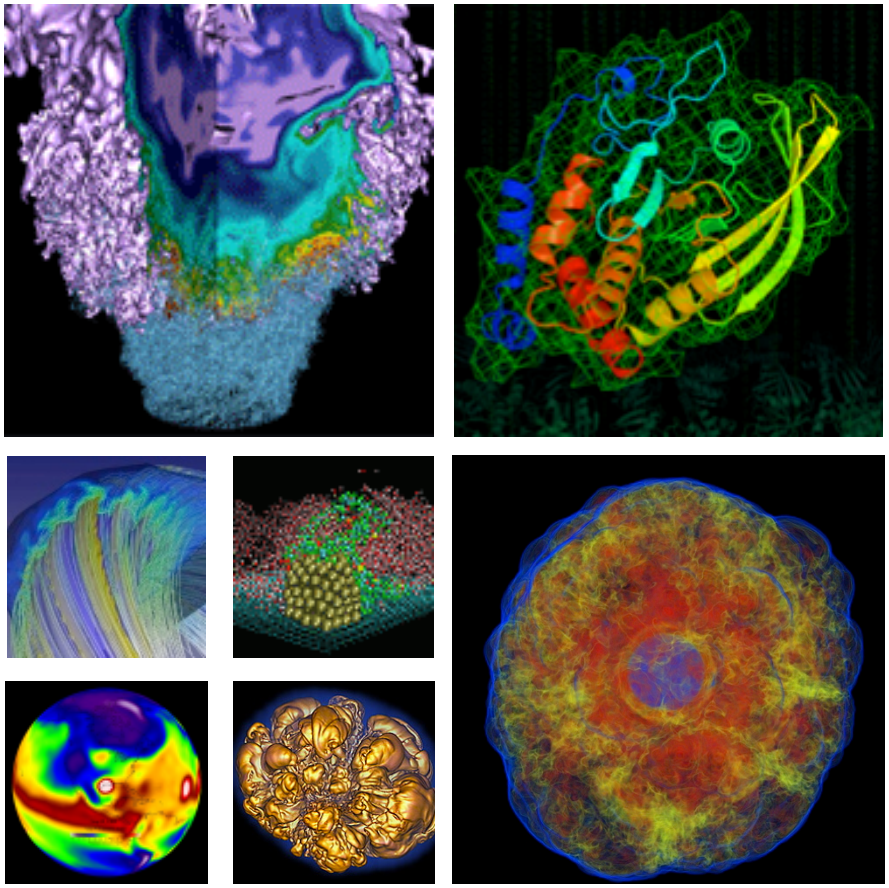
High performance optimizations for nuclear physics code MFDn on KNL



Brandon Cook, Pieter Maris, Meiyue Shao, Nathan Wichmann, Marcus Wagner, John O'Neill, Thang Phung and Gaurav Bansal

June 22, 2016

Overview

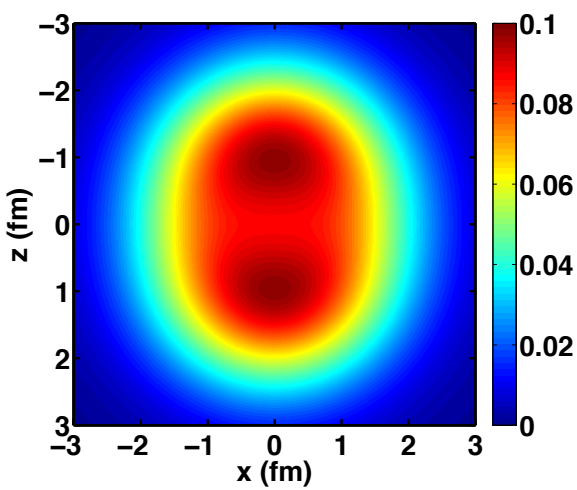
- What is MFDn?
- Systems and setup
- Matrix construction
- SPMM kernel

