OPTIMIZING LOBPCG: SPARSE MATRIX LOOP AND DATA TRANSFORMATIONS IN ACTION

Khalid Ahmad, Anand Venkat and Mary Hall

School of Computing
University of Utah

September 29, 2016

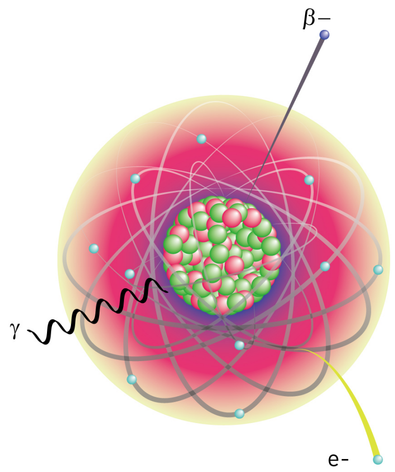
THE UNIVERSITY OF UTAH®

Motivation

- Compilers can't optimize sparse matrix computations due to the indirection in indexing and looping over the nonzero elements.

- Solution: inspector/executor methodology
  - ***Inspector*** analyzes indirect accesses at ***runtime*** and/or reorders data
  - ***Executor*** is the reordered computation

The application uses LOBPCG algorithm that studies the structure of light nuclei by solving iterative linear solvers which requires computing both SpMV and SpMVT at the same time



SpMV vs SpMM

	SpMV	K independent SpMV	SpMM
Flops	2 * NNZ	2K * NNZ	2K * NNZ
Words moved	NNZ + 2N	K * NNZ + 2K*N	NNZ + 2K *N

NNZ number of nonzero elements in the matrix.
N number of columns in the sparse matrix.
K number of dense vectors.

Take away message: SpMM is more efficient than iterative SpMV

Contributions

- We apply loop and data transformations to a real world application that uses sparse matrix computation
- Generated a parallel high performance multicore CSB SpMV/SpMM and SpMVT/SpMMT kernels
  - CSR → CSB
- Reduce data movement for indexing expressions and optimize AVX SIMD execution
- Compiler generated C code achieves speedup over the manually tuned Fortran77 code

CSR

- SpMM can be parallelized using CSR format
- SpMMT in parallel using the CSR format is difficult due to write conflicts on the output vector

11	12	13	14	0	0
0	22	23	0	0	0
0	0	33	34	35	36
0	0	0	44	45	0
0	0	0	0	0	56
0	0	0	0	0	66

