Optimization of Block Sparse Matrix-Vector Multiplication on Shared Memory Architectures

Ryan Eberhardt, Mark Hoemmen
Sandia National Laboratories

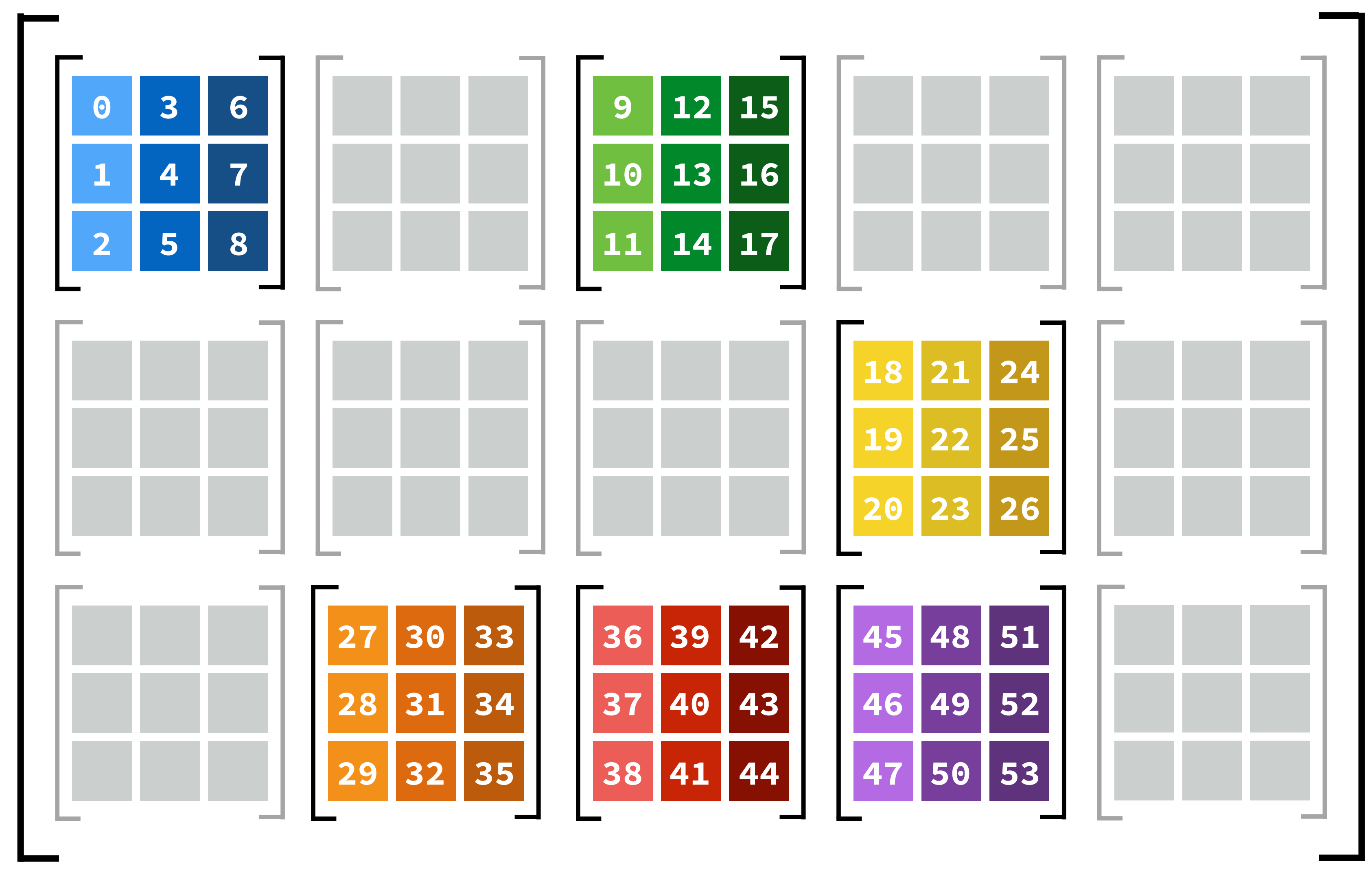
Exceptional service in the national interest

