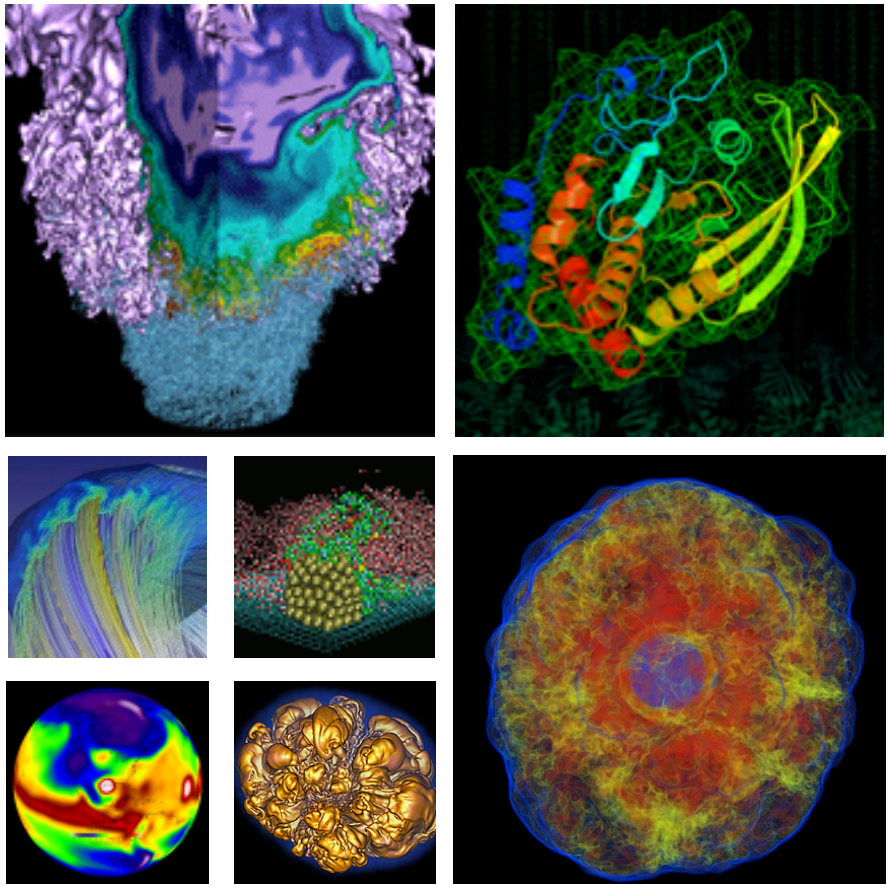
Lessons Learned from Selected NESAP Applications



Helen He

NCAR Multi-core 5 Workshop
Sept 16-17, 2015

NERSC 40 YEARS at the FOREFRONT 1974-2014

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

The Big Picture

- **The next large NERSC production system "Cori" will be Intel Xeon Phi KNL (Knights Landing) architecture**
  - Self-hosted (not an accelerator). 72 cores per node, 4 hardware threads per core
  - Larger vector units (512 bits)
  - On package high-bandwidth memory (HBM)
  - Burst Buffer

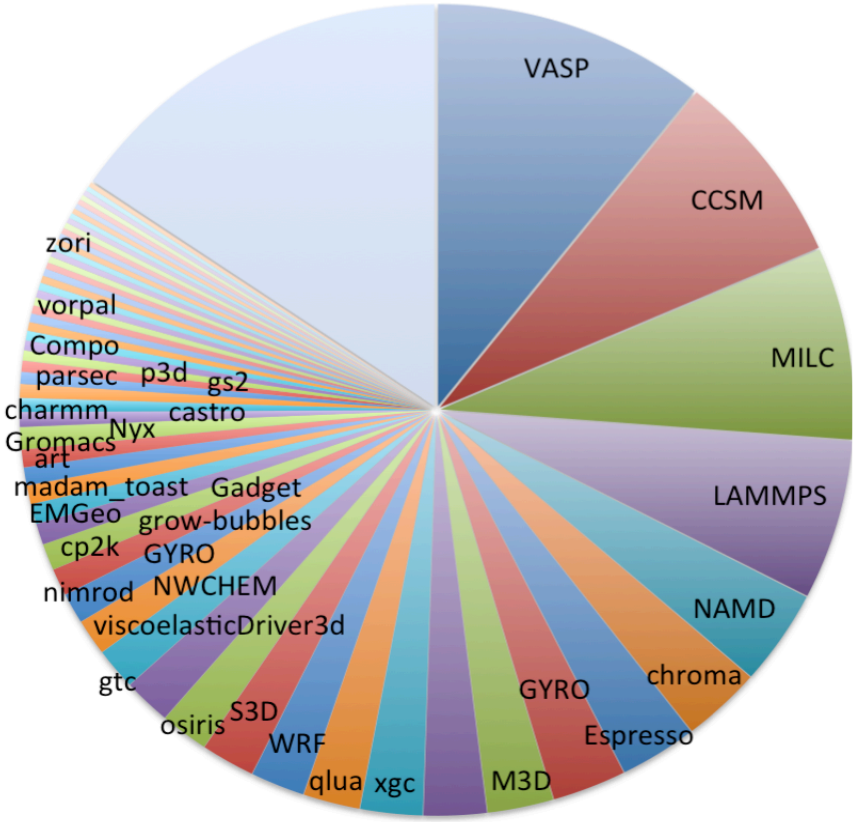
- **To achieve high performance, applications need to explore more on-node parallelism with thread scaling and vectorization, also to utilize HBM and burst buffer options.**
- **Hybrid MPI/OpenMP is a recommended programming model, to achieve scaling capability and code portability.**

