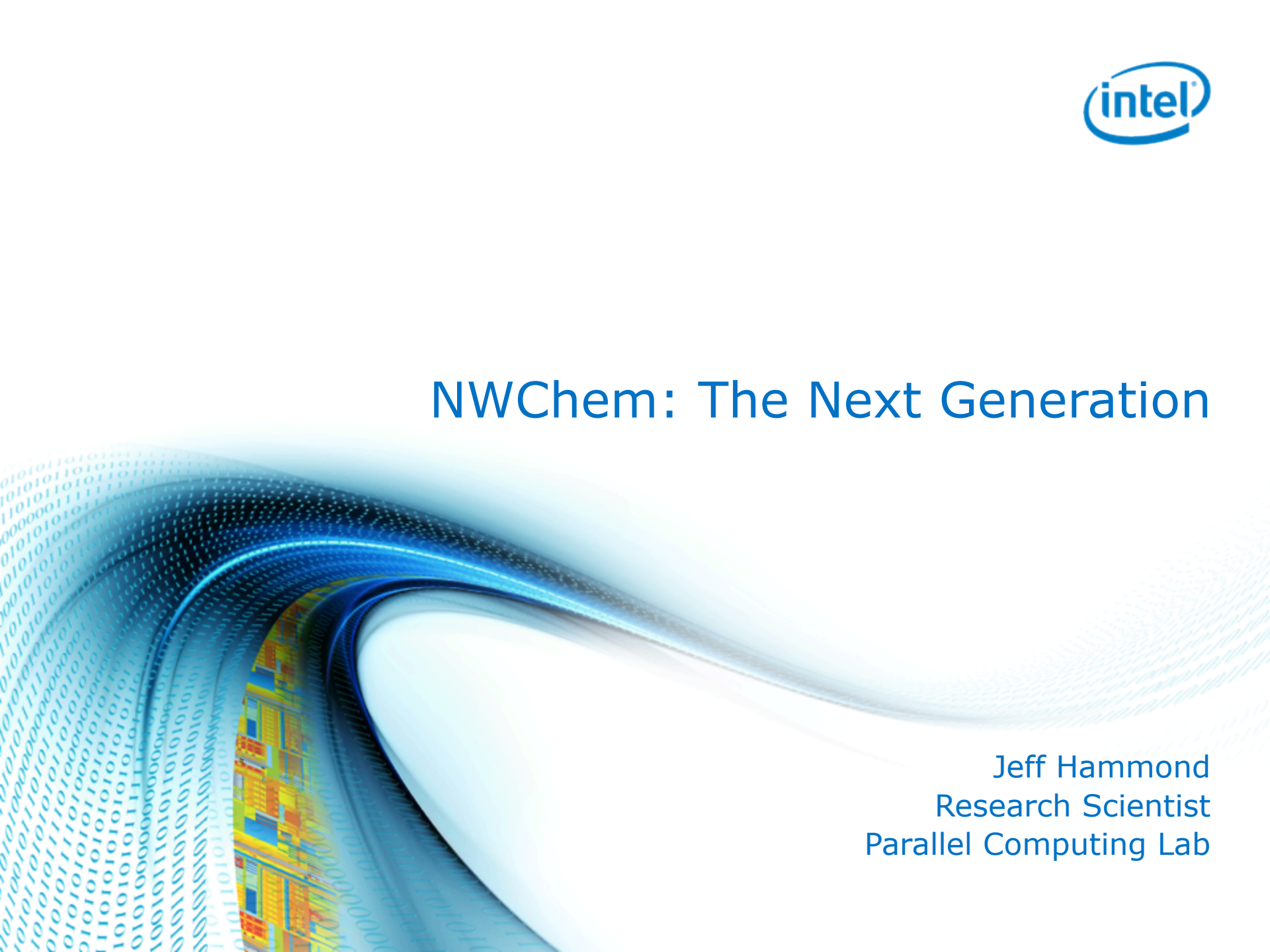


NWChem: The Next Generation

Jeff Hammond
Research Scientist
Parallel Computing Lab

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

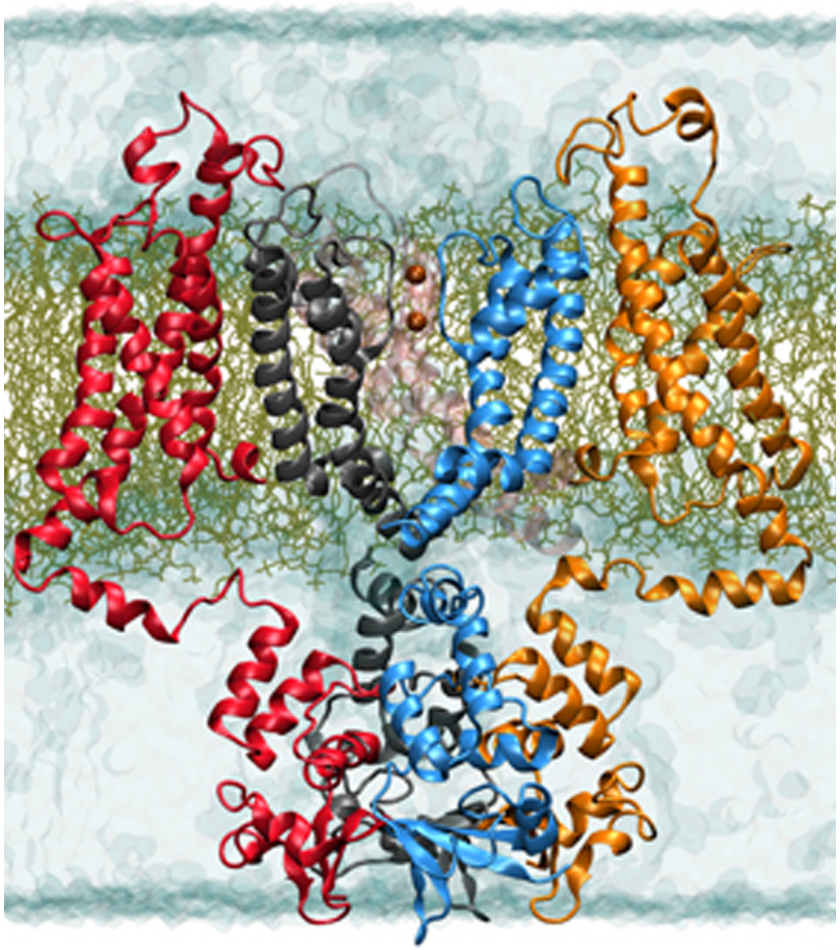
Extreme Scalability Group Disclaimer

- I work in Intel Labs and therefore don't know anything about Intel products.
- I work for Intel, but I am not an official spokesman for Intel. Hence anything I say are my words, not Intel's. Furthermore, I do not speak for my collaborators, whether they be inside or outside Intel.
