1] ; j++){
21    kk = col[j]/C;
22    k = col[j] - kk*C;
23    if(marked[kk].list_ptr == NULL){
24     struct list_item *new_entry =
25     malloc(sizeof(struct list_item));
26     for(i_ = 0; i_ < R; i_++)
27      for(k_ = 0; k_ < C; k_++)
28       new_entry->data[i_][k_] = 0;
29     new_entry->col = kk;
30     new_entry->next = list;
31     list = new_entry;
32     marked[kk].list_ptr = new_entry;
33     count++;
34    }
35    marked[kk].list_ptr->data[i][k] = A[j];
36   }
37   offset_index[ii+1] = count;
38 }
```

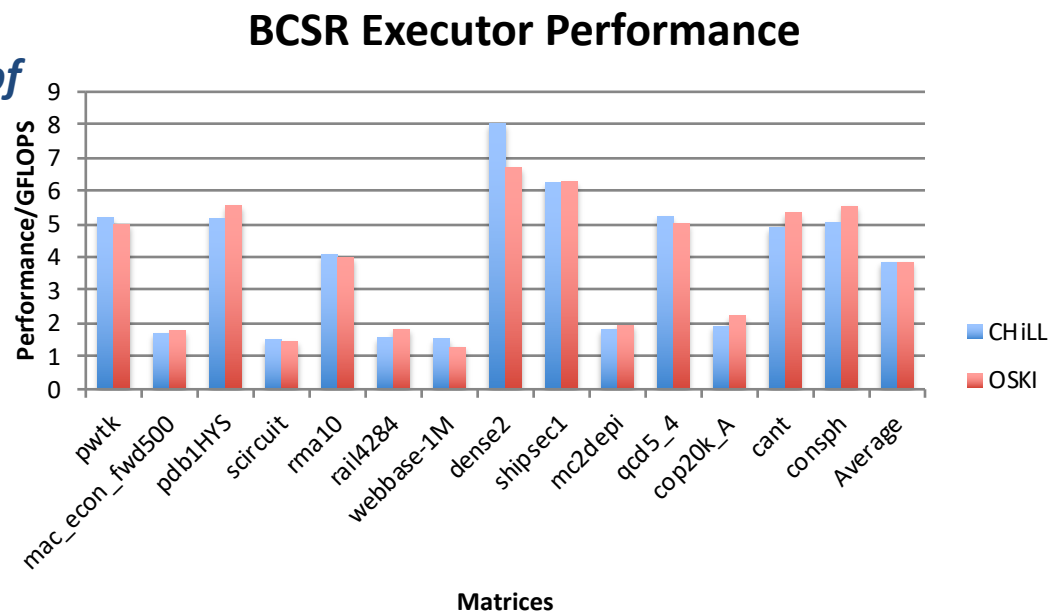
Listing 10: Optimized inspector for BCSR SpMV.

Loop and Data Transformations

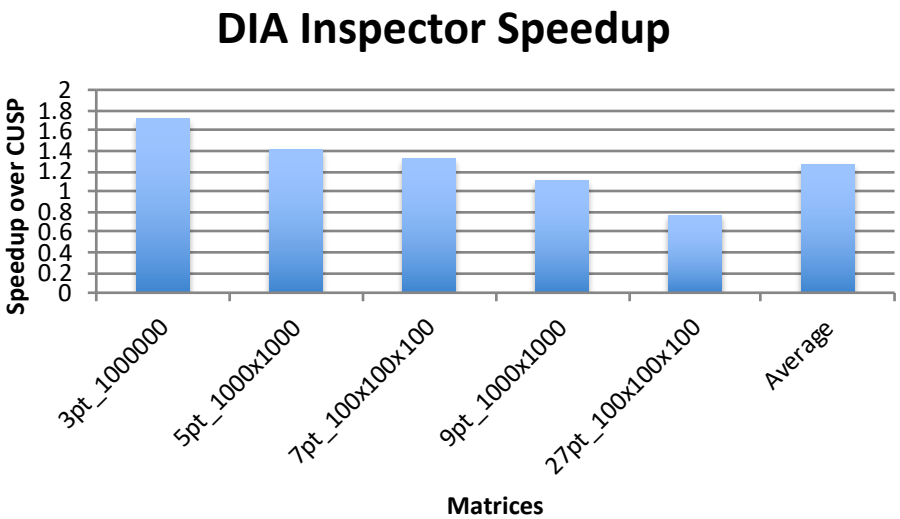


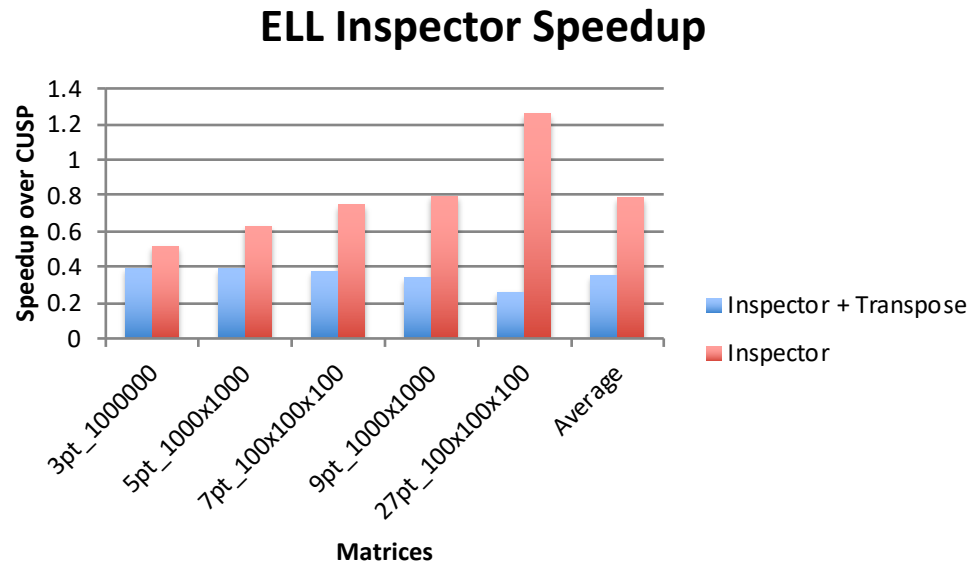
Inspector Code is 1.5x faster than OSKI

Executor Code within 1% of performance of OSKI

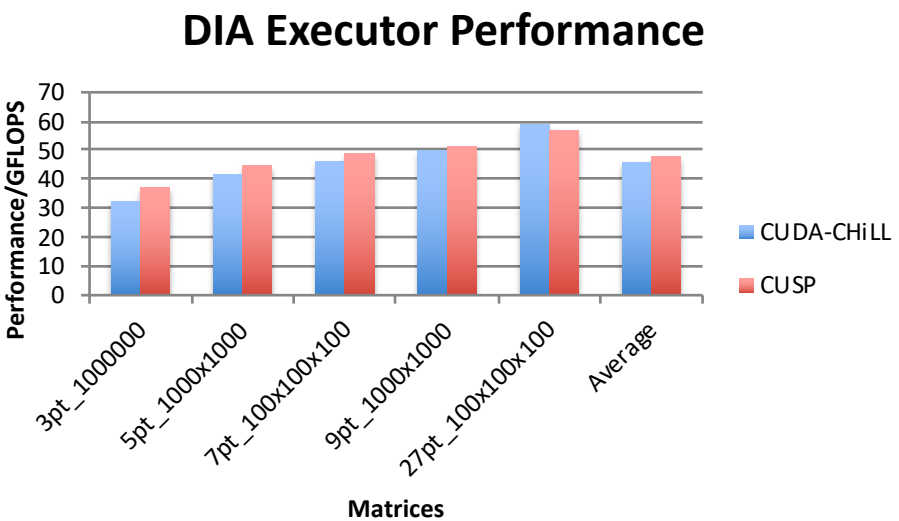


