Research Challenges in Compiler Technology for Sparse Tensors

Mary Hall

November 11, 2020

Acknowledgments



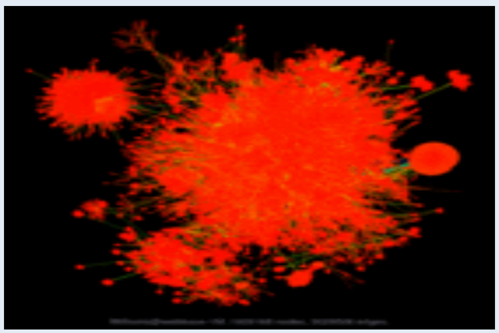

- Anand Venkat, Utah PhD, now at Intel
- Other Utah students
  - Khalid Ahmad, John Jolly, Mahesh Lakshminaranan, Payal Nandy, Tuowen Zhao
- University of Arizona collaborators:
  - Michelle Strout, Mahdi Mohammadi
- Boise State collaborators:
  - Cathie Olschanowsky, Eddie Davis, Tobi Popoola
- Intel collaborators:
  - Jongsoo Park, Hongbo Rong, Raj Barik

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the National Science Foundation under CCF- 1564074.

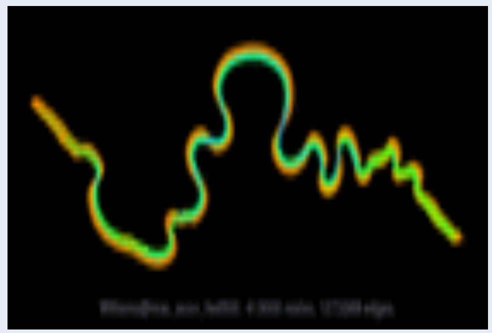
THE UNIVERSITY OF UTAH

Background

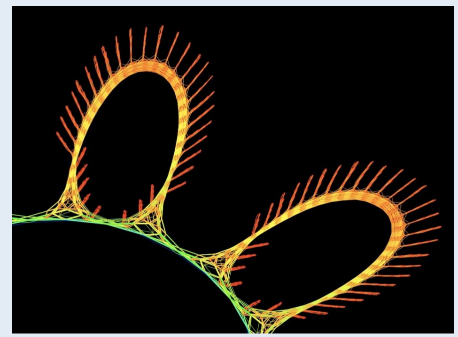
- Sparse matrices/tensors appear frequently in large systems of equations
- Sparse matrices/tensors have ***diverse*** applications
- **Density δ** often << .1



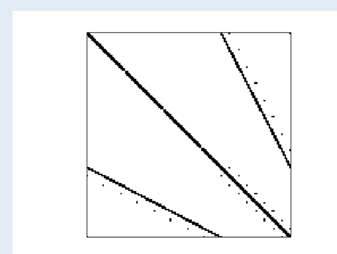
Network Theory
(Web connectivity)



Epidemiology
(2D Markov model of epidemic)



Finance
(Portfolio model)