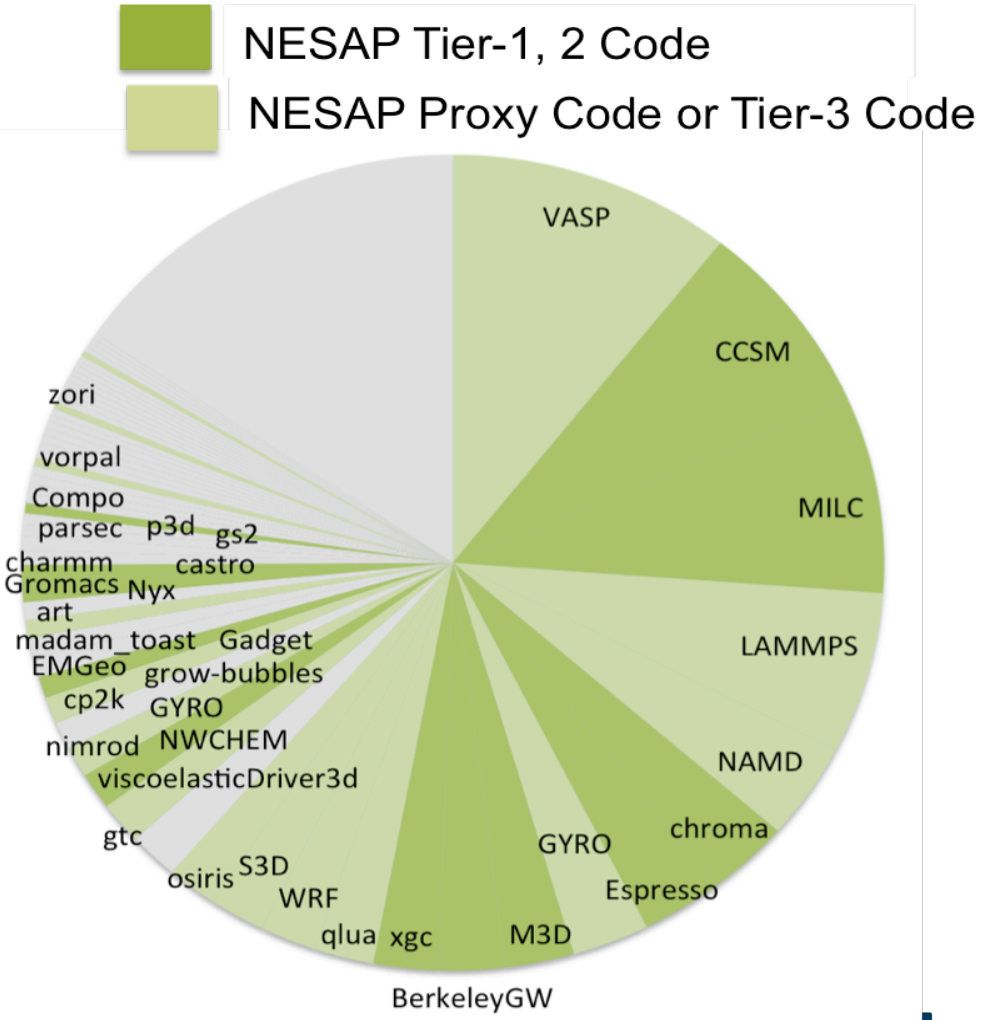
NERSC Exascale Science Application Program (NESAP)

- **Goal: to prepare DOE Office of Science user community for Cori manycore architecture**
- **20 applications were selected as Tier 1 (with postdocs) and Tier 2 applications to work closely with Cray, Intel and NERSC staff. Additional 26 Tier 3 teams. Share lessons learned with broader user community.**
- **Available resources are:**
  - Access to vendor resources and staff including "dungeon sessions" with Intel and Cray Center of Excellence
  - Early access to KNL "whitebox" systems
  - Early access and time on Cori
  - Trainings, workshops, and hackathons
  - Intel Xeon Phi User Group (IXPUG)