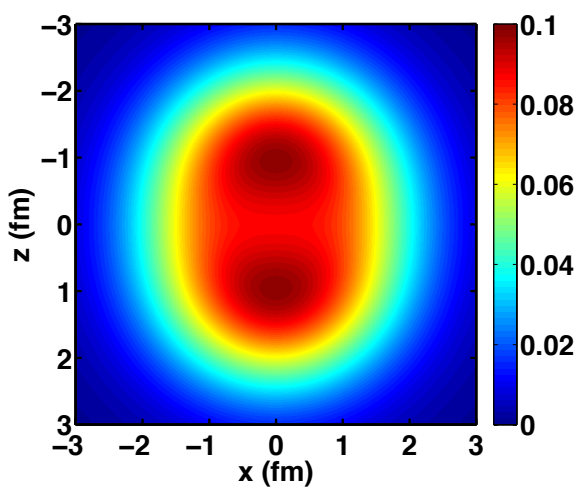
MFDn - Introduction

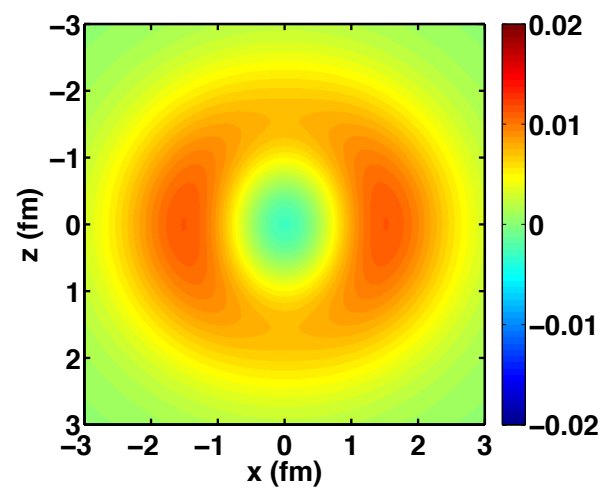
- **Many-Fermion Dynamics – nuclear (MFDn)**
- **Configuration Interaction (CI) for nuclear structure**
  - Realistic nucleon-nucleon and three-nucleon forces
- **Fortran90 code (+ very small amount of C)**
  - Platform independent
  - Hybrid MPI/OpenMP
- **Currently in use at multiple DOE centers**
  - Edison at NERSC
  - Mira at ALCF
  - Titan at ORNL

MFDn – Typical Calculation

- Generate many-body basis space
- Construct Hamiltonian matrix in basis
- Obtain lowest eigenpairs
- Calculate set of observables from eigenpairs







MFDn – Key Challenges

- **Effective use of aggregate memory**
  - Typical basis dimension = several 10^9
    - 10^{13} to 10^{14} nonzero elements (80-800 TB)
  - More memory
    - More nuclei
    - More accuracy
- **Efficient matrix construction**
- **Efficient sparse matrix-vector products**
- **Nontrivial communication patterns**
  - Two-dimensional distribution of matrix over MPI ranks

MFDn – Proxy Code

- **Runs on a single node**
- **Representative data for production**
- **All work minus communication**

- **Test case**
  - Production run designed for ~5000 nodes
    - Over 80 GB memory per node
  - 2 protons, 6 neutrons, 2-body forces
    - Full matrix: $10^{10} \times 10^{10}$
    - Local matrix: $10^{8} \times 10^{8}$
    - Local nonzeros: 8×10^{9}