data	11	12	13	14	22	23	33	34	35	36
							44	45	56	66

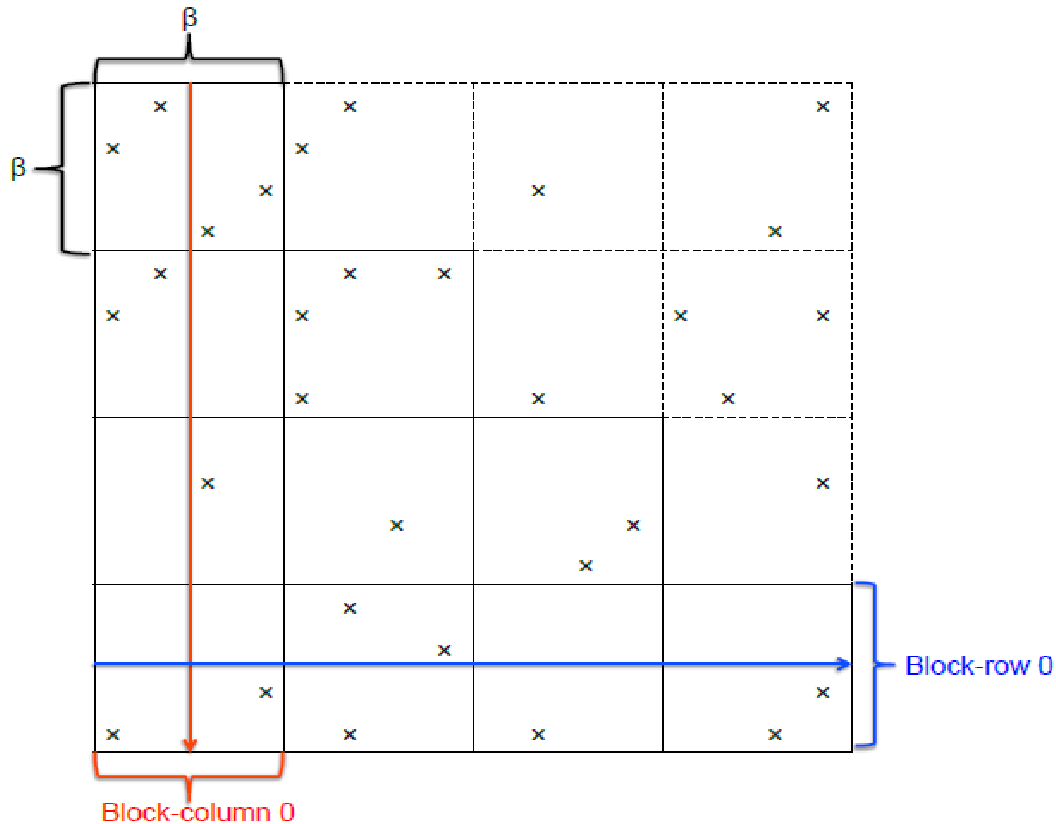
row pointer	0	4	6	10	12	13	14

column index	0	1	2	3	1	2	2	3	4	5	3	4	5	5

- CSB solves this by blocking the matrix dimensions into blocks

CSB

Compressed sparse block computes SpMV/SpMVT



11	12	13	14	0	0
0	22	23	0	0	0
0	0	33	34	35	36
0	0	0	44	45	0
0	0	0	0	0	56
0	0	0	0	0	66