Slide images: SuiteSparse Matrix Collection, sparse.tamu.edu

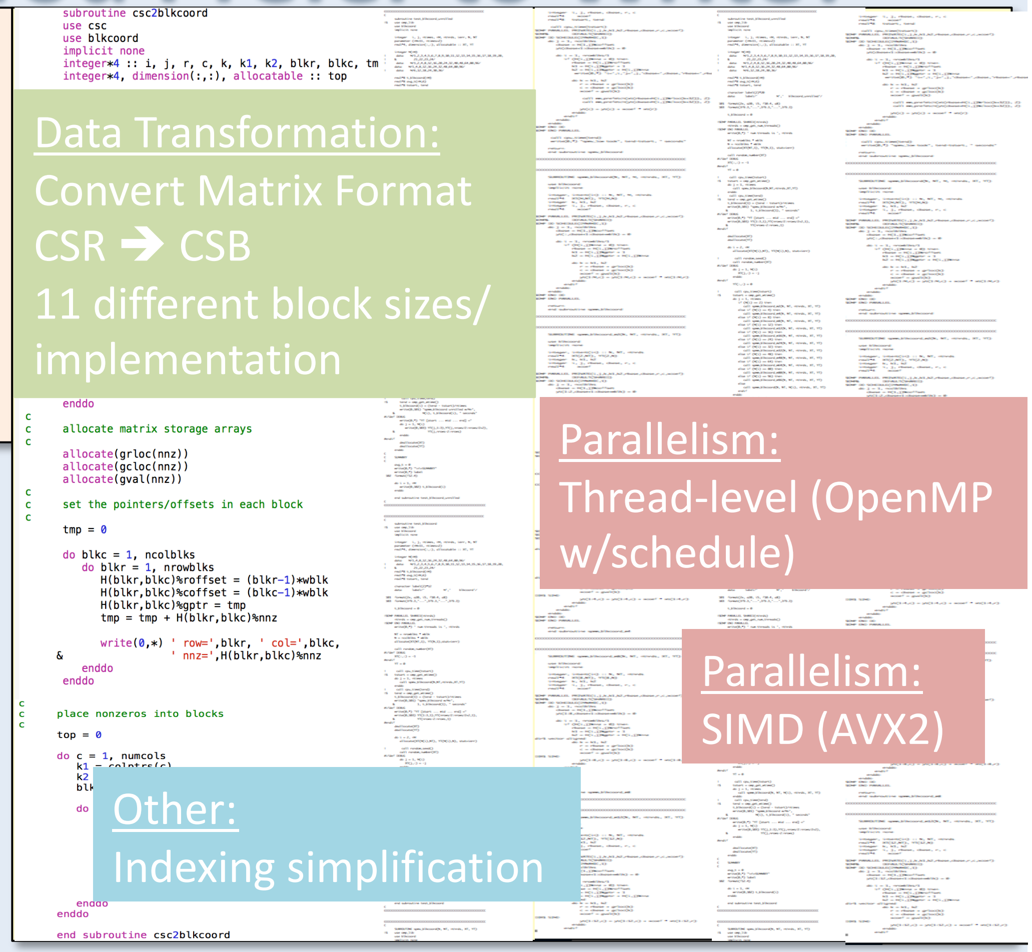
Optimizing Sparse Codes: Which Version Would You Prefer to Write?

```
/* 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

Question: Can a compiler generate **Code B** starting with **Code A**?

Answer: *YES (rest of talk)*



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

THE UNIVERSITY OF UTAH

Optimization Strategies: Compute Bound vs. Memory Bound

Optimizing Dense Linear Algebra – COMPUTE BOUND

- Exploit all forms of parallelism to approach peak flop rate
- Exploit locality of reused data in cache and registers
- Hide latency of initial cold misses

Optimizing Sparse Linear Algebra – BOUND BY DATA MOVEMENT

- Maximize memory bandwidth utilization
- Manage load imbalance
- Memory access pattern unpredictable – try to hide latency
- Select best sparse matrix representation - depends on nonzero pattern

These optimizations are usually architecture specific.

Research Challenges Work

PARALLEL SCHEDULE

- **Inspector/Executor:** Integrate runtime optimization based on input data into generated code
- **Integration:** Incorporate into the Sparse Polyhedral Framework (SPF)
- **Data dependent:** Support parallelization in the presence of data dependences

DATA REPRESENTATION

- **Format:** Convert from one to another format (e.g., CSR to BCSR)
- **Value:** Use mixed precision data values

```
/* 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 → Code B

Compiler Abstractions for Optimizing SpMV and SpMM

THE UNIVERSITY OF UTAH

Polyhedral Compiler Technology

- Mathematically represents loop nest computations and transformations applied to them
- Enables composition of transformations and correct code generation
- Abstractions representing loop nest computations
  - Iteration spaces as ***integer sets of points***
  - Transformations as ***relations on iteration spaces***
  - Statement macros as function of loop index variables
  - Underlying dependence graph to reason about safety of transformations

Polyhedral Compiler Technology for Dense Computations

Stage 1 :

Extract Loop Bounds and Construct Iteration Spaces

Input Code:
for(i=0; i < n; i++)
s0: a[i+4]=b[i+4];

Iteration Space (IS):
s0 ={[i] : 0 ≤ i < n}

Stage 2 :

Affine Loop Transformation (T)

Input IS:
{[i] : 0 <= i <= n}

T = {[i]➔[i+4]}

Output IS:
{[i] : 4 ≤ i < n + 4}

Stage 3 :