- You may or may not be able to reproduce any performance numbers I report.
- Hanlon’s Razor.

Atomistic simulation in chemistry

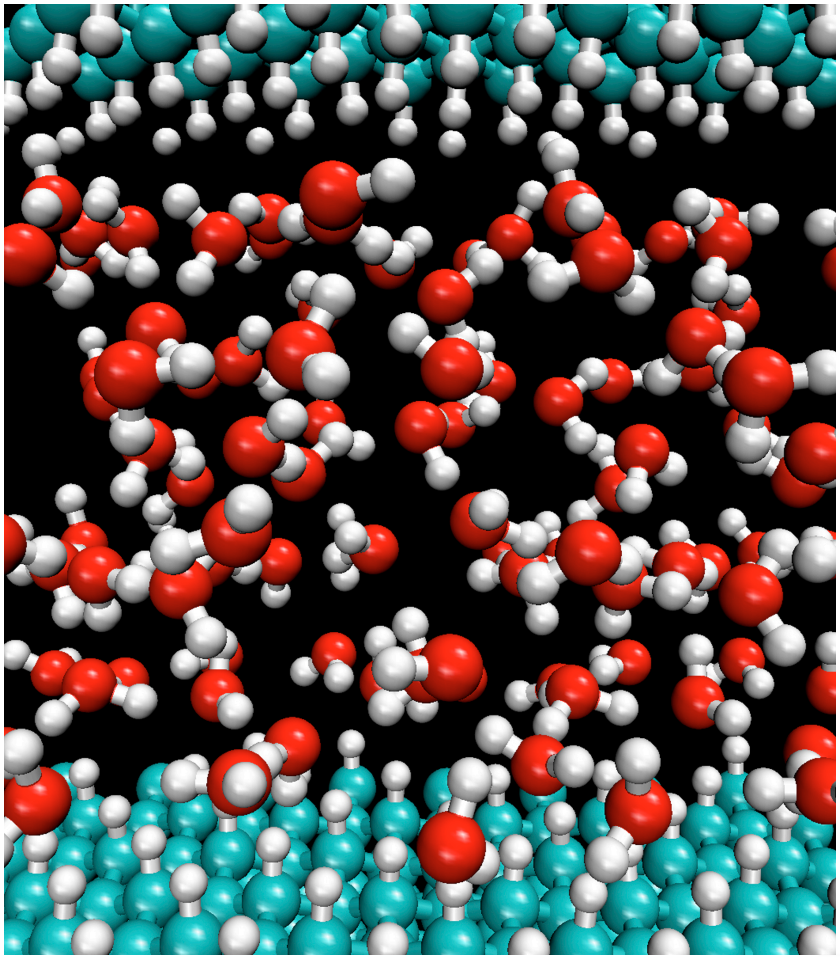
1. *Classical* molecular dynamics (MD) with empirical potentials
2. *Quantum* molecular dynamics based upon **density**-function theory (DFT)
3. *Quantum* chemistry with **wavefunctions** e.g. perturbation theory (PT), coupled-cluster (CC) or quantum monte carlo (QMC).