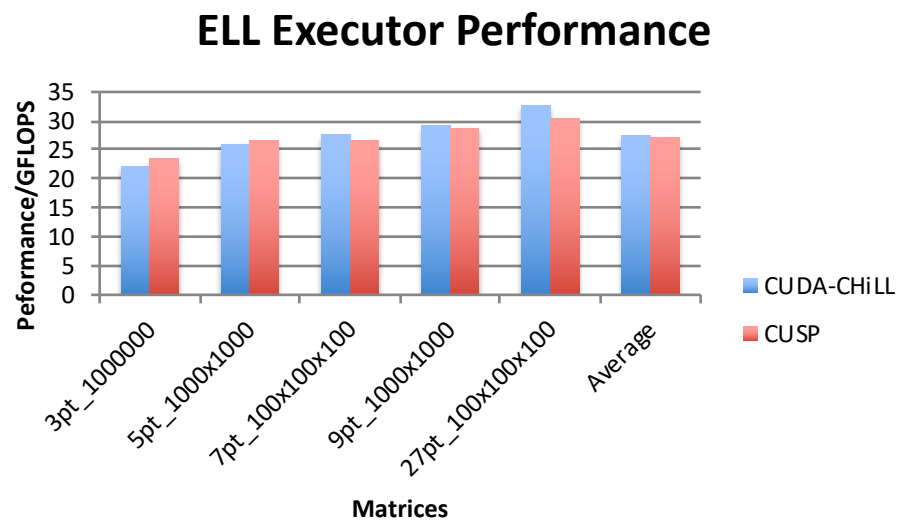
Loop and Data Transformations





Performance of Compiler generated Inspectors and Executors competitive with CUSP

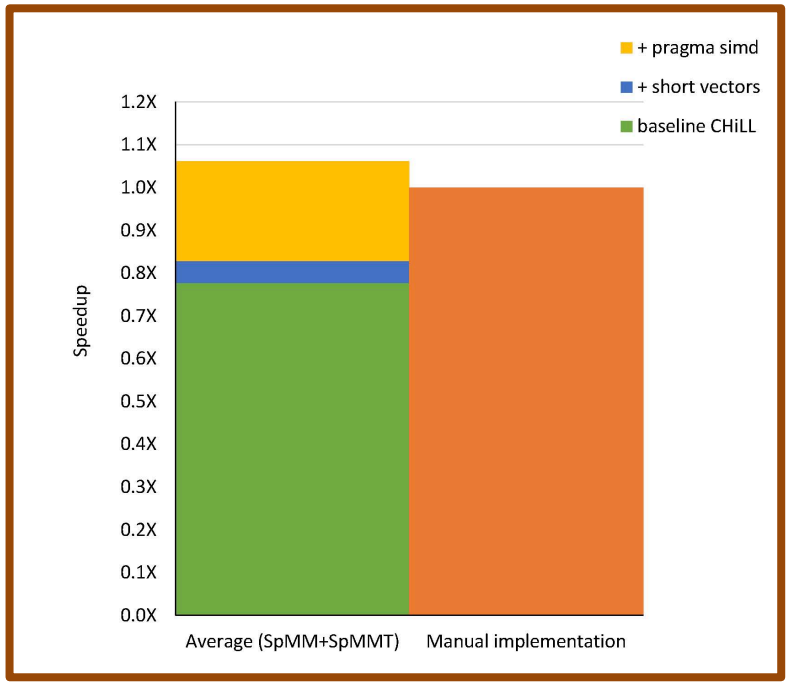


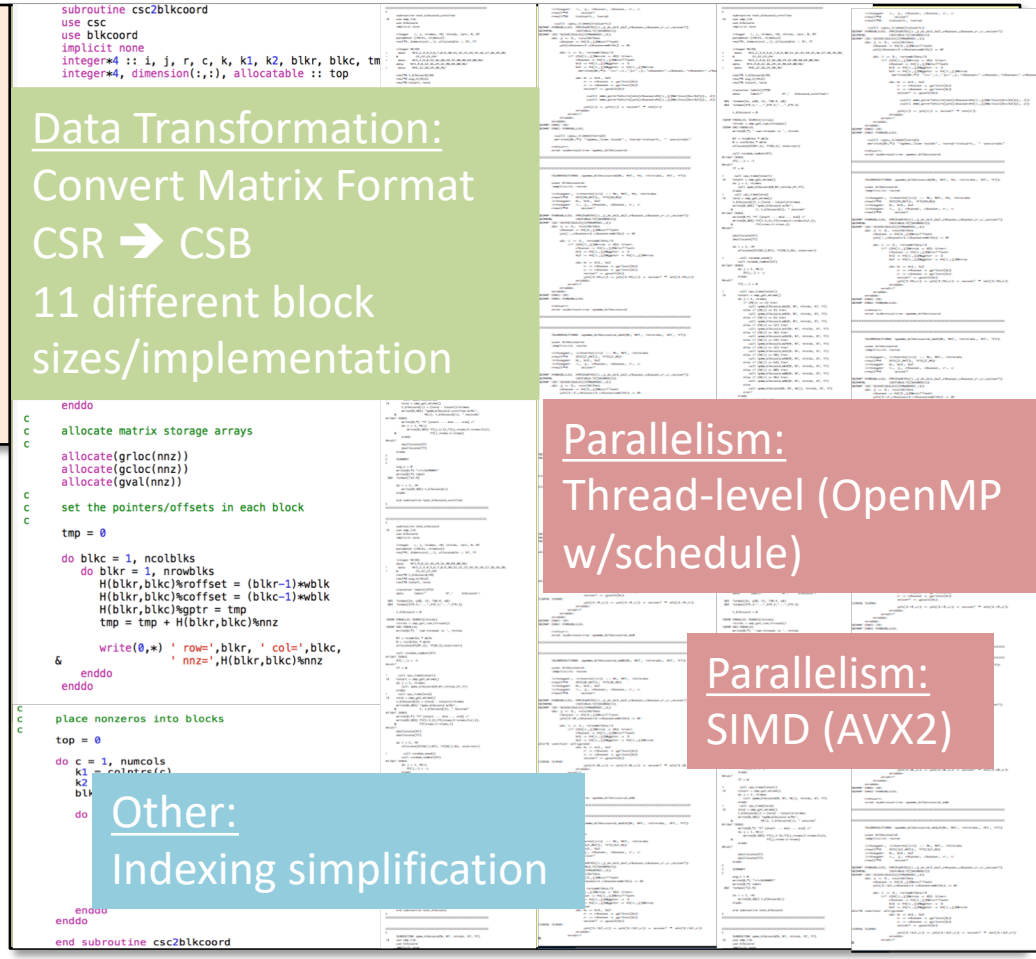


SpMM from LOBPCG (NUCLEI)

```
/* 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





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

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