Target Architecture

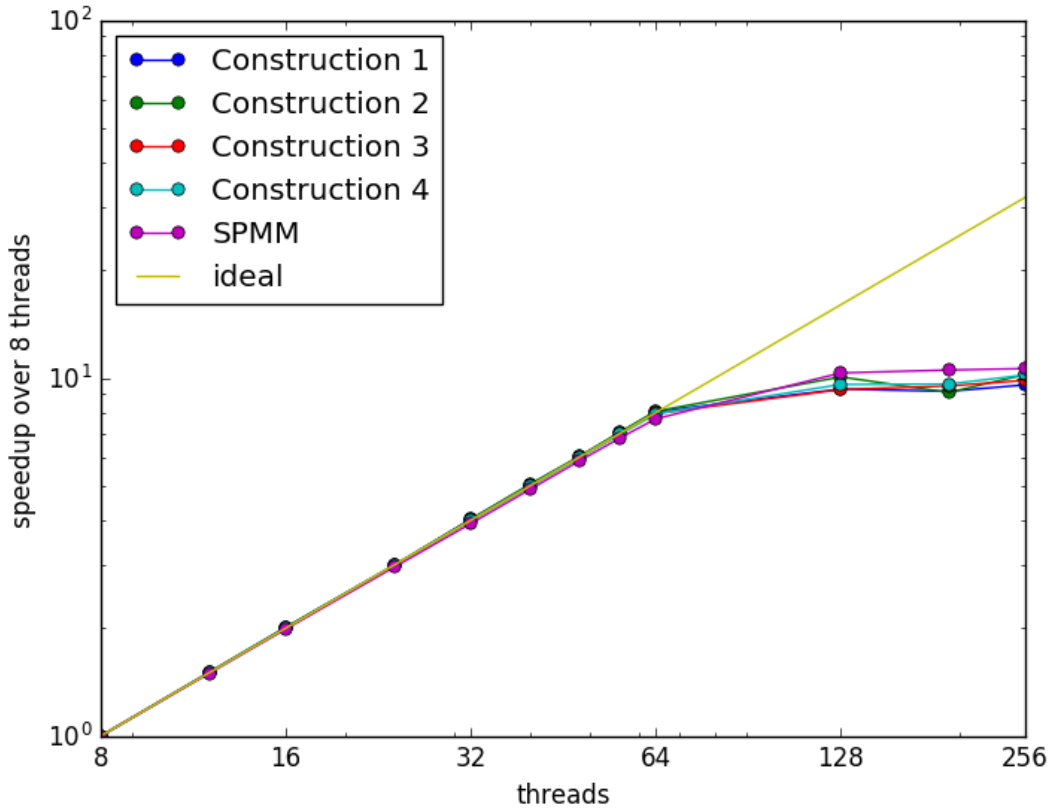
- **Knights Landing white boxes**
  - KNL preproduction, B0 stepping
  - 64 cores @ 1.3 GHz, 4 hyper-threads/core
  - 16 GB MCDRAM
  - 96 GB DDR4 @ 2133 MHz
- **Intel Haswell**
  - Cori Phase 1
  - 2x 16-cores @ 2.3 GHz
  - 128 GB DDR4 @ 2133 MHz

Measurements

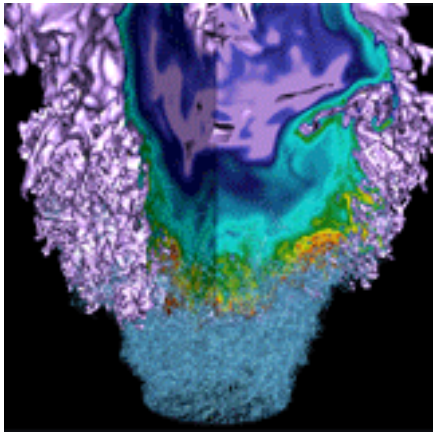
- Intel VTune for memory counters
- Intel SDE for FLOP counts
- Intel 16.0.2 compiler
- Intel MPI
- OpenMP thread placement
  - `KMP_AFFINITY=compact,granularity=fine`
  - `KMP_PLACE_THREADS=64c,4t`
- Memkind library and **`FASTMEM`** directives to allocate to MCDRAM

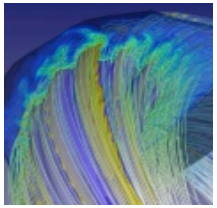
Measurements

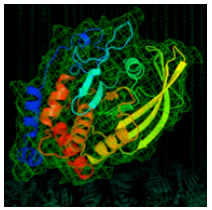
- **KNL in Quadrant+flat mode unless otherwise noted**
- **1 MPI rank per socket**
  - All parts of code have good thread scaling
- **Hyper-threads**
  - 4 on KNL
  - 1 on Haswell