NESAP Code Coverage

Breakdown of Application Hours on Hopper and Edison 2013



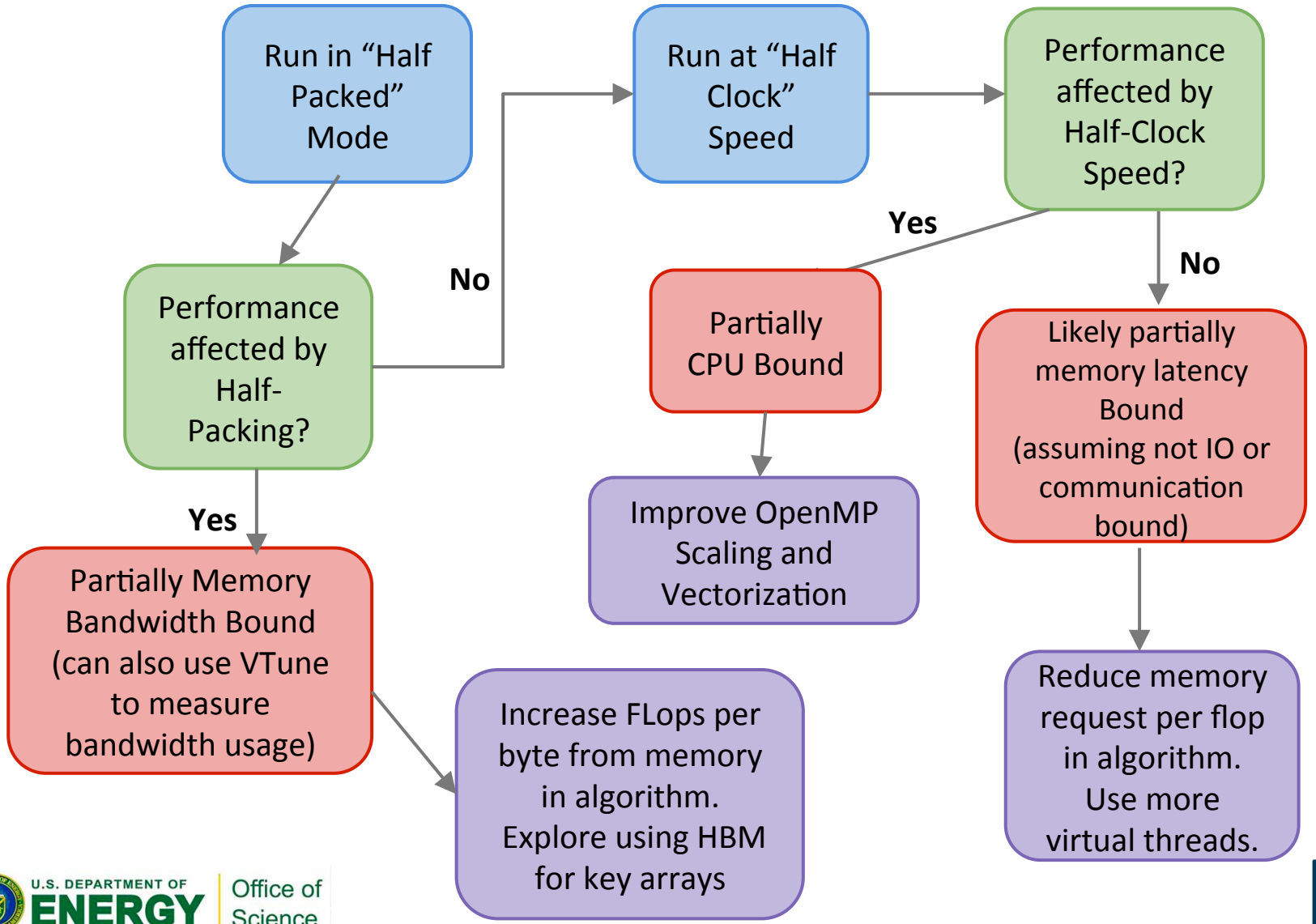


Lessons Learned from Selected Applications

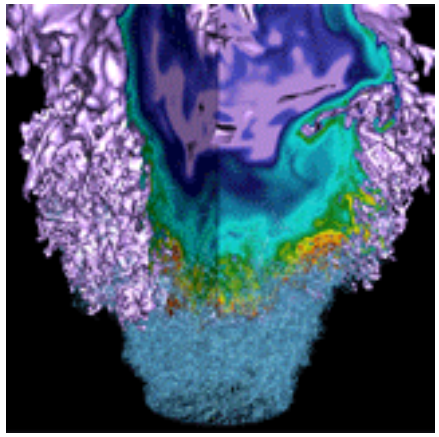
- Presentation materials contributed by NERSC Application Readiness Team (NERSC Staff) and NESAP teams (application developers, NERSC liaisons, Cray Center of Excellence staff, and Intel staff)

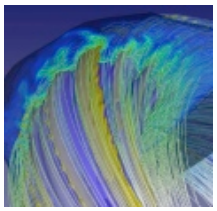
Application	Science Area	PI	NERSC Liaison
BerkeleyGW	Material Sciences	Jack Deslippe	Jack Deslippe
CESM	Climate	John Dennis	Helen He
EmGeo	Earth Science	Gregory Newman	Scott French
NWChem	Chemistry	Wibe De Jong, Eric Bylaska	Zhengji Zhao
XGC1	Fusion	Choong-Seock Chang	Helen He