data	11	12	22	13	14	23	33	34	44	35	36	45	56	66

blkptr	0,0	0,1	1,1	1,2	2,2

row index	0	0	1	0	0	1	0	0	1	0	0	1	0	1
column index	0	1	1	0	1	0	0	1	1	0	1	0	1	1

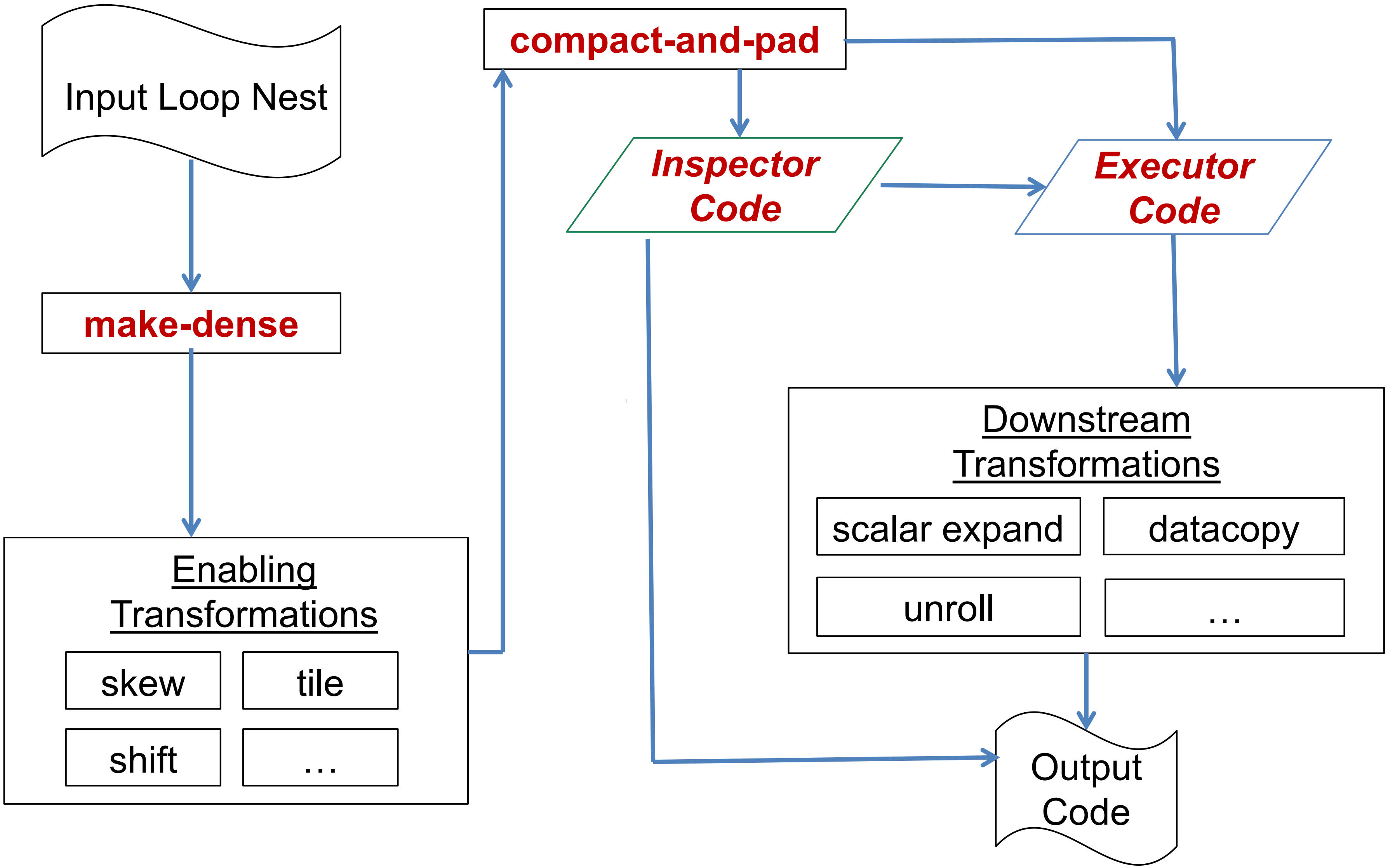
New transformations

- ***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

Venkat, Anand, Mary Hall, and Michelle Strout. "**Loop and data transformations for sparse matrix code**." *ACM SIGPLAN Notices*. Vol. 50. No. 6. ACM, 2015.

CHiLL overview



CHiLL script for SpMM based on the CSB

```
source: csb_v2.c # SpMM
procedure: csb
format : rose
loop: 0

original()
remove_dep(0,1)
fuse([0,1], 2)
split_with_alignment(0,1,4096)
split_with_alignment(1,1,4096)

make_dense(0,2,k)
known(lb == 0)
known(ub == 2412565)
known(n == 2412469)

#tile outer row and col loops by 4096
tile(0,2,4096,1,counted)
tile(0,2,4096,1,counted)

#normalize tiled loops
shift_to(0,4,0)
shift_to(0,3,0)
```

block
size

```
compact(0,[3,4],[A_prime], 0, [A])

distribute([0,1,2,3], 1)
permute(1,1,[2,1])

#OpenMP code generation
mark_omp_threads(0,[0])
mark_omp_threads(1,[0])
mark_omp_threads(2,[0])
mark_omp_threads(3,[0])

# simd code generation
mark_pragma(0,4, simd)
mark_pragma(1,4, simd)
mark_pragma(2,3, simd)
mark_pragma(3,3, simd)

#set number of OpenMP threads
omp_par_for(1,1,8)

known(index_ < index__)
known(m > 1)
```

Steps of generating the inspector

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

(a) SpMM after make-dense.

```
for(ii=0; ii < n/beta; ii++)
 for(ll=0; ll < n/beta; ll++)
  for(i=0; i < beta; i++)
   for(l=0; l < beta; l++)
    for(j=index[ii*beta + i]; j < index[ii*beta+i+1]; j++)
     for(k=0; k < m ; k++)
      if(ll*beta + l == col[j])
        y[ii*beta + i][k]+= A[j]*x[ll*beta + l][k];
```

(b) SpMM after tiling.