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. SAND2016-4883 C

Motivation

- Sparse matrices arising from higher-order discretizations and problems with multiple degrees of freedom often exhibit a dense block substructure in which dense blocks are constant size and aligned
- BCSR is a common storage format used to store these matrices
- Other formats may be more efficient, but there are development and runtime costs associated with translating between formats
- We seek to optimize BCSR mat-vec on various shared-memory architectures

Storage format: Block CSR




```
val = [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27...]

col_idx = [0, 2, 3, 1, 2, 3]

row_ptr = [0, 2, 3, 6]
```

Single-level parallel algorithm

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}
y_0	T0	T0	T0				T0	T0	T0						
y_1	T0	T0	T0				T0	T0	T0						
y_2	T0	T0	T0				T0	T0	T0						
y_3										T1	T1	T1			
y_4										T1	T1	T1			
y_5										T1	T1	T1			
y_6				T2	T2	T2	T2	T2	T2	T2	T2	T2			
y_7				T2	T2	T2	T2	T2	T2	T2	T2	T2			
y_8				T2	T2	T2	T2	T2	T2	T2	T2	T2			

Single-level parallel algorithm

```
#pragma omp parallel for
for(int target_block_row = 0; target_block_row < jb; target_block_row++) {
    int first_block = row_ptr(target_block_row);
    int last_block = row_ptr(target_block_row+1);

    double local_out[bs] = {0};

    for(int block = first_block; block < last_block; block++) {
        int target_block_col = col_ind(block);
        for(int col=0; col<bs; col++) {
            double vec_this_col = x[target_block_col][col];
            for(int row=0; row<bs; row++) {
                local_out[row] += val[block][col][row] * vec_this_col;
            }
        }
    }

    // Write output for this block row back to global output
    for(int row=0; row<bs; row++) {
        y[target_block_row][row] = local_out[row];
    }
}
```

GPUs

- CUDA has a large number of threads operating in SIMT (single instruction, multiple thread)
- A group of 32 threads (known as a warp) execute the same instruction in parallel
- Threads in a warp must cooperate
- Threads should also access contiguous segments of memory for “coalesced” accesses
- More levels of parallelism are needed — need to use finer-grain parallelization

GPUs: By-block algorithm

	x_0 x_1	x_2 x_3	x_4 x_5	x_6 x_7	x_8 x_9	x_{10} x_{11}	x_{12} x_{13}
y_0 y_1	T0 T2 T1 T3		T4 T6 T5 T7	T0 T2 T1 T3			T4 T6 T5 T7
y_2 y_3	T8 T10 T9 T11	T12 T14 T13 T15		T8 T10 T9 T11	T12 T14 T13 T15	T8 T10 T9 T11	
y_4 y_5		T16 T18 T17 T19			T20 T22 T21 T23	T16 T18 T17 T19	

GPUs: By-block algorithm

- Accesses to `val` will be fully coalesced. Accesses to `x` will be fully coalesced for `bs=2` and partially coalesced for larger block sizes

- Accesses to `row_ptr` and `col_idx` and writes to `global_out` are generally not coalesced, but this has a smaller impact, especially for large block sizes

- There is no interaction between threads until the final reduction. We exploit coherency of memory access within a warp to avoid synchronization

- If the block size is not a power of two, some threads in each warp will be idle. However, the high degree of parallelism keeps the memory bus saturated despite decrease in active number of threads

- Block size is limited to `bs=5`

GPUs: By-column algorithm

GPUs: By-column algorithm

- Like the previous algorithm, this algorithm has minimal interaction between threads and achieves coalesced or partially coalesced access to `val` and `x`
- However, each thread must calculate its target block index and target column (within a block) on every iteration. Memory accesses stall on these integer operations
- Despite the improved occupancy, the previous algorithm tends to outperform it for block sizes up to 5x5
- This algorithm can handle block sizes up to 32x32

GPUs: Row-per-thread algorithm

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}

y_0 T0 T0 T0 T0 T0 T0 T0 T0 T0 T0 T0 T0 T0 T0 T0 T0

y_1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1

y_2 T2 T2 T2 T2 T2 T2 T2 T2 T2 T2 T2 T2 T2 T2 T2 T2

y_3 T3 T3 T3 T3 T3 T3 T3 T3 T3 T3 T3 T3 T3 T3 T3 T3

. . .

y_8 T8 T8 T8 T8 T8 T8 T8 T8

y_9 T9 T9 T9 T9 T9 T9 T9 T9

y_{10} T10 T10 T10 T10 T10 T10 T10 T10

y_{11} T11 T11 T11 T11 T11 T11 T11 T11

. . .