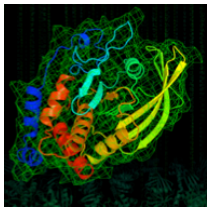
Recommended Optimization Path

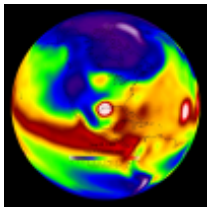


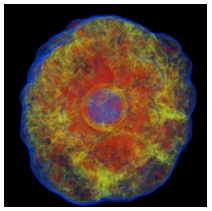
Kernel Optimizations Examples

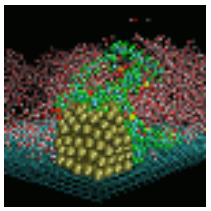










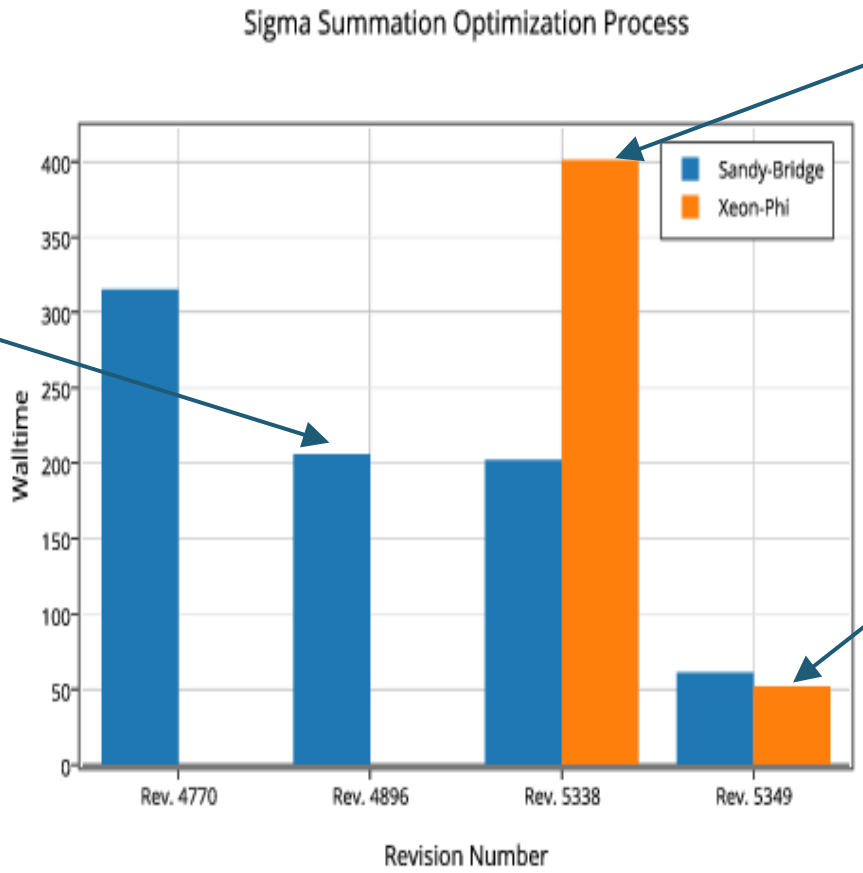


BerkeleyGW Optimization Steps

- **Target more on-node parallelism. (MPI model already failing users)**
- **Ensure key loops/kernels can be vectorized.**

Refactor to Have 3 Loop Structure:

Outer: MPI
Middle: OpenMP
Inner: Vectorization



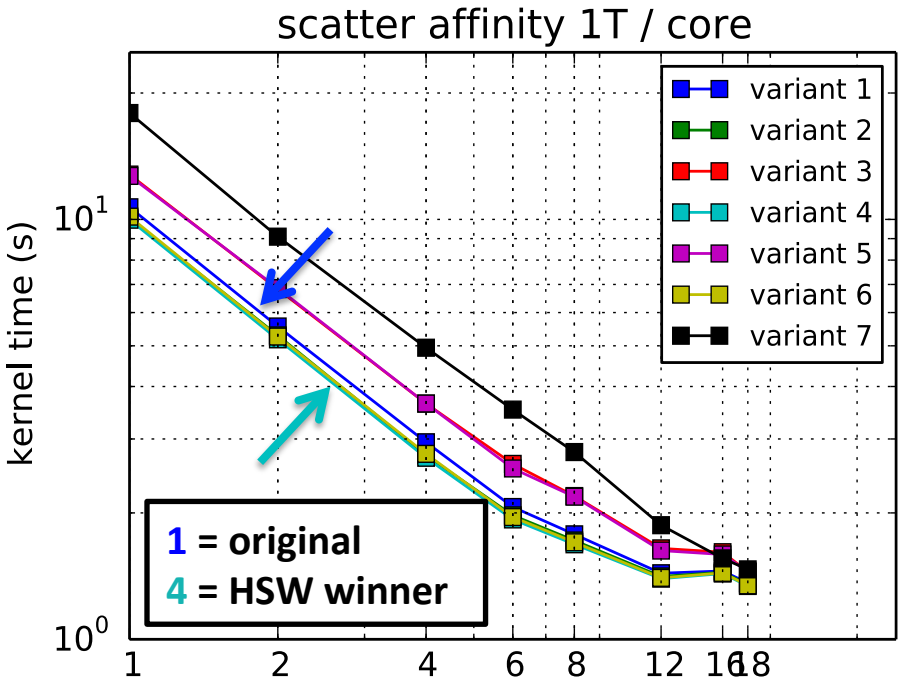
Add OpenMP

Ensure Vectorization

Emgeo: 7 SpMV Kernel Variants

- **Span the space of likely optimizations to assess performance impact on non-KNL architectures**
  - Alignment tweaks; Loop reordering, unrolling; Memory layout optimizations; Fortran "SIMD-ization"
- **Ready for profiling when we have KNL access**
- **Winner: Only ~8% speedup over the original code**
  - Only certain variants show vectorization speedup on HSW

Thread scaling on HSW EX (AVX2)



What does the code look like?

Original

```
!$omp parallel do private(j,ztmp)
  do i = 1, m
    ztmp =  (0.0d0, 0.0d0)
    do j = 1, ndiag
      ztmp = ztmp + mat(j,i) * x(ind(j,i))
    end do
    z(iorig(i)) = ztmp
  end do
```

Too many streams?

```
!$omp parallel do private(ztmp)
  do i = 1, m
    ztmp =         mat(i, 1) * x(ind(i, 1))
    ztmp = ztmp + mat(i, 2) * x(ind(i, 2))
    ztmp = ztmp + mat(i, 3) * x(ind(i, 3))
      ... snip ...
    z(iorig(i)) = ztmp
  end do
```

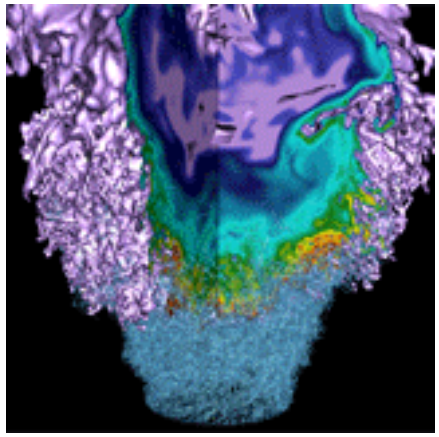
HSW winner

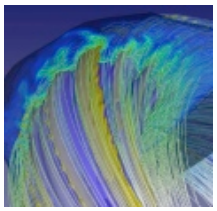
```
!$omp parallel do private(ztmp)
  do i = 1, m / SIMDWIDTH
    ztmp =         mat(:, 1,i) * x(ind(:, 1,i))
    ztmp = ztmp + mat(:, 2,i) * x(ind(:, 2,i))
    ztmp = ztmp + mat(:, 3,i) * x(ind(:, 3,i))
      ... snip ...
    z(iorig(i)) = ztmp
  end do
```

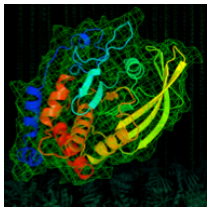
** omitting alignment-related directives, etc.