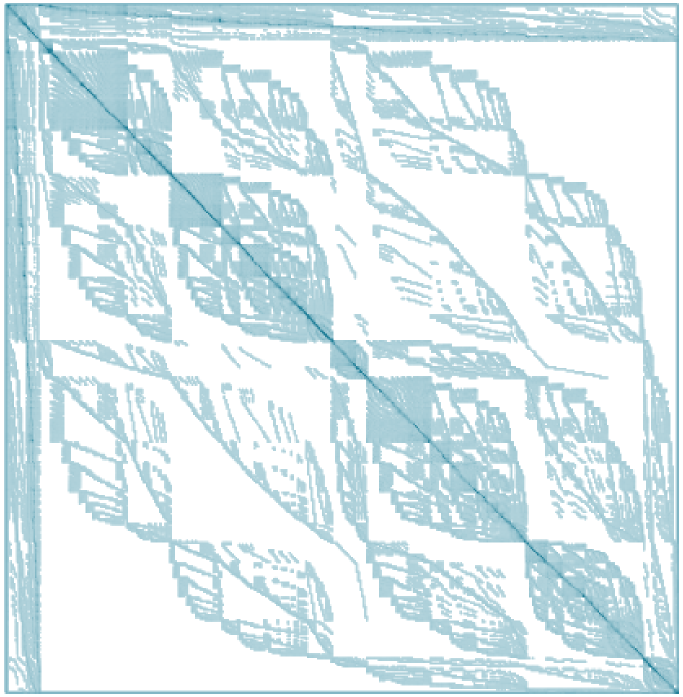
```
for (ii = 0; ii <= 587; ii += 1)
    for (ll = 0; ll <= 589; ll += 1) {
      _P1[590 * ii + ll] = 0;
      _P_DATA1[590 * ii + ll + 1] = 0;
    }
  for (ii = 0; ii <= 587; ii += 1)
    for (i = 0; i <= 4095; i += 1)
      for (j = index_(4096 * ii + i); j <= index__(4096 * ii + i) - 1; j += 1) {
        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.

Application matrix

To simplify the computation we can assume that the matrix is symmetric

To reduce the memory requirements further we can store the matrix in single precision



2,412,469 rows