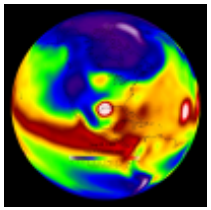


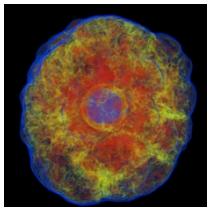
Matrix Construction

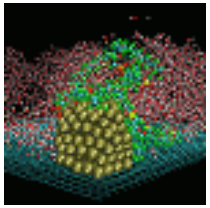












State representations

- **Many-body basis states are composed of antisymmetrized products of single particle states**
- **Many-body states can be represented in two ways**
  - BIN(Φ_i) = …01000100…000001000…
    - each bit corresponds to a single particle state which is either occupied or not
    - for n nucleons, there are n bits set to 1
  - $\Phi_i = \{s_1, s_2, \ldots s_n\}$
    - a set of n 16-bit integers indicating for each nucleon which single particle state it occupies
    - non-zero and ordered ($0 < s_i < s_{i+1}$)

Matrix construction

Matrix elements in Hamiltonian can only be nonzero if for a pair of (row, column) many-body state at most 2 nucleons are in different single particle states

1. **Count nonzero tiles in Hamiltonian**
2. **Construct nonzero tile structure**
3. **Count nonzero matrix elements in each tile**
4. **Construct Hamiltonian in CSB_Coo format**

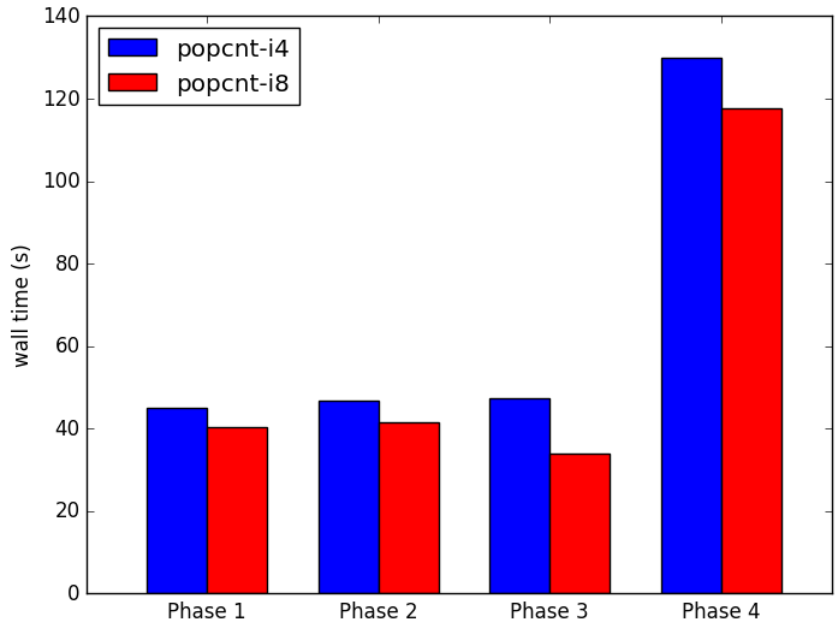
- **Steps 1-3 contain 0 flops**
  - Bit manipulations and integer comparisons
- **Step 4 has few flops, lookup tables**

Typical loop structure

```
for Φj  in column states:
   for Φi  in row states:
      if ( quick check H(i,j) != 0 ):
         careful check H(i,j) != 0
         (optional) compute H(i,j)
      end if
   end loop
end loop
```

Quick test

- **Counting number of 1's in 1st 32 bits of**
  - XOR(BIN(Φ_i) , BIN(Φ_j))
- **1st optimization: use 64 bit popcnt instruction**
  - 13% reduction across all matrix construction phases