GPUs: Row-per-thread algorithm

- In this implementation, accesses to **x** are not coalesced. We can address this by loading **x** into shared memory

GPUs: Row-per-thread algorithm

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 | x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}

y																
y_0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0
y_1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1
y_2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2
y_3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3
. . .				. . .								. . .				
y_8									T8	T8	T8	T8	T8	T8	T8	T8
y_9									T9	T9	T9	T9	T9	T9	T9	T9
y_{10}									T10	T10	T10	T10	T10	T10	T10	T10
y_{11}									T11	T11	T11	T11	T11	T11	T11	T11
. . .				. . .								. . .				

GPUs: Row-per-thread algorithm

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}

y																
y_0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0	T0
y_1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1
y_2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2	T2
y_3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3	T3
. . .				. . .								. . .				
y_8									T8	T8	T8	T8	T8	T8	T8	T8
y_9									T9	T9	T9	T9	T9	T9	T9	T9
y_{10}									T10	T10	T10	T10	T10	T10	T10	T10
y_{11}									T11	T11	T11	T11	T11	T11	T11	T11
. . .				. . .								. . .				

GPUs: Row-per-thread algorithm

- Achieves fully-coalesced accesses to **`val`** and **`x`** when the block size is a multiple of 16, and partially-coalesced accesses when this is not the case
- Threads use few registers, depend on little arithmetic for memory requests, and do not interact with other threads; therefore, occupancy is high and latency is low
- Performs best for block sizes ≥ 16
- Performance does not degrade as much as we might expect for `bs=17`

Experimental Results

- Compared algorithms to comparable algorithms in Intel MKL, NVIDIA cuSPARSE, and KSPARSE (Abdelfattah et al.)
- Ran each algorithm 3000 times to determine average execution time
- Divided execution time by the amount of unique data read (i.e. the total size of `x`, `val`, `col_idx`, and `row_ptr`) to determine achieved throughput. This throughput does not include time taken for population of matrix or zero-filling of output vector
- All computations were performed using double-precision values

Experimental Results

Plot	Name	bs	Dimensions (in blocks)	nnzb (nnzb/row)	Description
	GT01R	5	1.60K	20.37K (13)	2D inviscid fluid
	raefsky4	3	6.59K	111.4K (17)	Container buckling problem
	bmw7st_1	6	23.6K	229.1K (10)	Car body analysis
	pwtk	6	36.9K	289.3K (8)	Pressurized wind tunnel
	RM07R	7	545K	1.504M (28)	3D viscous turbulence
	audikw_1	3	314K	4.471M (14)	AUDI crankshaft model

Matrices from University of Florida Sparse Matrix Collection

Experimental Results: SNB/KNC

Compton testbed at Sandia National Labs

- Used two 8-core Sandy Bridge Xeon E5-2670 processors at 2.6GHz and a KNC Xeon Phi 3120A card
- Intel ICC 15.02 and MKL 11.2.2.164
- Compared against MKL's `mkl_dcsrmv` and `mkl_dbsrmv` (`mkl_dbsrmv` is not multi-threaded)
- Also tested a version of our algorithm that pads columns within blocks to multiples of 8 — each column then falls on a 64-byte boundary

	Xeon E5-2670	Xeon Phi 3120A
Clock speed	2.60 GHz	1.10 GHz
Cores	8	57
Threads	16	228
L2 cache	256KB/core	28.5 MB
L3 cache	20 MB	
Max mem bandwidth	51.2 GB/s	240 GB/s

We wish to acknowledge our appreciation for the use of the Advanced Architecture Test Bed(s), xxxx, at Sandia National Laboratories. The test beds are provided by NNSA's Advanced Simulation and Computing (ASC) program for R&D of advanced architectures for exascale computing.