2,412,566 columns

429,895,762 nonzero elements

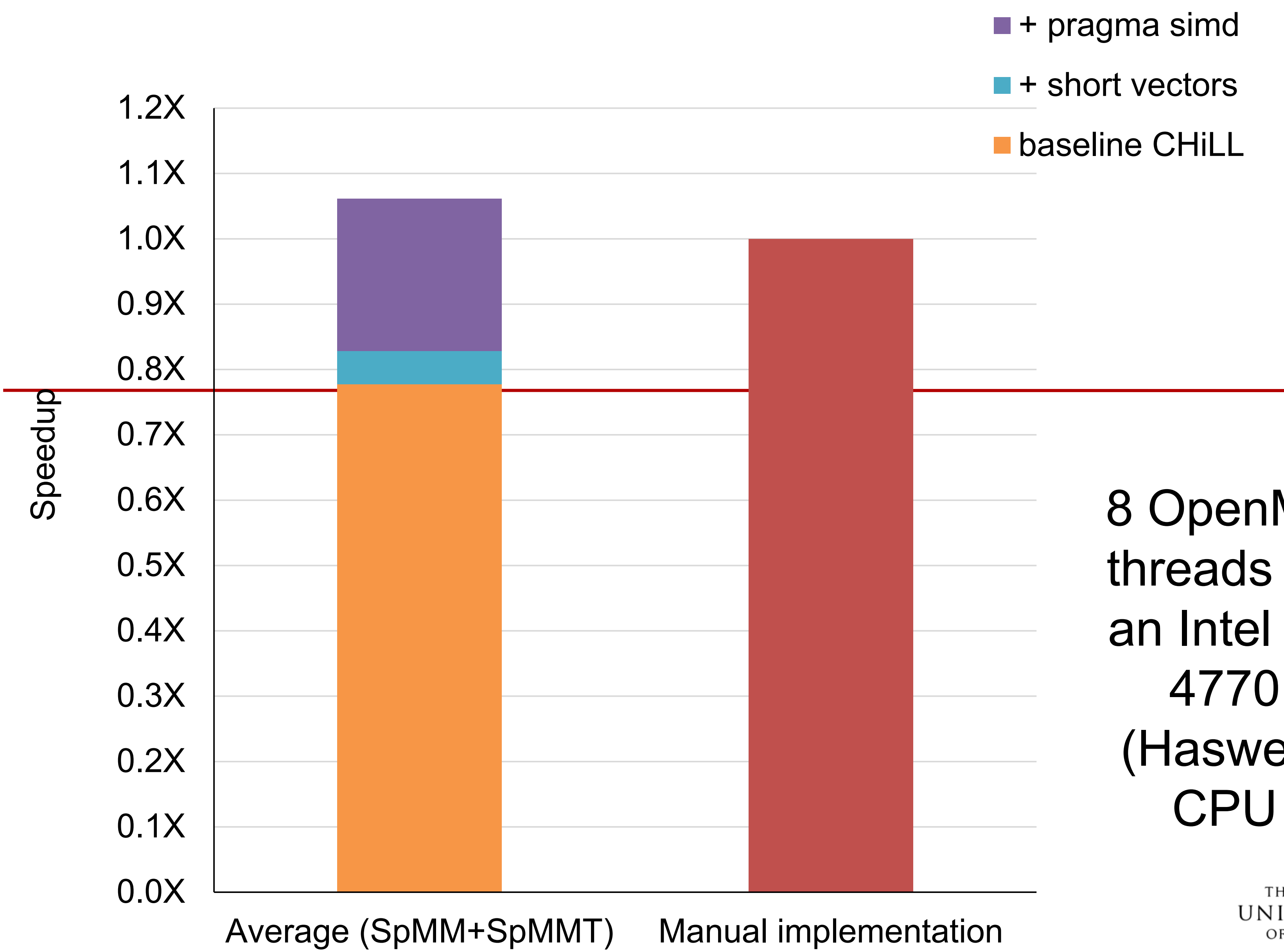
Performance measurement

- Measurements are the median of a 100 runs
- Code initialization is not included in the timings

$$GFLOPs = \frac{NNZ * 2 * NVD}{t * 10^9}$$

Where:
NNZ = number of nonzeros
NVD = number of dense vectors
t = time of execution in seconds

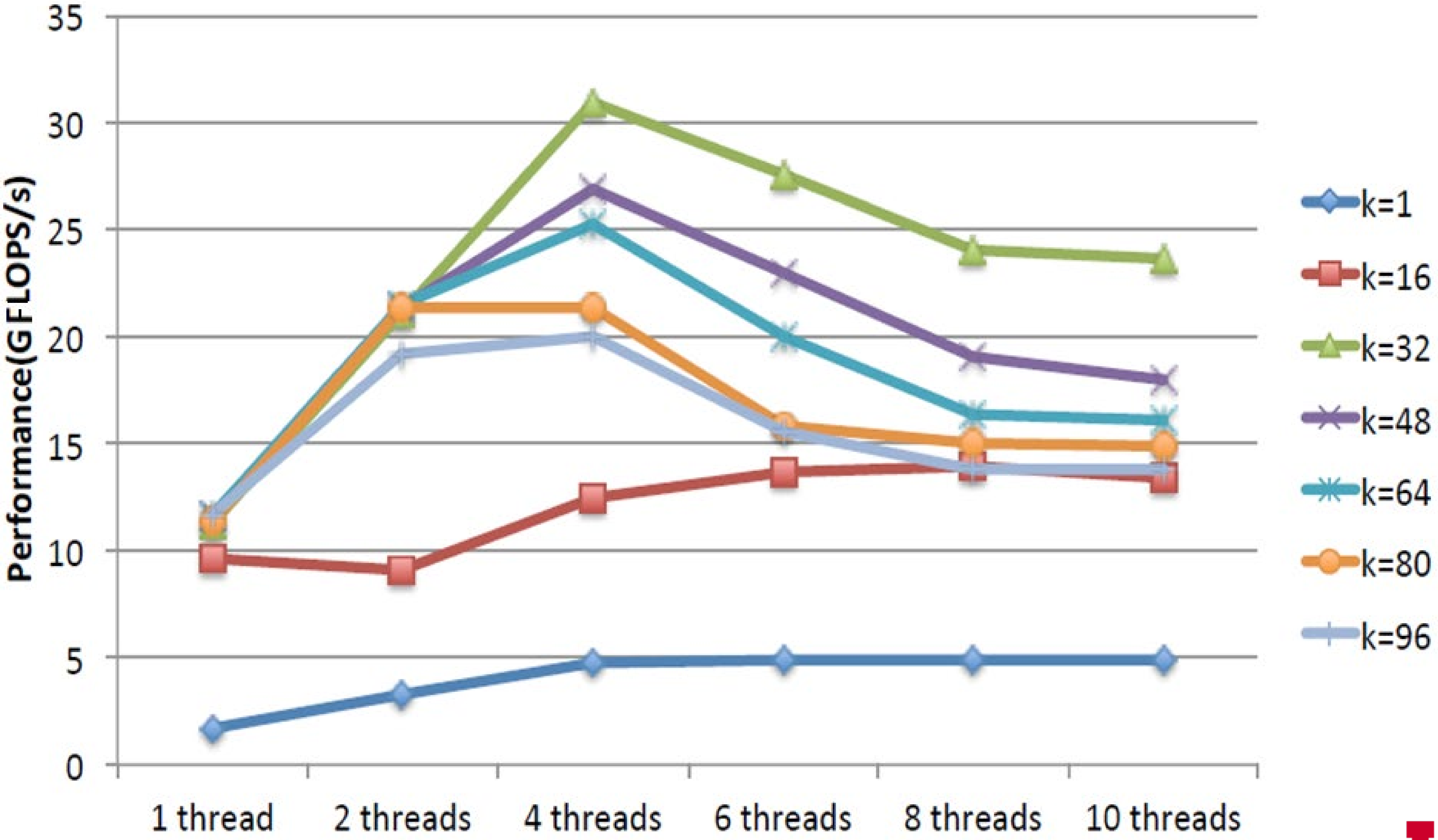
Performance comparison with the manually tuned code



8 OpenMP threads on an Intel i7-4770 (Haswell) CPU

THE UNIVERSITY OF UTAH

Multi-threaded SpMM performance



THE UNIVERSITY OF UTAH

Related Work

- **"Loop and data transformations for sparse matrix code"** (Venkat, Anand, Mary Hall, and Michelle Strout)
  - Make dense
  - Compact and pad

- “**Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations**“ (H. M. Aktulga, A. Buluc, S. Williams, and C. Yang)
  - Manually tuned Fortran77 application code

Summary & Future Work

- Generated a parallel high performance multicore CSB SpMV/SpMM SpMVT/SpMMT kernel
- Evaluated the compiler generated C code obtain from CHiLL with Fortran77 code
- Ongoing work: developing an optimal CSB SpMV/SpMM SpMVT/SpMMT kernel implementation for GPU and Xeon phi