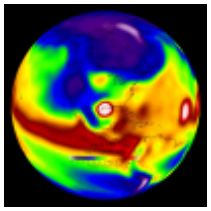
- **Some traverse many streams of data concurrently**
  - Others are more conservative (including the winning variant)
  - Will the more bandwidth-hungry variants do better on KNL? Also show largest instruction count drop from AVX2 to AVX512.

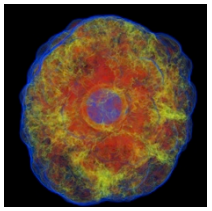
Improve OpenMP Scaling Examples

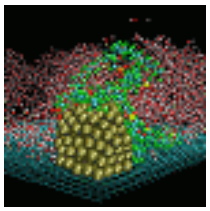












XGC1: Remove "-heap-arrays 64" Compiler Flag

- This Intel compiler flag puts automatic arrays and temp of size 64 kbytes or larger on heap instead of stack.
- Surprisingly it slows down both the collision and pushe kernels by >6X.
- Allocation and access of private copies on the heap are very expensive.
- Does not affect explicit-shape arrays.
- Removed this flag for the collision kernel, and set OMP_STACKSIZE to a large value
- Run time improves from 348 sec to 43 sec.
- Alternative: use !$OMP THREADPRIVATE. Downside: data has to be static, not allocatable.

XGC1: Explore Nested OpenMP

- **Always make sure to use best thread affinity. Avoid using threads across NUMA domains.**
- **Currently:**

```
export OMP_NUM_THREADS=6,4
export OMP_PROC_BIND=spread,close
export OMP_NESTED=TRUE
Export OMP_STACKSIZE=8000000
aprun -n 200 -N 2 -S 1 -j 2 -cc numa_node ./xgca
```

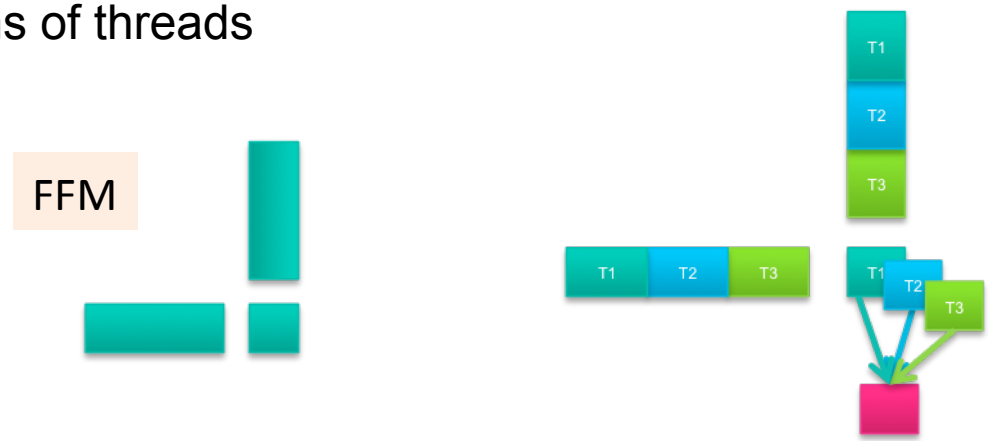
- **Is a bit slower than (work ongoing):**

```
export OMP_NUM_THREADS=24
export OMP_NESTED=TRUE
export OMP_STACKSIZE=8000000
aprun -n 200 -d 24 -N 2 -S 1 -j 2 -cc numa_node ./xgca
```

- **Refer to NERSC "Nested OpenMP" web page for achieving process and thread affinity using different compilers on different NERSC systems:**
  - https://www.nersc.gov/users/computational-systems/edison/running-jobs/using-openmp-with-mpi/nested-openmp/

NWChem: OpenMP "Reduce" Algorithm

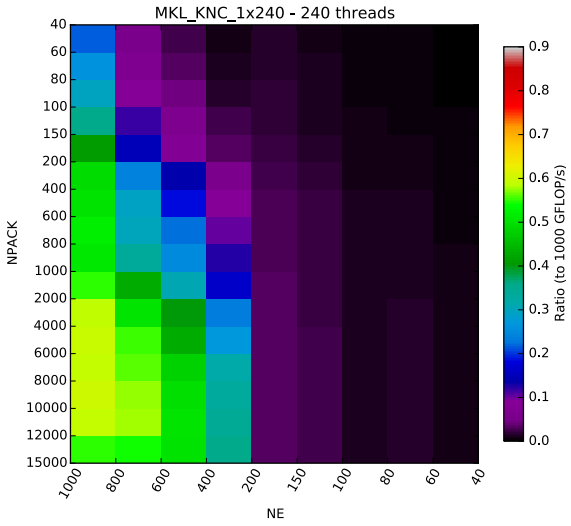
- **Plane wave Lagrange multiplier**
  - Many matrix multiplications of complex numbers, C = A x B
  - Smaller matrix products: FFM, typical size 100x10,000x100
  - Original threading scaling with MKL not satisfactory
- **OpenMP "Reduce" or "Block" algorithm**
  - Distribute work on A and B along the k dimension
  - A thread puts its contribution in a buffer of size m x n
  - Buffers reduced to produce C
  - OMP teams of threads