Code generation

T_inv modifies array subscripts. Then, Polyhedra Scanning

T_inv = {[i]➔[I-4]}

Output Code:
for(i=4; i < n + 4; i++)
s0: a[i]=b[i];

THE UNIVERSITY OF UTAH

Won't Work for SpMV: Non-Affine Loop Bounds and Subscripts

```
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

Non-affine subscript

col: Column for element in A
index: First location from row i in A

Uninterpreted Functions can be used to Represent Non-Affine Loop Bounds

Most Polyhedral Compilers

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

???

Can't represent bounds for loop j

Observations:

- index is invariant within loop nest
- some loop transformations may be safe if index can be represented

Uninterpreted function:

Represent index as a function in relations [Pugh and Wonnacott, TOPLAS 1998]

Extend to support

- Loop bounds
- Parameters beyond loop indices
- Transformations
- Code generation

Uninterpreted Functions Enable Transformations on Loops with Non-Affine Bounds

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

T_{tile} = {[i,j]->[i,jj,j] | exists (a | jj = 4a ∧ a ≥ 0 ∧ jj ≤ j < jj + 4)}

Represent j loop bounds as uninterpreted functions

IS = {[i,j] : 0 ≤ i < n ∧ index_(i) ≤ j < index_(i+1)}

Now tiling is possible!

```
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++)
          y[i] += (a[j] * x[col[j]]);
```

[CGO14] Venkat et al.

Inspector/Executor Transformations: Compile-Time and Runtime Collaboration

THE UNIVERSITY OF UTAH

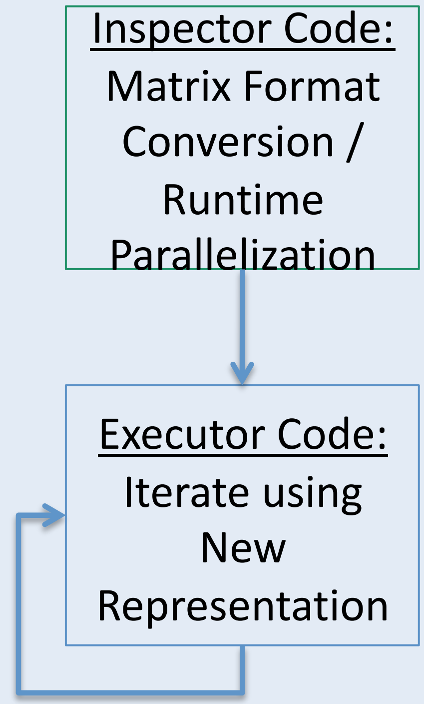
Inspector/Executor Motivation

Runtime information is needed for many optimizations to understand memory access pattern and sparse matrix nonzero structure

- ***Inspector*** analyzes indirect accesses at ***runtime*** and/or reorders data
- ***Executor*** is the reordered computation

Original concept: Mirchandaney and Saltz, ICS 1988

Both inspector and executor are generated at compile time, but inspector examines input matrix once at runtime.



Similar to sparse matrix libraries like OSKI, PETSc

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
  - 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

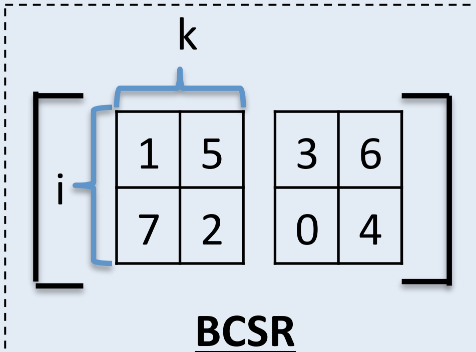
[1 5 7 2 3 6 4]

A (in BCSR):
2x2 blocks

1	5	3	6
7	2	0	4

THE UNIVERSITY OF UTAH

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];
```

[PLDI15] Venkat et al.