Experimental Results: SNB/KNC

Performance comparison on Sandy Bridge

throughput (GB/s)

220
165
110
55
0

GT01R (bs=5)
raefsky4 (bs=3)
bmw7st_1 (bs=6)
pwtk (bs=6)
RM07R (bs=7)
audikw_1 (bs=3)

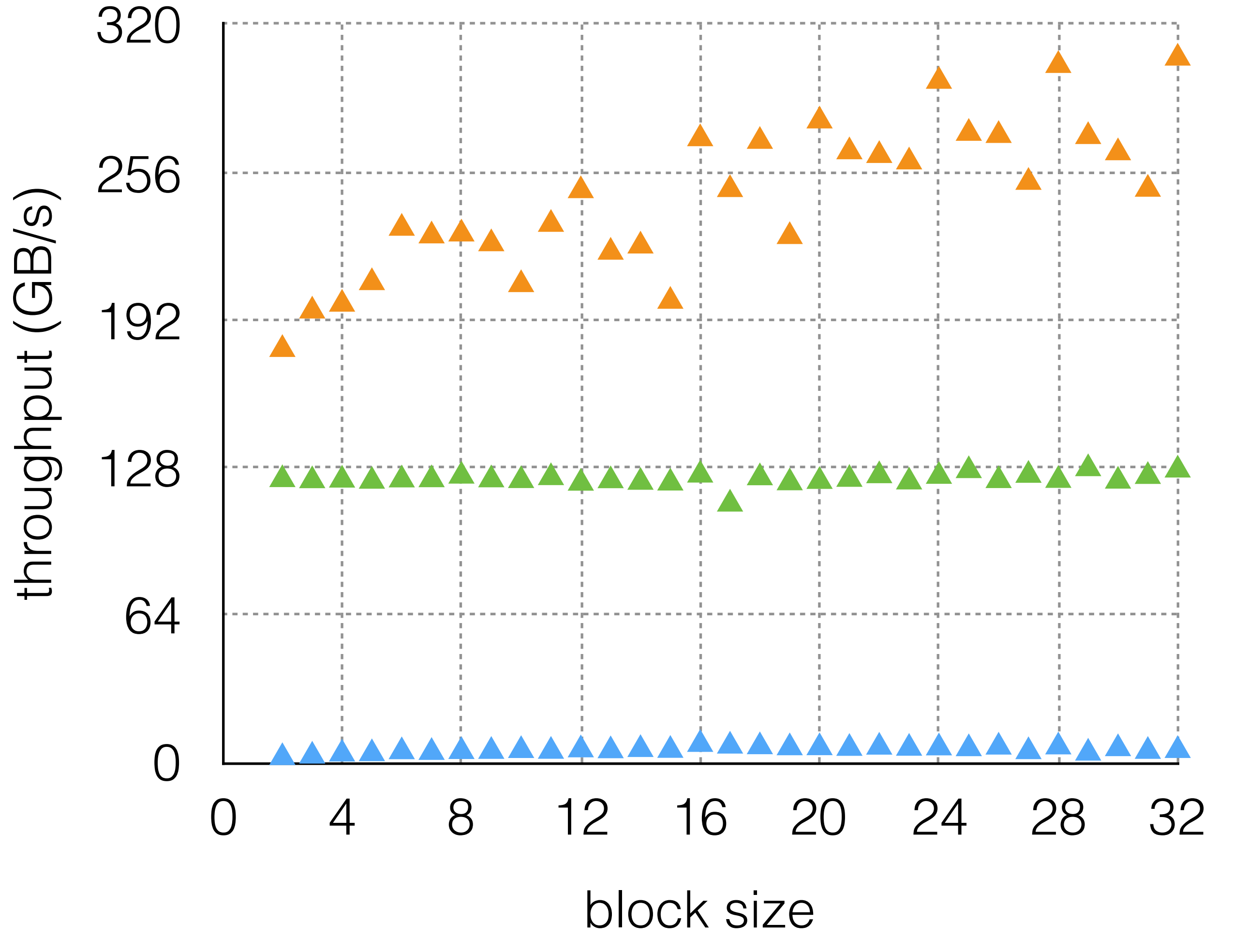
Performance comparison on Knights Corner