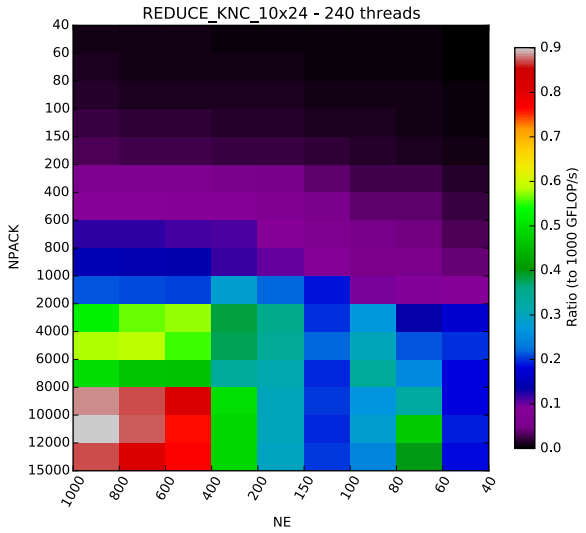
NWChem: OpenMP "Reduce" Algorithm

- Better for smaller inner dimensions, i.e. for FFMs
- Multiple FFMs can be done concurrently in different thread pools
- Threading enables us to use all 240 hardware threads
- Best Reduce: 10 MPI, 6 teams of 4 threads

MKL
1MPI, 240 threads

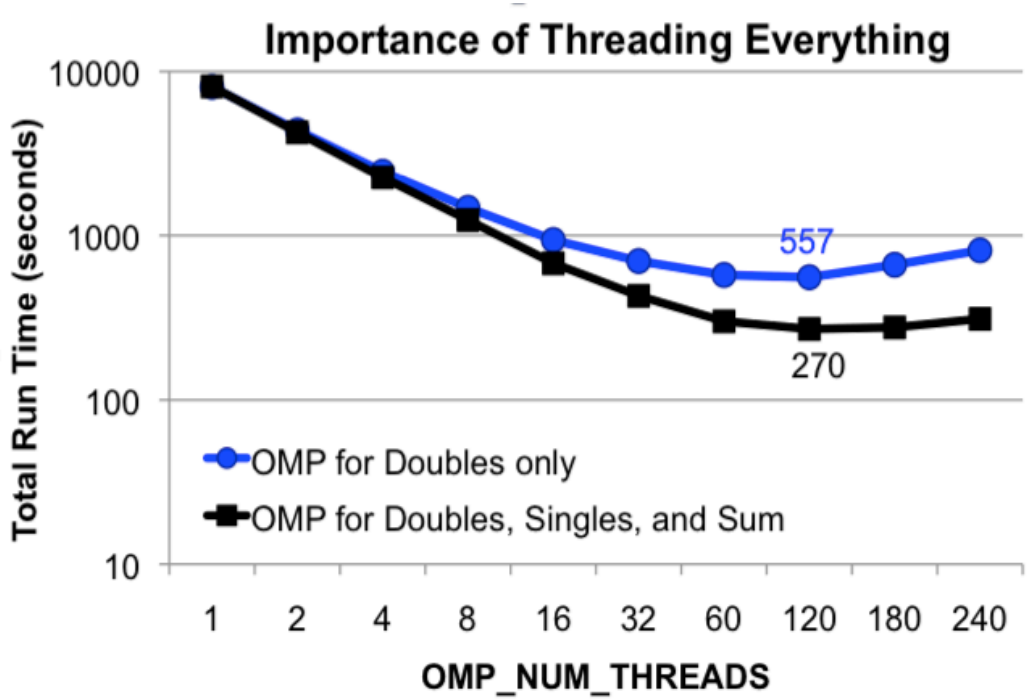


Best "Reduce"
10 MPI, 6 teams of 4 threads



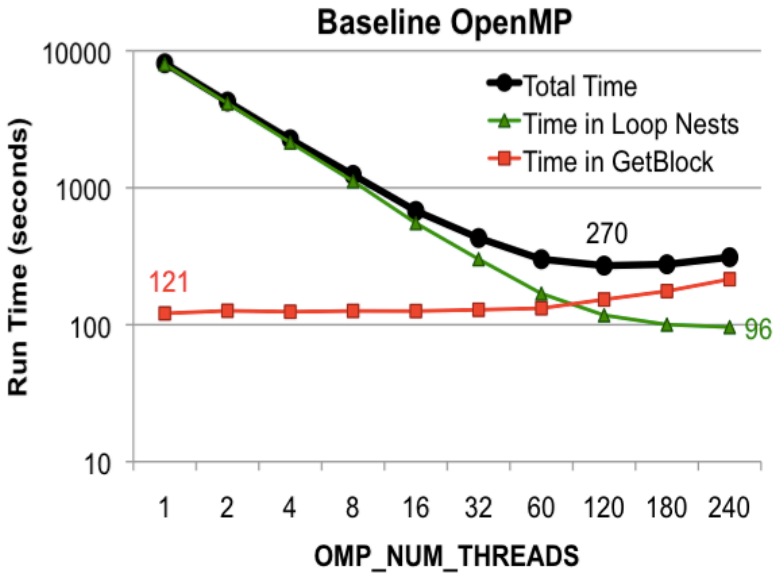
NWChem: OpenMP Scaling in CCSD(T)