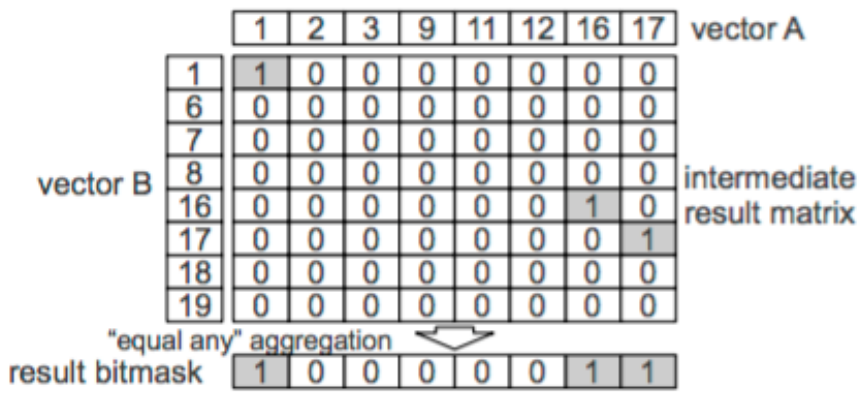
- 32bit and 64bit have nearly the same cost, but 64 filters out more zero elements
- unfortunately there are no popcnt instructions for vector registers

Hand coded popcnt

- Have only achieved parity with native popcnt instruction so far
- AVX-512BW instructions greatly simplify the implementation but are unfortunately not available on KNL
- WORK IN PROGRESS

SSE4.2 string comparison instructions

- $\Phi_i = \{s_1, s_2, \ldots s_n\}$
  - a set of 16-bit integers indicating which single particle state each nucleon occupies
  - non-zero and ordered ($0 < s_i < s_{i+1}$)
- **SSE4.2 PCMPISTRM instruction can compare two xmm registers**



Schlegel et al. ADMS 2011

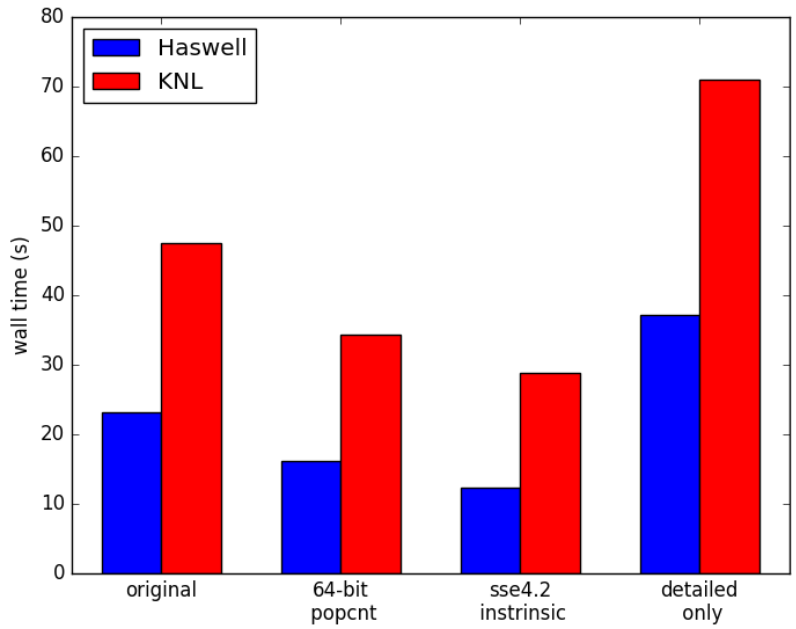
SSE4.2 string comparison instructions

- For 8 or less occupied single particle states
- In practice we unroll by 4 to reduce L1/L2 traffic

```
#include "nmmintrin.h"
int countnnz(short *A, short *B, int ncolstates, int nrowstates, int maxdiffs) {
  const int mode = _SIDD_SWORD_OPS | _SIDD_CMP_EQUAL_ANY |
    _SIDD_BIT_MASK | _SIDD_MASKED_NEGATIVE_POLARITY;
  int i,j,ndiffs,count=0;
  for (i=0; i<8*ncolstates; i+=8) {
    __m128i v_a = _mm_load_si128((__m128i*) &A[i]);
    for (j=0; j<8*nrowstates; j+=8) {
      __m128i v_b = _mm_load_si128((__m128i*) &B[j]);
      __m128i res_v = _mm_cmpistrm(v_b,  v_a,  mode);
      ndiffs = _mm_popcnt_u32(_mm_extract_epi32(res_v, 0));
      if (ndiffs <= maxdiffs) {
        count++;
      }
    }
  }
  return count;
}
```