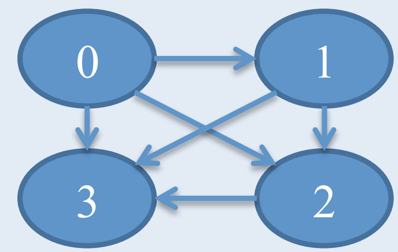
Inspector/Executor: Runtime Dependence Testing for Wavefront Parallelism

Dense Triangular Solve

- (Lower) Triangular (Forward) Solve
- Rows cannot be processed in parallel
- x[0] has to be computed before x[1]
  x[1] has to be computed before x[2]…
- Outer ***i*** loop cannot be parallelized

Dense

1	0	0	0
9	2	0	0
3	7	10	0
4	8	5	12



Dependence Graph

Sparse Triangular Solve

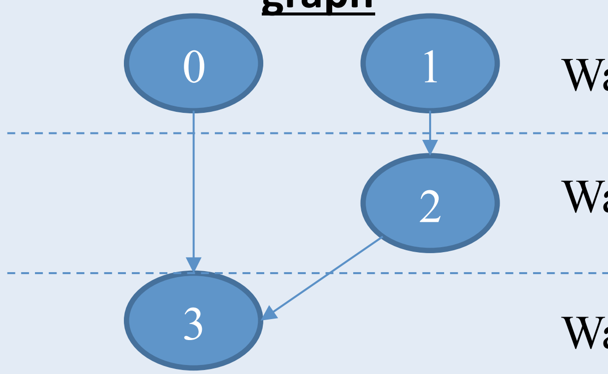
- Sparse (Lower) Triangular (Forward) Solve Kernel
- Some rows can be processed in parallel

Sparse

1	0	0	0
0	10	0	0
0	5	7	0
9	0	6	8

Dependence graph

Inspector builds dependence graph



Wavefront 0

Wavefront 1

Wavefront 2

Executor traverses wavefronts

- Parallel wavefront scheduled computation (***i*** loop partially parallel)

Parallelism is dependent on input structure

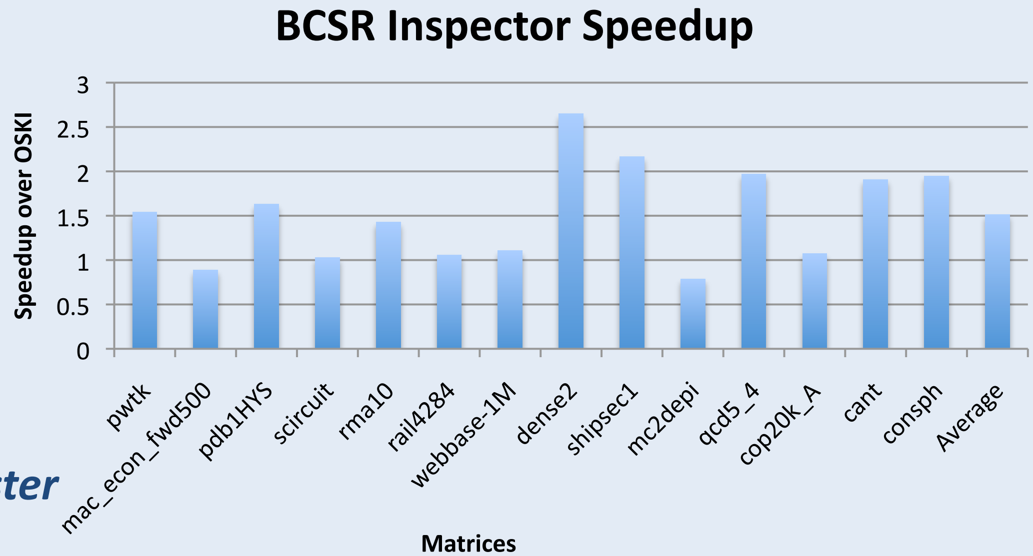
[SC16] Venkat et al. [PLDI19] Mohammadi et al.

THE UNIVERSITY OF UTAH

Performance Results Examples:

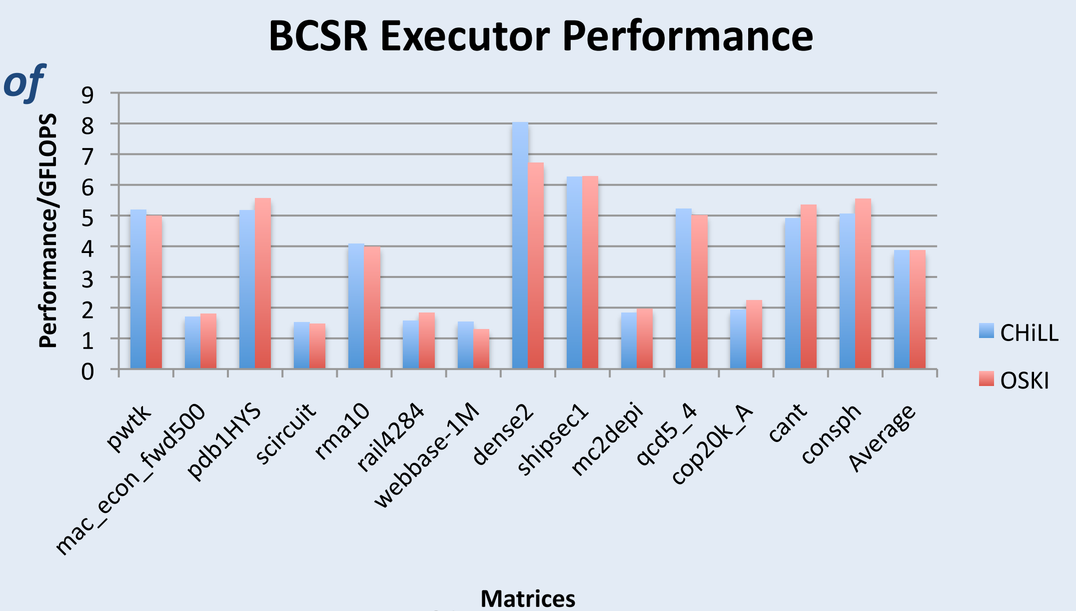
Compiler-Generated Code Performs Comparably to Manually-Written