90
67.5
45
22.5
0

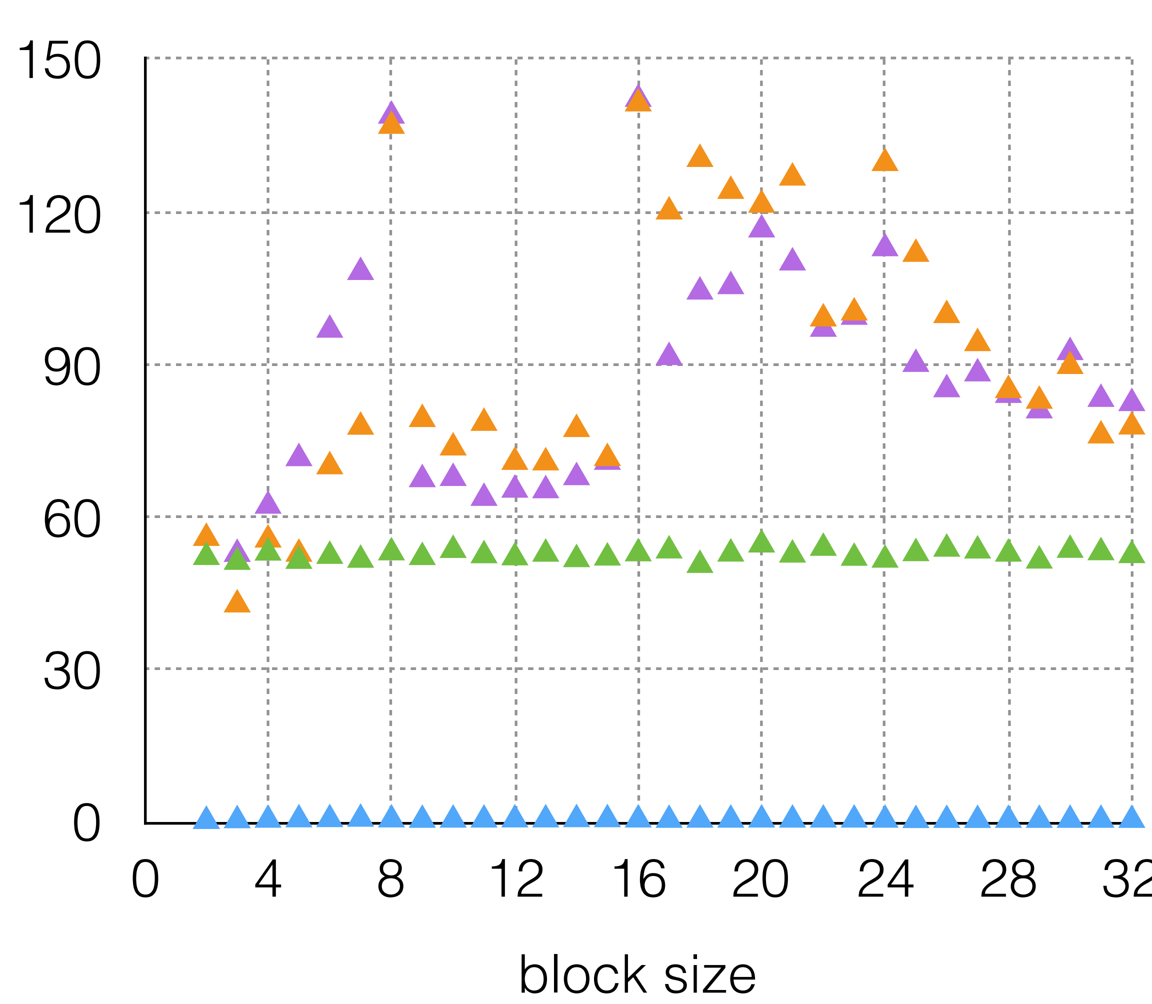
GT01R (bs=5)
raefsky4 (bs=3)
bmw7st_1 (bs=6)
pwtk (bs=6)
RM07R (bs=7)
audikw_1 (bs=3)

MKL BSRMV
MKL CSRMV
Our BSRMV
Our BSRMV (padded/aligned)