Classical molecular dynamics



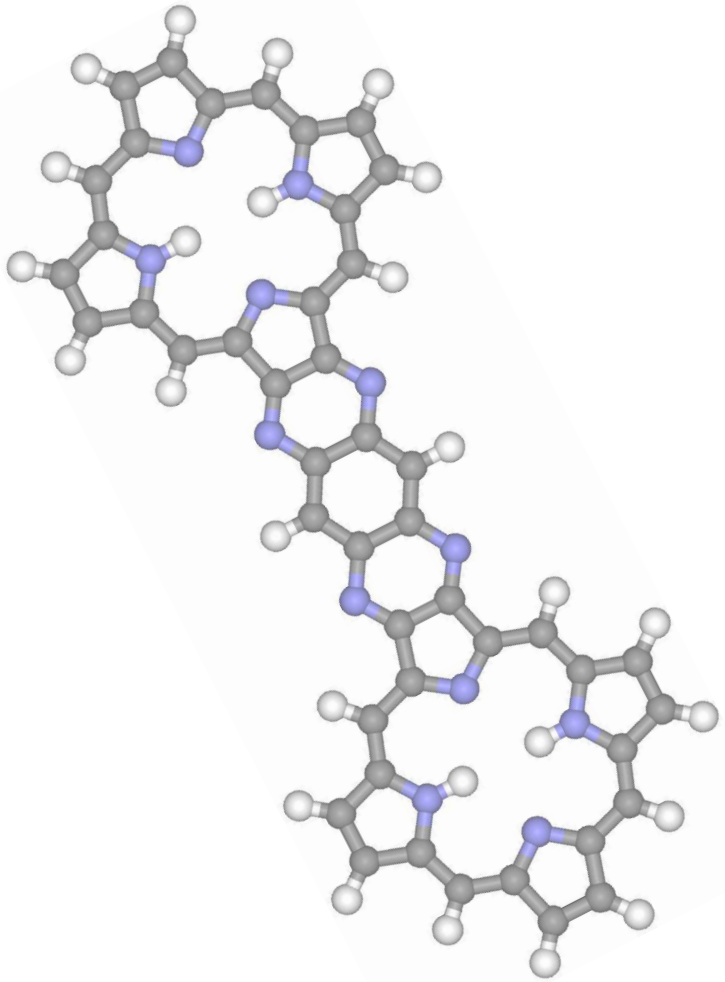
- Solves Newton's equations of motion with empirical terms and classical electrostatics.
- Size: K to M of atoms
- Time: nanoseconds per day
- Scaling: $O(N_{atoms})$
- Math: N-body

Quantum molecular dynamics



- Forces obtained from solving an approximate single-particle Schrodinger equation.
- Size: hundreds of atoms
- Time: picoseconds per day
- Scaling: $O(N_{el}^3)$
- Math: 3D FFT, dense linear algebra.