Loop and Data Transformation

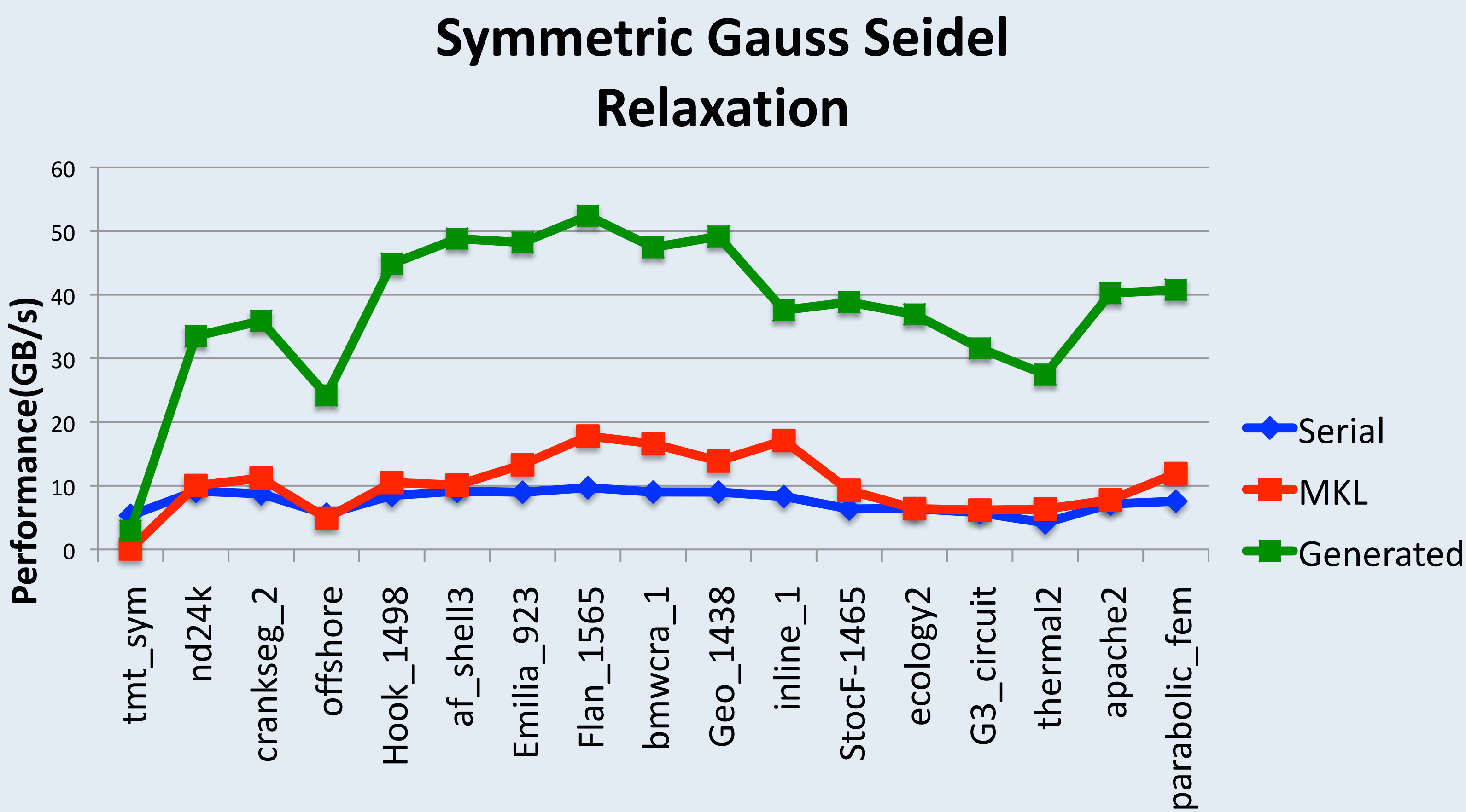


Inspector Code is 1.5x faster than OSKI

Executor Code within 1% of performance of OSKI



Wavefront Parallelization Results



THE UNIVERSITY OF UTAH

Detailed Case Study: SpMM from LOBPCG

Code A → Code B

Generated Inspector

```
for (ii = 0; ii <= 587; ii += 1)
    for (ll = 0; ll <= 589; ll += 1) {
      _P1[590 * ii + ll] = 0;
      _P_DATA1[590 * ii + ll + 1] = 0;
    }
  for (ii = 0; ii <= 587; ii += 1)
    for (i = 0; i <= 4095; i += 1)
      for (j = index_(4096 * ii + i); j <= index__(4096 * ii + i) - 1; j += 1) {
        ll = (col[j] - 0) / 4096;
        l = (col[j] - 0) % 4096;
        _P_DATA5 = ((struct a_list *)(malloc(sizeof(struct a_list ) * 1)));
        _P_DATA5 -> next = _P1[590 * ii + ll];
        _P1[590 * ii + ll] = _P_DATA5;
        _P1[590 * ii + ll] -> A = 0;
        _P1[590 * ii + ll] -> col_[0] = i;
        _P1[590 * ii + ll] -> col_[1] = l;
        chill_count_1 += 1;
        _P_DATA1[590 * ii + ll + 1] += 1;
        _P1[590 * ii + ll] -> A = A[j];
      }
  for (ii = 0; ii <= 587; ii += 1) {
    if (ii <= 0) {
      _P_DATA2 = ((unsigned short *)(malloc(sizeof(unsigned short ) * chill_count_1)));
      _P_DATA3 = ((unsigned short *)(malloc(sizeof(unsigned short ) * chill_count_1)));
      A_prime = ((float *)(malloc(sizeof(float ) * chill_count_1)));
    }
    for (ll = 0; ll <= 589; ll += 1) {
      _P_DATA5 = _P1[590 * ii + ll];
      for (newVar0 = 1 - _P_DATA1[590 * ii + ll + 1]; newVar0 <= 0; newVar0 += 1) {
        _P_DATA2[_P_DATA1[590 * ii + ll] - newVar0] = _P_DATA5 -> col_[0];
        _P_DATA3[_P_DATA1[590 * ii + ll] - newVar0] = _P_DATA5 -> col_[1];
        A_prime[(_P_DATA1[590 * ii + ll] - newVar0) * 1] = _P_DATA5 -> A;
        _P_DATA5 = _P_DATA5 -> next;
      }
      _P_DATA1[590 * ii + ll + 1] += _P_DATA1[590 * ii + ll];
    }
  }
```

(c) SpMM generated inspector code.