SSE4.2 string comparison instructions

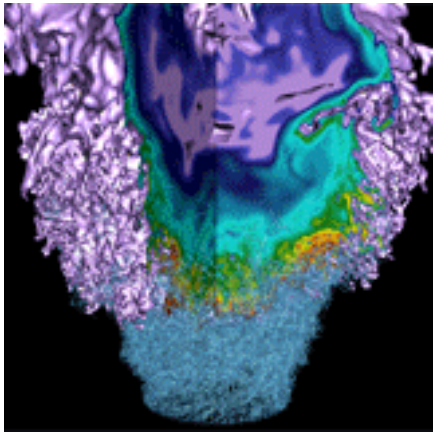
speedup	vs original	vs 64-bit popcnt
Haswell	1.88x	1.2x
KNL	1.65x	1.3x

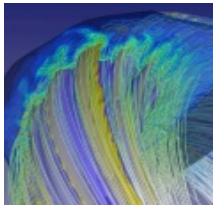


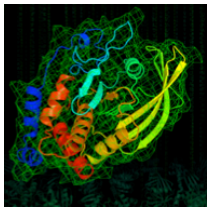
MFDn Matrix construction - Summary

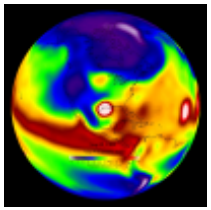
- **Lack of 16-bit intrinsics for AVX-512 complicates optimization**
- **At best a hand-coded popcnt achieves parity with instruction despite having to copy to int register**
- **SSE4.2 string comparison intrinsics are used to speedup integer comparison operations**
- **WORK IN PROGRESS**
  - Main challenges:
    - Vectorization of pairwise comparisons
    - Vectorization of computing nonzero matrix elements

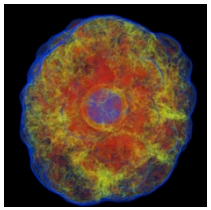
Sparse matrix-vector products

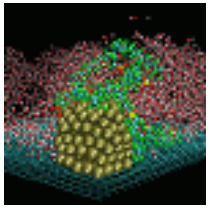












Matrix format

- **CSB_Coo format**
  - Partition matrix in k x k blocks with $k \leq \beta$
  - Coo within each block on space filling curve
  - Efficient for SPMV and SPMV_T
    - Can thread across rows or columns easily
  - 2x 16-bit for index
  - 1x 32-bit for matrix element
- **Local matrix: 10^8 x 10^8**
- **Local nonzeros: 8 x 10^9**
- **Quasi-random sparsity**
- **β =16,000**

CSB_Coo details:
Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations
IPDPS 2014
http://gauss.cs.ucsb.edu/~aydin/ipdps14aktulga.pdf

Matrix format

- CSB_Coo
- non-zeros in blue
- blocks in red
- blocks are built up out of tiles
- store only half of the symmetric matrix