Wavefunction theory



- Properties obtained from solving an approximate many-particle Schrodinger equation, truncated using perturbation theory or clusters.
- Size: dozens of atoms
- Time: femtoseconds per day (almost useless)
- Scaling: $O(N_{bf}^{x})$, x=5-7
- Math: Tensor contractions.

NWChem*

- Designed from the ground-up at the dawn of MPP/ cluster revolution; focused on true HPC systems.
- Developed at the EMSL at PNNL
  - EMSL: Environmental Molecular Sciences Laboratory
  - PNNL: Pacific Northwest National Laboratory
- URL: http://www.nwchem-sw.org
- Open-source Apache*-like (ECL 2.0) license.
- Portable to essentially every machine on earth, with some effort (more on this later).
- Supports essentially all of the common methods: DFT, QM/MM, AIMD, CC, MP2, MCSCF, etc.

Collaborators





- Karol Kowalski – NWChem Team Lead, PNNL
- Edo Apra – Senior NWChem Developer, PNNL
- Michael Klemm – Senior Appl. Engineer, Intel SSG
- Jim Dinan – HPC Network Arch., Intel DCG
- Pavan Balaji – MPICH Team Lead, Argonne

Coupled Cluster

- Explicit electron correlation using quantum many-body perspective (see also perturbation theory).
- Size-extensive: can be used for large systems.
- Truncated in orders CCSD, CCSDT, CCSDTQ,…
- CCSD(T) most accurate per compute cost, requiring $O(n_{iter}*N^6+N^7)$ flops and $O(N^4)$ storage.
- All CC methods are cast in terms of a variety of tensor contractions; these are often mapped to DGEMM for performance.

$$T_{k,i,j}^{b,a,c} \leftarrow D_d^{a,b,i} V_{k,j,c}^d$$

Tensor Contractions

$$I_{kl}^{ij} \leftarrow V_{ef}^{ij} T_{kl}^{ef}$$

$$I_{(kl)}^{(ij)} \leftarrow V_{(ef)}^{(ij)} T_{(kl)}^{(ef)}$$

$$I_{a}^{b} \leftarrow V_{c}^{b} T_{a}^{c}$$

$$I_{bj}^{ia} \leftarrow V_{be}^{im} T_{mj}^{ea}$$

$$I_{bj,ia} \leftarrow V_{be,im} T_{mj,ea}$$

$$J_{bi,ja} \leftarrow W_{bi,me} U_{me,ja}$$

$$J_{bi}^{ja} \leftarrow W_{bi}^{me} U_{me}^{ja}$$

$$J_{(bi)}^{(ja)} \leftarrow W_{(bi)}^{(me)} U_{(me)}^{(ja)}$$

$$J_{x}^{z} \leftarrow W_{x}^{y} U_{y}^{z}$$

The V->W, T->U and J->I transpositions require O(mn+mk+kn) mops, as compared to the O(mnk) flop cost of DGEMM, which also requires O(mn+mk+kn) mops. However, the transpositions are necessarily strided access.

DGEMM Consider Harmful

$$T_{k,i,j}^{b,a,c} \leftarrow D_{d}^{a,b,i} V_{k,j,c}^{d}$$

TCE uses tiling to decompose data and achieve load-balance. In CCSD(T), the tilesize T is 16-24.

Here (m,n,k)=(T^3,T^3,T), which means $O(T^6)$ mops and $O(T^7)$ flops.

Compare T/4 flop/mop to misses associated with strided memory access.

There are 18+9 different permutation variants of the above, so persistent redistribution is impossible.

DGEMM Consider Harmful

NERSC Edison
T=16 using 48 threads (2x12x2)

DGER(T^4,T^2): 1.1 GF/s
DGEMM(T^3,T^3,T): 34.1 GF/s
DGEMM(T^3,T^3,T^{2+}): 435.3 GF/s (peak: 460 GF/s)