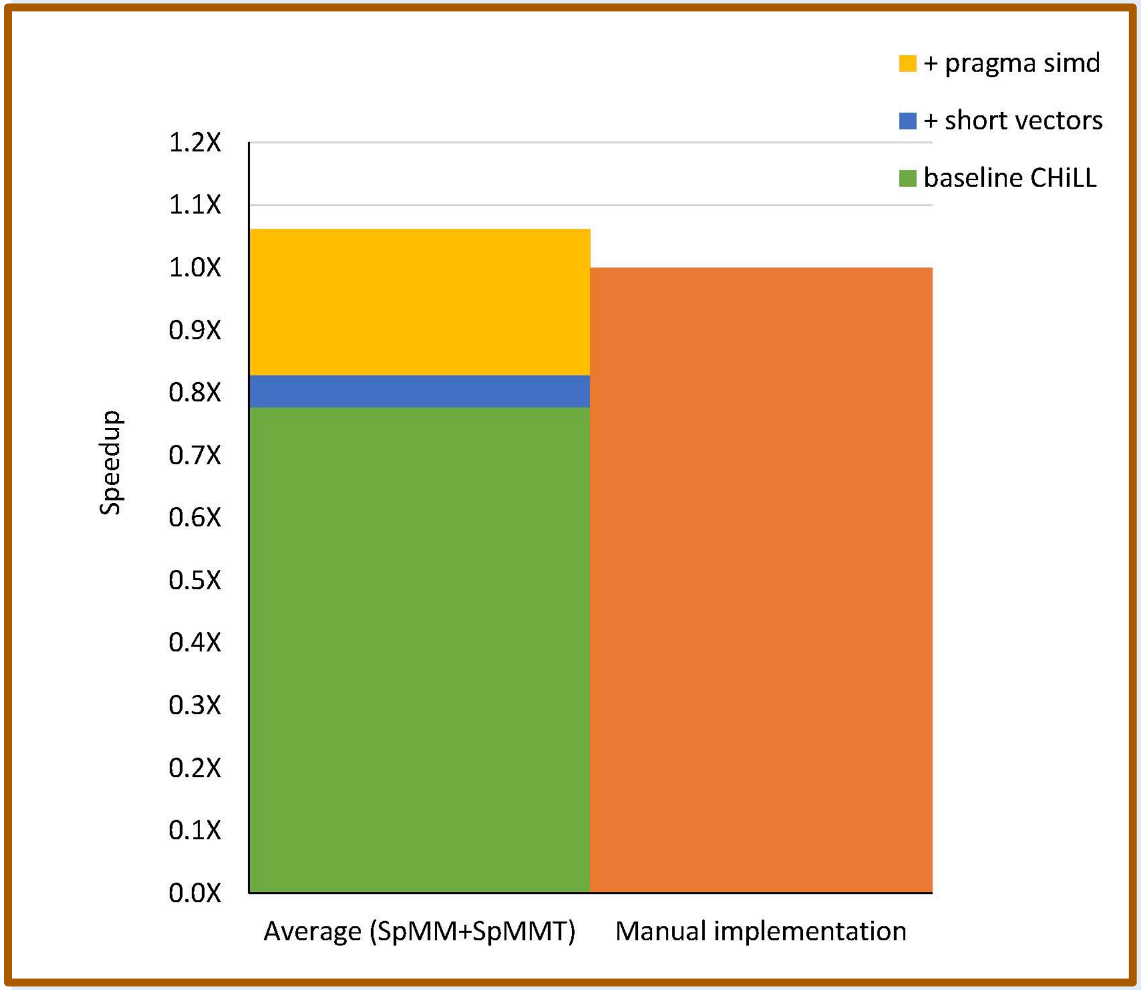
Generated Optimized Executor

```
#pragma omp parallel  private(ii,ll,i,k)
{
  #pragma omp for  schedule(dynamic,1)
  for(ii=0; ii < n/beta; ii++)
    for(ll=0; ll < n/beta; ll++)
      for(i=offset_index[ii][ll]; i < offset_index[ii][ll+1]; i++)
        #pragma simd
        for(k=0; k < m ; k++)
          y[ii*beta + expl_index_1[i]][k]+= A[i]*x[ll*beta + expl_index_2[i]][k];
}

#pragma omp parallel  private(ii,ll,i,k)
{
  #pragma omp for  schedule(dynamic,1)
  for(ll=0; ll < n/beta; ll++)
    for(ii=0; ii < n/beta; ii++)
      for(i=offset_index[ii][ll]; i < offset_index[ii][ll+1]; i++)
        #pragma simd
        for(k=0; k < m ; k++)
          y[ii*beta + expl_index_1[i]][k]+= A[i]*x[ll*beta + expl_index_2[i]][k];
}
```

SpMM Results from LOBPCG (Code A and Code B)

Intel i7-4770 (Haswell) CPU, 8 OpenMP threads



- Baseline CHiLL performance falls short of manual implementation
- Further optimization reduces data movement of index arrays (short vectors)
- **#pragma simd** for vector execution of innermost loop

Optimized Code A outperforms Code B!