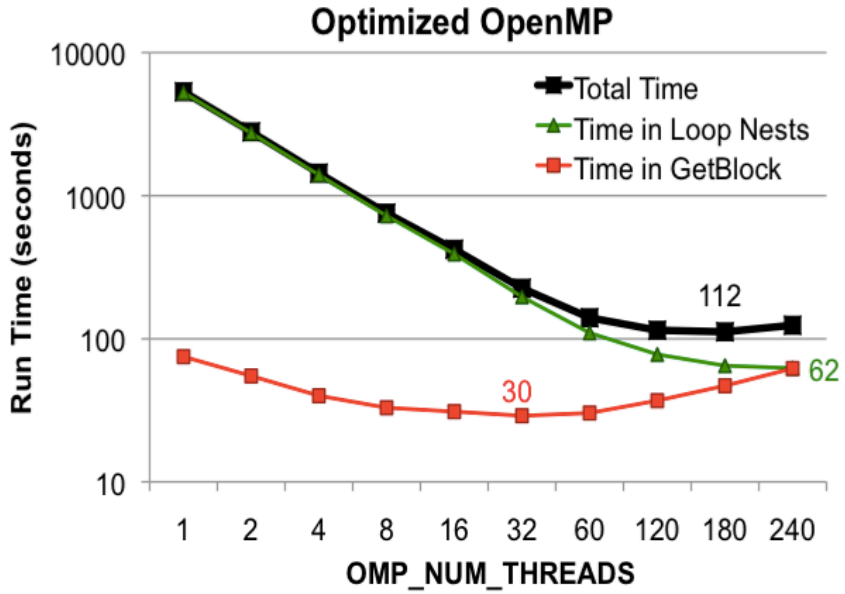
- Double terms usually dominate in (T) term
- Other terms become new performance bottleneck on many-core architectures - Amdahl's Law



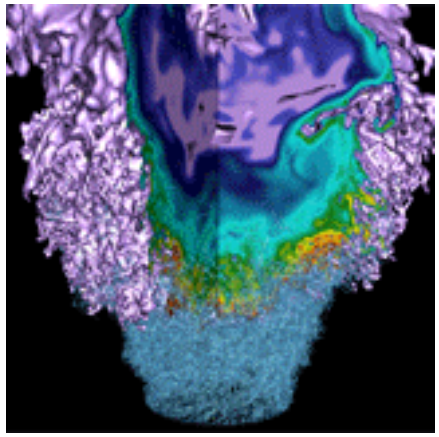
NWChem: OpenMP Scaling in CCSD(T)

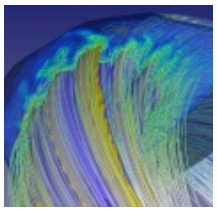
- Threading enables us to use all 240 hardware threads
- Optimized code performs 2.5X better than baseline
- Up to 65X better compared to 1 MPI rank

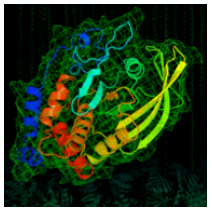


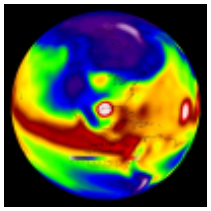


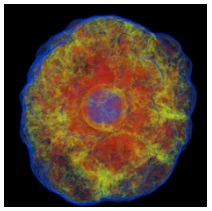
Vectorization Examples

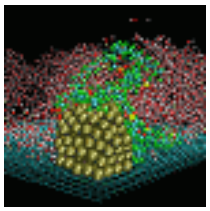












XGC1: Collision Kernel

Split dimensions, interchange array index, unroll loops, 40% kernel speedup

Original

```
real(8),dimension
     (5,(col_f_nvr-1)*(col_f_nvz-1),
     (col_f_nvr-1)*(col_f_nvz-1)) :: Ms

do index_ip = 1, mesh_Nzm1
  do index_jp = 1, mesh_Nrm1
    index_2dp = index_jp+mesh_Nrm1*(index_ip-1)

    tmp_vol = cs2%local_center_volume(index_jp)
    tmp_f_half_v = f_half(index_jp, index_ip) *
      tmp_vol
    tmp_dfdr_v = dfdr(index_jp, index_ip) *
      tmp_vol
    tmp_dfdz_v = dfdz(index_jp, index_ip) *
      tmp_vol

    tmpr(1:3)= tmpr(1:3)+
      Ms(1:3,index_2dp,index_2D)*
      tmp_f_half_v
    tmpr(5)  = tmpr(5)   +
      Ms(4,index_2dp,index_2D)*tmp_dfdr_v +
      Ms(2,index_2dp,index_2D)*tmp_dfdz_v
    tmpr(6)  = tmpr(6)   +
      Ms(3,index_2dp,index_2D)*tmp_dfdz_v +
      Ms(5,index_2dp,index_2D)*tmp_dfdr_v
  enddo !index_jp
enddo !index_ip
```

Optimized

```
real (8),dimension
          ((col_f_nvr-1),5,(col_f_nvz-1),
          (col_f_nvr-1)*(col_f_nvz-1)) :: Ms

do index_ip = 1, mesh_Nzm1
  do index_jp = 1, mesh_Nrm1
    index_2dp = index_jp+mesh_Nrm1*(index_ip-1)
    tmp_vol = cs2%local_center_volume(index_jp)
    tmp_f_half_v = f_half(index_jp, index_ip) *
          tmp_vol
    tmp_dfdr_v = dfdr(index_jp, index_ip) * tmp_vol
    tmp_dfdz_v = dfdz(index_jp, index_ip) *
          tmp_vol

    tmpr(index_jp,1) = tmpr(index_jp,1) +
          Ms(index_jp,1,index_ip,index_2D)*
          tmp_f_half_v
    tmpr(index_jp,2) = tmpr(index_jp,2) +
          Ms(index_jp,2,index_ip,index_2D)*
          tmp_f_half_v
    tmpr(index_jp,3) = tmpr(index_jp,3) +
          Ms(index_jp,3,index_ip,index_2D)*
          tmp_f_half_v
    tmpr(index_jp,5) = tmpr(index_jp,5) +
          Ms(index_jp,4,index_ip,index_2D)*
          tmp_dfdr_v +          Ms(index_jp,
2,index_ip,index_2D)* tmp_dfdz_v
    tmpr(index_jp,6) = tmpr(index_jp,6) +
          Ms(index_jp,3,index_ip,index_2D)*
          tmp_dfdz_v +          Ms(index_jp,
5,index_ip,index_2D)* tmp_dfdr_v
  enddo !index_jp
enddo !index_ip
do i=1,6
  tmpr(1,i)=sum(tmpr(:,i))
enddo
```