Loops (DGER-equiv.): 3.97-4.80 GF/s
Loops (DGEMM-equiv.): 55.2-71.7 GF/s

O(T^6) and O(T^7) contractions take the same time…

DGEMM Consider Harmful

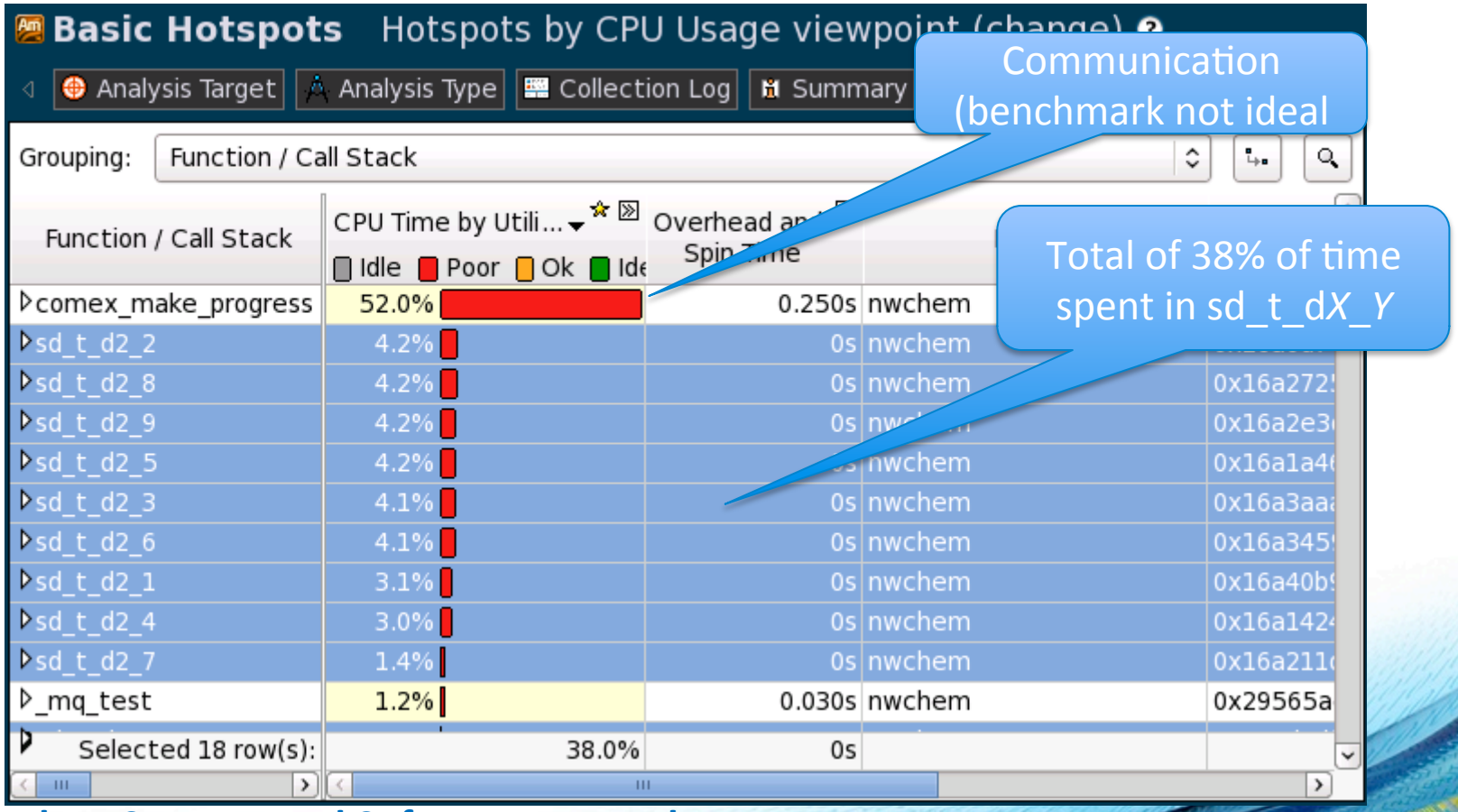
There is nothing wrong with Intel(R) MKL(TM) – all BLAS libraries have issues with small-k DGEMM, and small (m,n,k)-DGEMM in general.

Even if DGEMM is perfect, loops are still faster because they do half the memory access.