Experimental Results: SNB/KNC

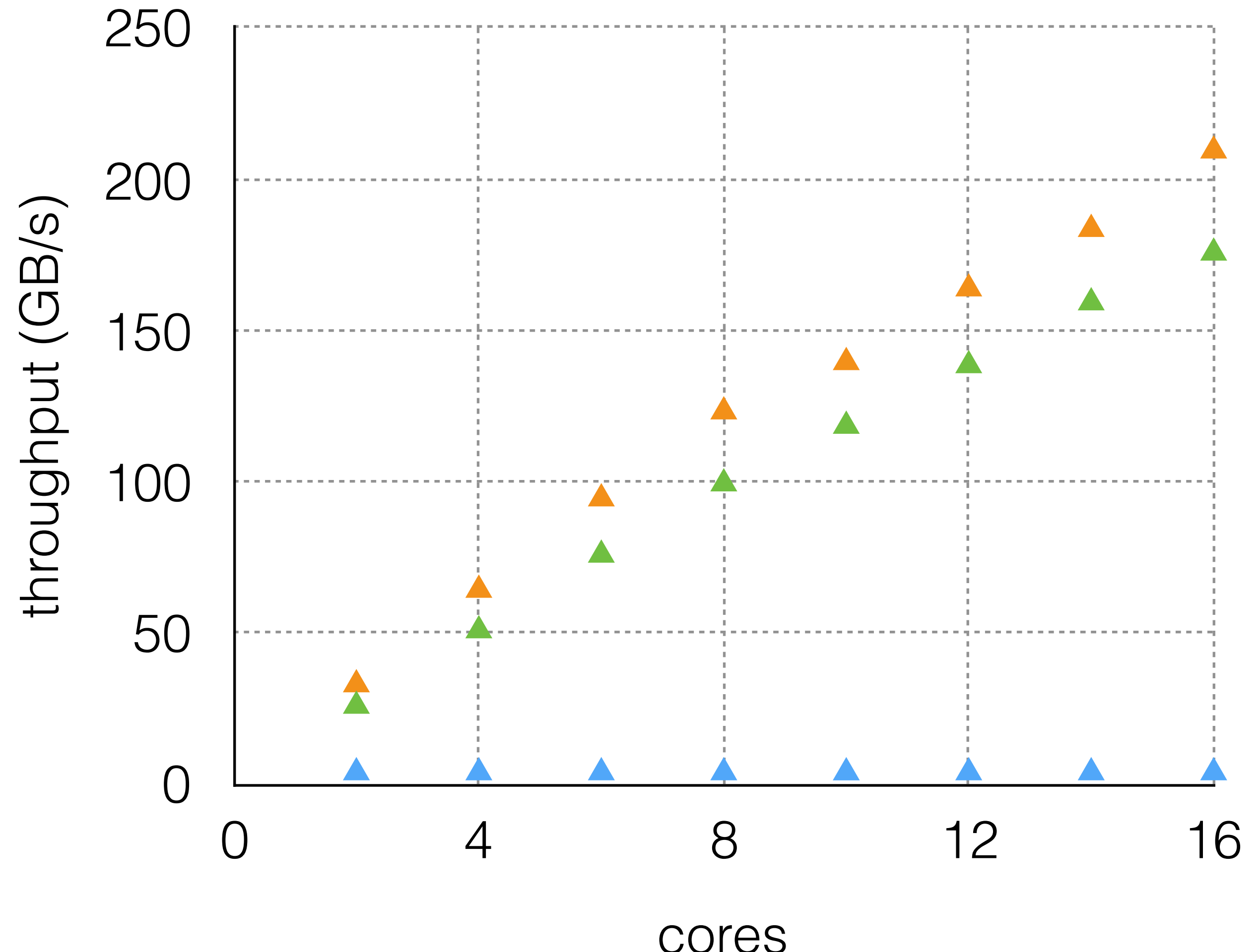


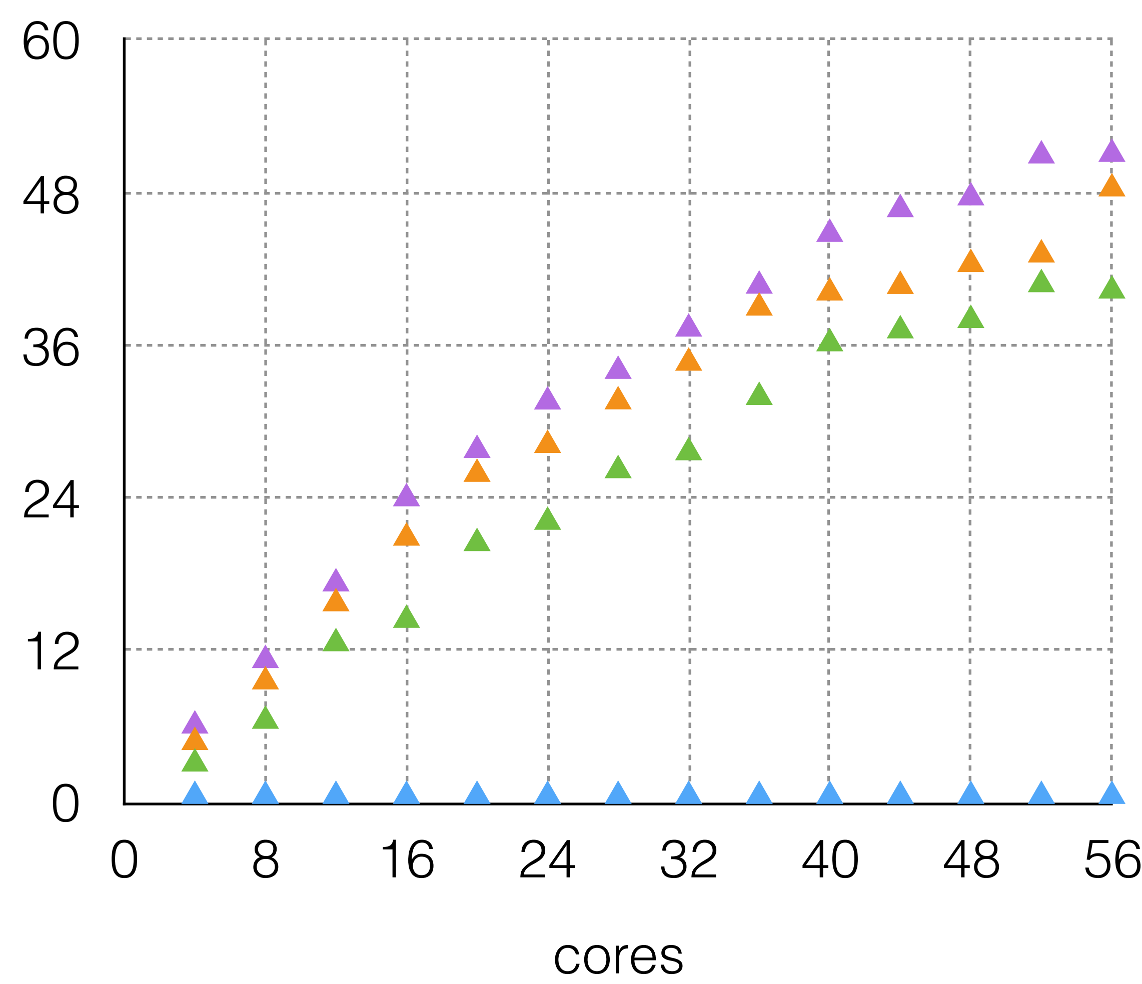
Variable block sizes on Knights Corner



- MKL BSRMV
- MKL CSRMV
- Our BSRMV
- Our BSRMV (padded/aligned)

Experimental Results: SNB/KNC





Experimental Results: Kepler

Shannon testbed at Sandia National Labs

- Used an NVIDIA K80S dual-GPU card with only one GPU used to run the tests
- ECC was on and Boost Clock was off
- GCC 4.9.0 and CUDA 7.0.18
- Compared against cuSPARSE's `cusparseDbsrmv` and KSPARSE's `ksparse_dbsrmv`

Core clock speed	562 MHz
Memory clock speed	2505 MHz
Max memory bandwidth (with ECC off)	240 GB/s
STREAM benchmarked bandwidth	145 GB/s

We wish to acknowledge our appreciation for the use of the Advanced Architecture Test Bed(s), xxxx, at Sandia National Laboratories. The test beds are provided by NNSA's Advanced Simulation and Computing (ASC) program for R&D of advanced architectures for exascale computing.

Experimental Results: Kepler

throughput (GB/s)

160
120
80
40
0

KSPARSE BSRMV
cuSPARSE BSRMV
By-block algorithm
By-column algorithm

GT01R (bs=5)
raefsky4 (bs=3)
bmw7st_1 (bs=6)
pwtk (bs=6)
RM07R (bs=7)
audikw_1 (bs=3)

Experimental Results: Kepler

throughput (GB/s)

128
96
64
32
0

0 4 8 12 16 20 24 28 32

block size

KSPARSE BSRMV
cuSPARSE BSRMV
By-block algorithm
By-column algorithm
Row-per-thread algorithm