BerkeleyGW

3X faster on SandyBridge, 8X faster on KNC

```
!$OMP DO reduction(+:achtemp)
 do my_igp = 1, ngpown
  ...
  do iw=1,3

   scht=0D0
   wxt = wx_array(iw)

   do ig = 1, ncouls

    !if (abs(wtilde_array(ig,my_igp) * eps(ig,my_igp)) .lt. TOL) cycle

    wdiff = wxt - wtilde_array(ig,my_igp)
    delw = wtilde_array(ig,my_igp) / wdiff
    ...
    scha(ig) = mygpvar1 * aqsntemp(ig) * delw * eps(ig,my_igp)
    scht = scht + scha(ig)

   enddo ! loop over g
   sch_array(iw) = sch_array(iw) + 0.5D0*scht

  enddo

  achtemp(:) = achtemp(:) + sch_array(:) * vcoul(my_igp)

 enddo
```

ngpown typically in 100's to 1000s. Good for many threads.

Original inner loop. Too small to vectorize!

ncouls typically in 1000s - 10,000s. Good for vectorization.

Attempt to save work breaks vectorization and makes code slower.

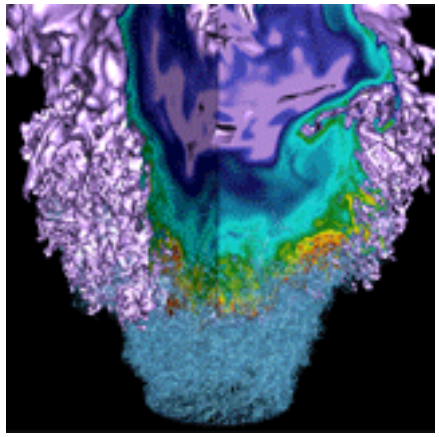
CESM MG2 Kernel: OMP SIMD ALIGNED

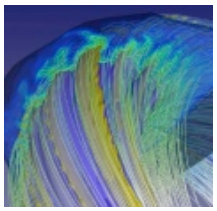
- **!$OMP SIMD ALIGNED (…)**
  - **OpenMP standard, portable**
  - Tells the compiler that particular arrays in the list are aligned
  - Asserts there are no dependencies
  - Requires to use PRIVATE or REDUCTION clauses to ensure correctness
  - Forces the compiler to vectorize, whether or not it thinks if it helps performance.
- **!DIR$ ASSUME_ALIGNED (…)**
  - Tells the compiler that particular arrays in the list are aligned
  - Intel specific, not portable
- **!DIR$ VECTOR_ALIGNED**
  - Tells the compiler all arrays in a loop are aligned
  - Intel specific, not portable

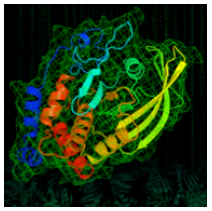
CESM MG2 Kernel: OMP SIMD ALIGNED

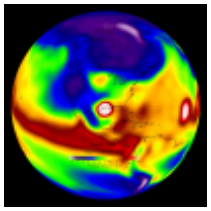
- **Using the "ALIGNED" attribute achieved 8% performance gain when the list is explicitly provided.**
- **However, the process is tedious and error-prone, and often times impossible in large real applications.**
  - !$OMP SIMD ALIGNED added in 48 loops in MG2 kernel, many with list of 10+ variables
- **Inquired with Fortran Standard:**
  - Equivalent of "!$DIR ATTRIBUTES ALIGNED: 64 :: A"
    - C/C++ standard: float A[1000] __attribute__((aligned(64)));
    - Not in Fortran standard yet
  - Equivalent of the "-align array64byte" compiler flag
    - Exist in Intel (Fortran only) and Cray compilers
    - What about other compilers?

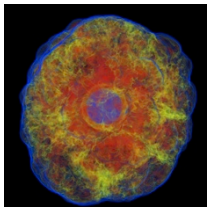
Using HBM Examples

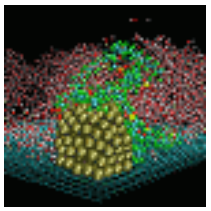












Simulate HBM Effect on a Dual Socket System

- **Identify the candidate (key arrays) for HBM**
  - VTune Memory Access tool can help to find key arrays
  - Using NUMA affinity to simulate HBM on a dual socket system
  - Use FASTMEM directives and link with jemalloc/memkind libraries

On Edison (NERSC Cray XC30):

```
real, allocatable :: a(:,:), b(:,:), c(:)
!DIR$ ATTRIBUTE FASTMEM :: a, b, c
% module load memkind jemalloc
% ftn -dynamic -g -O3 -openmp mycode.f90
% export MEMKIND_HBW_NODES=0
% aprun -n 1 -cc numa_node numactl --membind=1 --cpunodebind=0 ./myexecutable
```

On Haswell:

```
% numactl --membind=1 --cpunodebind=0 ./myexecutable
```

Application	All memory on far memory	All memory on near memory	Key arrays on near memory
BerkeleyGW	baseline	52% faster	52.4% faster
EmGeo	baseline	40% faster	32% faster
XGC1	baseline		24% faster

Conclusions

- NERSC is bringing a lot of resources to help users: training, postdocs, Cray and Intel staff, deep dive sessions.
- Optimizing code for Cori will likely require good OpenMP scaling, Vectorization and/or effective use of HBM.
- Applications can optimize on SandyBridge, IvyBridge, Haswell, and KNC architectures to prepare for Cori.
- Always profiling and understand your code first on where to work on improving performance. Use tools such as VTune, vector advisor.
- Creating kernels is much more efficient than working on full codes.
- Optimizing your code targeting KNL will improve performance on all architectures.
- Keep portability in mind, use portable programming models.

Thank you.