THE UNIVERSITY OF UTAH

Related Work

Inspector/Executor

Mirchandaney, Saltz et al., ICS 1988
Rauchwerger, 1998
Basumallik and Eigenmann, PPoPP 2006
Ravishankar et al., SC 2012

Polyhedral Support for Indirection

Omega: Pugh and Wonnacott, TOPLAS 1998
SPF: Strout et al., LCPC 2012

Compilers for Sparse Computations

SIPR: Shpeisman and Pugh, LCPC 1998
Bernoulli: Mateev et al., ICS 2000
taco: Kholstad et al., OOPSLA 2017, PLDI 2020

Sparse Data Representations

Sublimation: Bik and Wijshoff, TPDS 1996
Ding and Kennedy, PLDI 1999
Mellor-Crummey et al., IJHPCA 2004
LL: Gilad et al., ICFP 2010
van der Spek and Wijshoff, LCPC 2010

Prior work did not integrate all of these optimizations, and mostly did not compose with other optimizations.

Research Challenges

PARALLEL SCHEDULE

- **Inspector/Executor:** Integrate runtime optimization from input data into generated code
- **Integration:** Incorporate into Sparse Polyhedral Framework (SPF)
- **Data dependent:** Parallelize w/ data dependences

DATA REPRESENTATION

- **Format:** Convert from one to another format (e.g., CSR to BCSR)
- **Value:** Use mixed precision data values

DATA LAYOUT/STORAGE

- **Physical Order:** Reorder in memory to improve reuse, reduce data movement (e.g., Morton order)
- **Data Footprint:** Reduce footprint and speed up data movement using temporaries

DEPLOY

- **Implement:** Domain-specific compiler technology in Multi-Level Intermediate Representation (MLIR) compiler, part of LLVM Foundation

Publications

[PLDI20] Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors

M. Mohammadi, T. Yuki, K. Cheshmi, E. Davis, M. Hall, M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, M. Strout

[TACO19] Data-Driven Mixed Precision Sparse Mat\rix Vector Multiplication for GPUs

K. Ahmad, H. Sundar, M. Hall, ACM TACO, Dec. 2019.

[SC16] Automating Wavefront Parallelization for Sparse Matrix Computations

Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Strout and Mary Hall (SC 2016), Best Paper Finalist.

[IA^3 16] Compiler Transformation to Generate Hybrid Sparse Computations

H. Zhang, A. Venkat, M. Hall, (IA^3 Workshop 2016).

[IPDPS16] Synchronization Trade-offs in GPU Implementations of Graph Algorithms

Rashid Kaleem, Anand Venkat, Sreepathi Pai, Mary Hall and Keshav Pingali (IPDPS 2016)

[PLDI15] Loop and Data Transformations for Sparse Matrix Code

Anand Venkat, Mary Hall and Michelle Strout (PLDI 2015)

[CGO14] Non-affine Extensions to Polyhedral Code Generation

Anand Venkat, Manu Shantharam, Michelle Strout and Mary Hall (CGO 2014)

[IMPACT16] Combining Polyhedral and AST Transformations in CHiLL

Huihui Zhang, Anand Venkat, Protonu Basu and Mary Hall (IMPACT 2016)

[LCPC16] Optimizing LOBPCG: Sparse Matrix Loop and Data Transformations in Action

K. Ahmad, A. Venkat and M. Hall, LCPC 2016.

[IMPACT18] Abstractions for Specifying Sparse Matrix Data Transformations

Payal Nandy, Mary Hall, Michelle Strout, Mahdi Mohammadi, Cathie Olschanowsky, Eddie Davis

[PIEEE18] The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code

M. Strout, Mary Hall, Cathie Olschanowsky, Proceedings of the IEEE, 2018.

THE UNIVERSITY OF UTAH