The ideal solution is a framework for generating DGEMM-quality tensor contractions for based upon known tensor layouts and dimensions. Until then, the Intel(R) Fortran compiler will have to suffice ☺

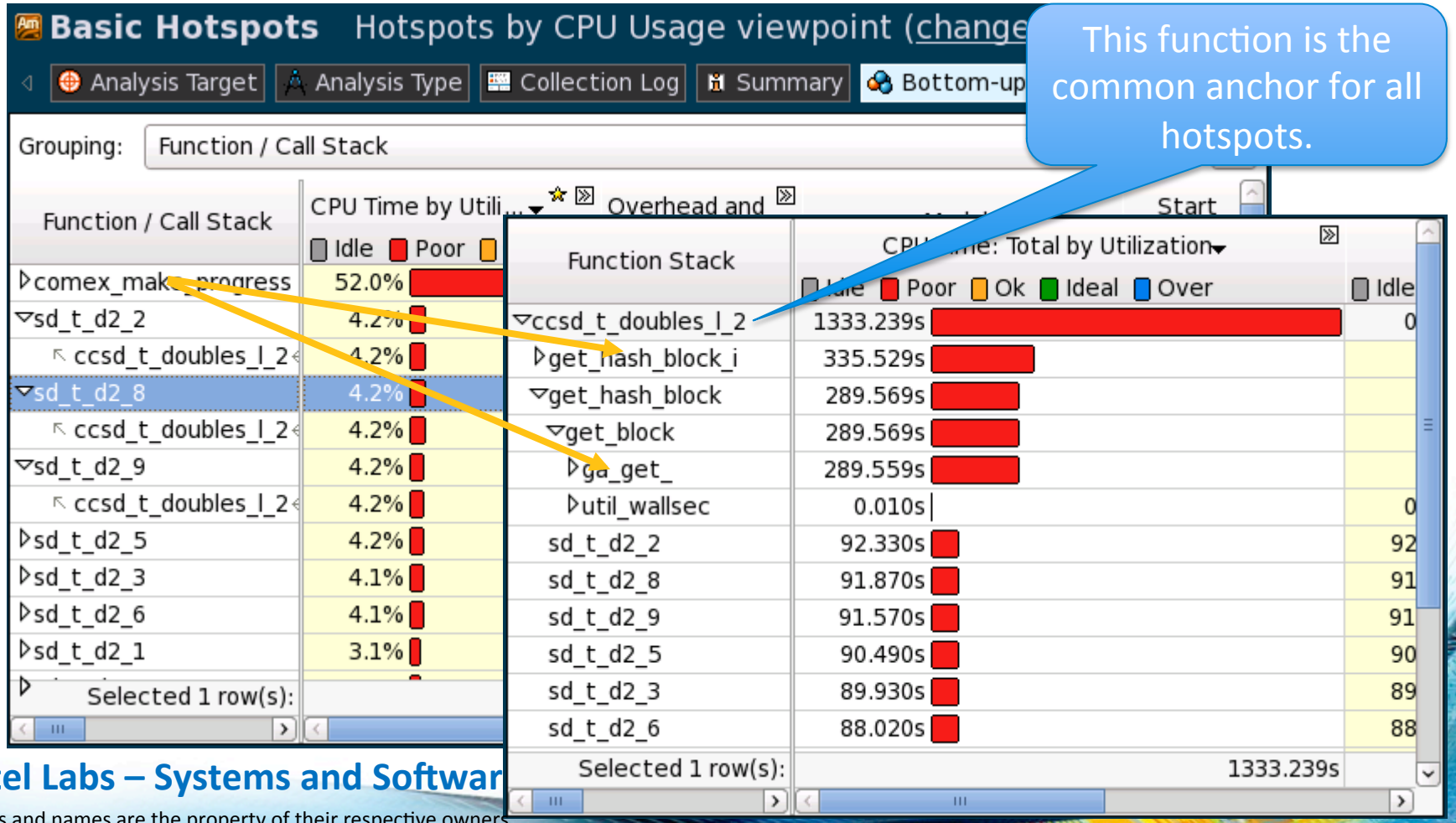
Example: NWChem Hotspots

- NWChem hotspot profile



Example: NWChem Hotspots

- Call-tree analysis shows relationship of hotspots



Example: Loop Analysis

- All kernels expose the same structure
- 7 perfectly nested loops
- Trip count per loop is equal to "tile size" (20-30)
- Naïve per-kernel solution is obvious