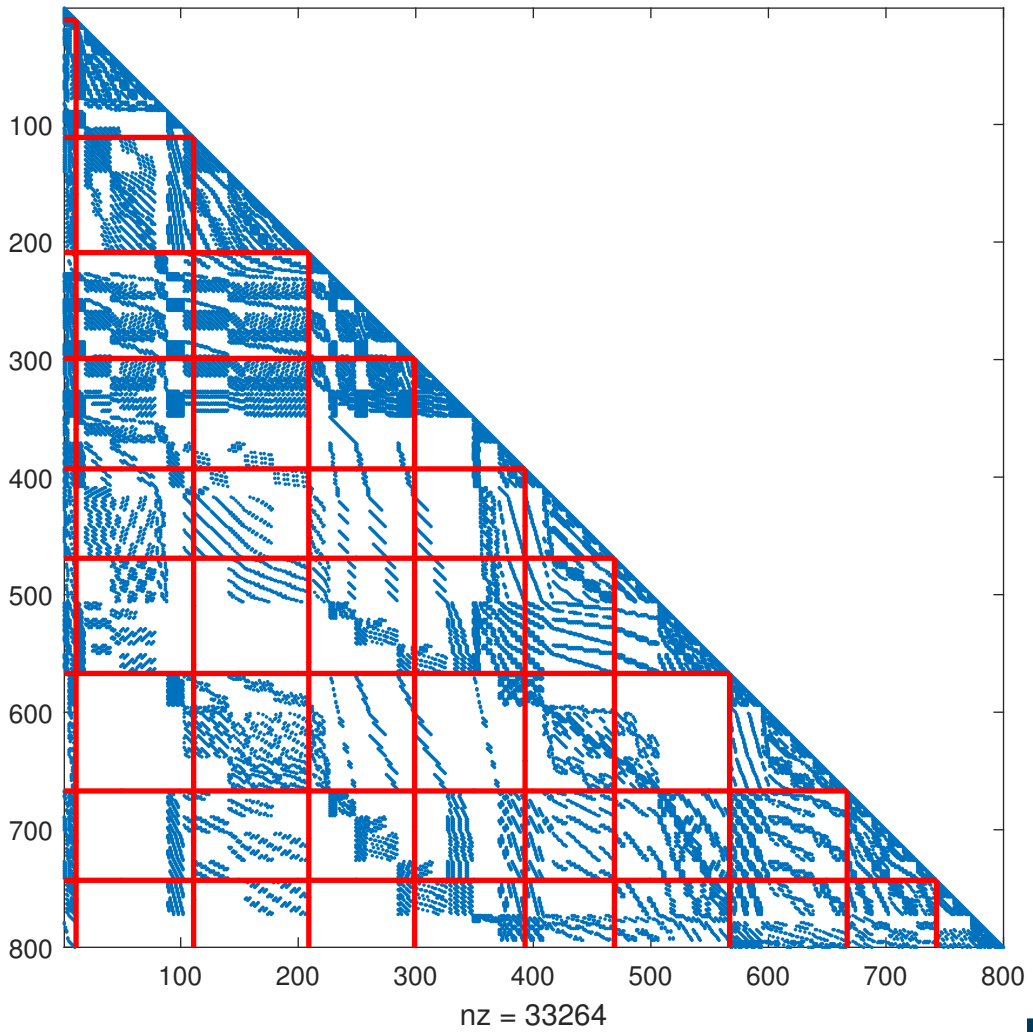


Figure: C. Yang 2016

Replace SPMV with SPMM

```
y += A*x  ──────────►  Y(1:m) += A*X(1:m)
```

2 MPI ranks and 16 OpenMP threads on Haswell

m	AI	GFLOP/s	GB/s
1	0.23	23.2	125
4	0.62	56.8	125
8	0.80	67.5	125

AI = Arithmetic Intensity = FLOPs / byte
Theoretical peak bandwidth: 137 GB/s

Replace SPMV with SPMM

```
y += A*x  ──────────→  Y(1:m) += A*X(1:m)
```

X and Y Explicitly in MCDRAM

m	AI_{DDR}	AI_{MCDRAM}	AI_{TOTAL}	GFLOP/s	DDR GB/s	MCDRAM GB/s
1	0.20	0.33	0.13	17.1	83	55
4	0.80	0.36	0.25	62.4	80	180
8	1.57	0.37	0.30	109.1	71	300