Conclusions and Future Work

- By optimizing memory access patterns and minimizing visible latencies, we can achieve high bandwidth utilization and outperform vendor-optimized implementations
- Performance of KNC may be optimized by developing a cooperative threading strategy to improve temporal cache locality of **`x`** for hardware threads
- Data structure transformations may be required to improve performance by a significant margin. Blocks might be grouped to and processed as tiles to reduce the sizes of **`row_ptr`** and **`col_idx`** and to improve cache performance for **`x`**
  - May be possible to avoid altering the BCSR format by using metadata pointing to tiles in the matrix

Sandia
National
Laboratories

GPUs: By-block algorithm

```
const int i = thread_block_idx * thread_block_size + thread_idx;
const int warp_id = i/WARP_SIZE;
const int lane = i%WARP_SIZE;
const int target_block_row = warp_id;
const int first_block = row_ptr[target_block_row];
const int last_block = row_ptr[target_block_row+1];
const int col = (lane / bs) % bs;
const int row = lane % bs;

// Allocate shared memory for reduction step
double *shared_out = <allocate thread_block_size*sizeof(double) bytes smem>;
shared_out[thread_idx] = 0;
```

GPUs: By-block algorithm

```
// Only process whole blocks (disable threads that can only cover partial blocks):
if(lane < (WARP_SIZE/(bs*bs))*(bs*bs)) {
    double local_out = 0;
    for(int block = first_block + lane/(bs*bs); block < last_block;
            block += WARP_SIZE/(bs*bs)) {
        local_out += val[block][col][row] * x[col_ind(block)][col];
    }

    // Reduce across the warp to produce the final row results
    shared_out[thread_idx] = local_out;
    int stride = round_up_to_power_of_two((32 / bs) / 2);
    for(; stride>=1; stride /= 2) {
        if(lane < stride*bs && lane + stride*bs < WARP_SIZE)
            shared_out[thread_idx] += shared_out[thread_idx + stride*bs];
    }

    // Write the final reduced block for this row to global mem
    if(lane < bs) {
        y[target_block_row][lane] = shared_out[thread_idx];
    }

}
```

GPUs: By-column algorithm

```
const int i = thread_block_idx * thread_block_size + thread_idx;
const int warp_id = i/WARP_SIZE;
const int lane = i%WARP_SIZE;
const int target_block_row = warp_id;
const int first_block = row_ptr[target_block_row];
const int last_block = row_ptr[target_block_row+1];
const int row = lane % bs;

// Allocate shared memory for reduction step
double *shared_out = <allocate thread_block_size*sizeof(double) bytes smem>;
shared_out[thread_idx] = 0;
```

GPUs: By-column algorithm

```
// Only process whole columns (disable threads that can only cover partial columns):
if(lane < (WARP_SIZE/bs)*bs) {
    double local_out = 0;
    for(int target_nnz_col = first_block*bs + lane/bs; target_nnz_col < last_block*bs;
            target_nnz_col += WARP_SIZE/bs) {
        int block = target_nnz_col / bs;
        int col = target_nnz_col - block*bs;
        local_out += val[target_nnz_col*bs + row] * x[A.col_ind(block)][col];
    }

    // Reduce across the warp to produce the final row results
    shared_out[thread_idx] = local_out;
    int stride = round_up_to_power_of_two((32 / bs) / 2);
    for(; stride>=1; stride /= 2) {
        if(lane < stride*bs && lane + stride*bs < WARP_SIZE)
            shared_out[thread_idx] += shared_out[thread_idx + stride*bs];
    }

    // Write the final reduced block for this row to global mem
    if(lane < bs) {
        y[target_block_row][lane] = shared_out[thread_idx];
    }

}
```

GPUs: Row-per-thread algorithm

```
const int i = thread_block_idx * thread_block_size + thread_idx;
const int warp_id = i/WARP_SIZE;
const int lane = i%WARP_SIZE;
const int target_block_row = warp_id;
const int first_block = A.row_ptr(target_block_row);
const int last_block = A.row_ptr(target_block_row+1);

// Allocate shared memory for storing segments of x
double *shared_vec = <allocate thread_block_size*sizeof(double) bytes smem>;

if(lane < bs) {
    double local_out = 0;
    for(int block = first_block; block < last_block; block++) {
        shared_vec[thread_idx] = x(A.col_ind(block), lane);
        for(int col = 0; col<bs; col++) {
            local_out += shared_vec[col] * A.val[block][col][lane];
        }
    }

    y[target_block_row][lane] = local_out;
}
```