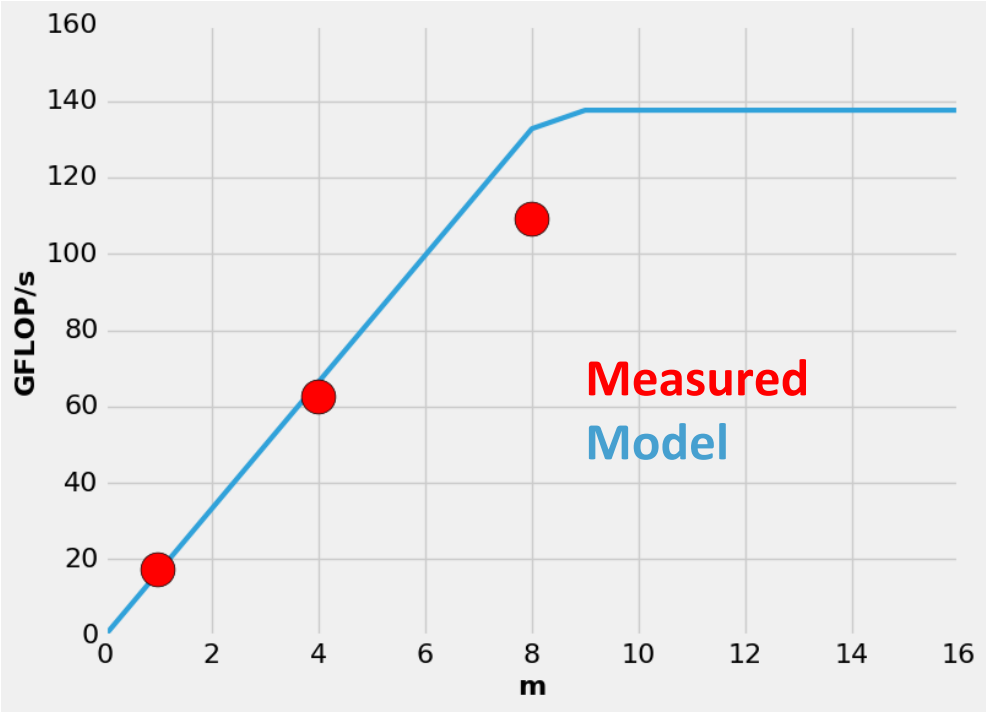
MCDRAM peak bandwidth: >460 GB/s
DDR4 peak bandwidth: 102 GB/s

MCDRAM Sustained (1R/1W): 372 GB/s
DDR4 Sustained (1R/1W): 77 GB/s

Performance model

$$\min[B_1 * AI_1(m),\ B_2 * AI_2(m)]$$

1 = DDR
2 = MCDRAM
B_i= maximum bandwidth



AI for MCDRAM ~const

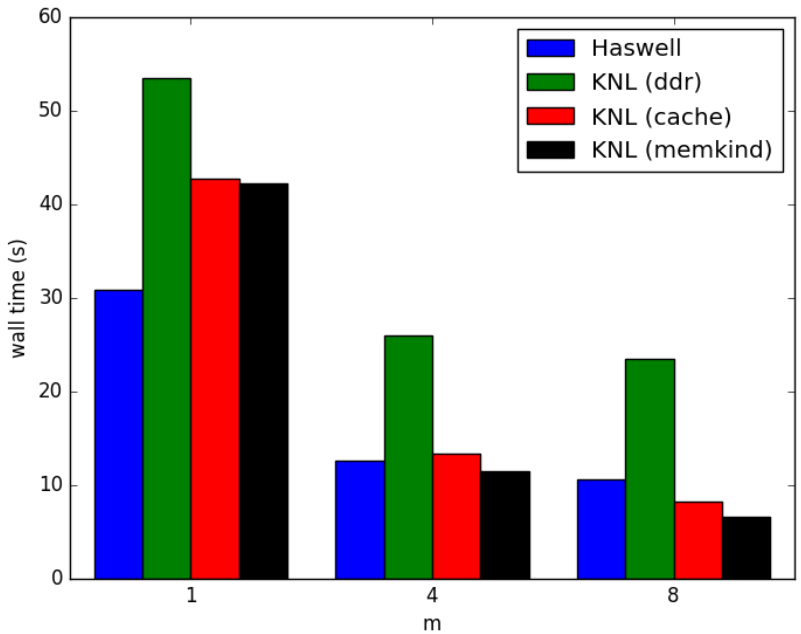
$B_1/B_2 \approx 4.5$

$AI_{ddr} \sim 0.2*m$

$AI_{mcdram} \sim 0.37$

MFDn SPMM Summary

- **Running out of DDR only is very slow**
- **Cache mode is OK, but reduces total memory**
- **Explicit management 1.2x faster than cache**
- **Multiple vectors speedup**
  - 2.9x on Haswell
  - 6.4x on KNL with memkind
- **1.6x speedup over Haswell**
  - Only with spmm



National Energy Research Scientific Computing Center