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d,h2d,p6d,h7d)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h2=1,h2d
      do h3=1,h3d
      do h7=1,h7d
        triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
      end
```

offload

multi-threading

SIMD

Issues w/ Naïve Offload Solution

Offloading individual kernels requires GBs of data transfers (1-2 GB per offload)

Outer loop does not expose enough parallelism (20-30 threads)

Vectorization potential too low (about 80%)

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d,h2d,p6d,h7d)
cdir$ offload target(mic) in(t2sub:length(h7d*p4d*p5d*h1d))
     1        in(v2sub:length(h3d*h2d*p6d*h7d))
     2        inout(triplesx:length(h3d*h2d*h1d*p6d*p5d*p4d))
!$omp parallel do
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h2=1,h2d
      do h3=1,h3d
      do h7=1,h7d
        triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
!$omp end parallel do
      end
```

offload

multi-threading

SIMD

Optimization of Data Transfers

- Use call-tree analysis to find common anchor for hotspots
  - Hoist data transfers up as high as possible
  - Make offload regions as large as possible

data env.

```
cdir$ offload_transfer target(mic) nocopy(triplesx:length(triplesx_l) ALLOC)
cdir$ offload_transfer target(mic) nocopy(t2sub:length(t2sub_l) ALLOC)
cdir$ offload_transfer target(mic) nocopy(v2sub:length(v2sub_l) ALLOC)
cdir$ offload target(mic) nocopy(triplesx:length(0) REUSE)
      call zero_triplesx(triplesx)
      do ...
        if (...)
cdir$ offload target(mic) in(triplesx:length(0),REUSE)
     1        in(t2sub:length(2sub_l),REUSE)
     3        in(v2sub:length(v2sub),REUSE)
     2        in(h3d,h2d,h1d,p6d,p5d,p4d,h7d)
          call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2sub,v2sub)
        endif

c     sd_t_d1_2 until sd_t_d1_9
      enddo
c     Similar structure for sd_t_d2_1 until sd_t_d2_9

cdir$ offload_transfer target(mic) out(triplesx:length(triplesx_l) REUSE)
```

offload

Kernel Optimizations

Outer loop does not expose enough parallelism (20-30 threads)

Use `collapse` clause to increase parallelism by 1-2 orders of magnitude.

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d,h2d,p6d,h7d)
!$omp parallel do collapse(3)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h2=1,h2d
      do h3=1,h3d
      do h7=1,h7d
        triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
!$omp end parallel do
      end
```

multi-threading

Kernel Optimizations, Part 2

- Loop ordering not optimal for SIMD execution
  - Too low trip count for inner loop
- Index analysis shows that loops can be reordered
  - Swap h7 and h3
  - Swap h7 and h2 again

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d,h2d,p6d,h7d)
!$omp parallel do collapse(3)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h2=1,h2d
      do h3=1,h3d
      do h7=1,h7d
        triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
!$omp end parallel do
      end
```

multi-threading

Kernel Optimizations, Part 2

- Loop ordering not optimal for SIMD execution
  - Too low trip count for inner loop
- Index analysis shows that loops can be reordered
  - Swap h7 and h3
  - Swap h7 and h2 again
- Loops h2 and h3 can be collapsed

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d,h2d,p6d,h7d)
!$omp parallel do collapse(3)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2=1,h2d
      do h3=1,h3d
        triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
!$omp end parallel do
      end
```

multi-threading

Kernel Optimizations, Part 2

- Loop ordering not optimal for SIMD execution
  - Too low trip count for inner loop
- Index analysis shows that loops can be reordered
  - Swap h7 and h3
  - Swap h7 and h2 again
- Loops h2 and h3 can be collapsed

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      double precision triplesx(h3d,h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d,h2d,p6d,h7d)
!$omp parallel do collapse(3)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2=1,h2d
      do h3=1,h3d
        triplesx(h3,h2,h1,p6,p5,p4)=triplesx(h3,h2,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
!$omp end parallel do
      end
```

multi-threading

Kernel Optimizations, Part 2

- Loop ordering not optimal for SIMD execution
  - Too low trip count for inner loop
- Index analysis shows that loops can be reordered
  - Swap h7 and h3
  - Swap h7 and h2 again
- Loops h2 and h3 can be collapsed

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d*h2d,p6d,h7d)
!$omp parallel do collapse(3)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2h3=1,h2d*h3d
        triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
      enddo
      enddo
      enddo
      enddo
      enddo
      enddo
!$omp end parallel do
      end
```

multi-threading

SIMD

about 50% speed-up

Kernel Optimizations, Multi-versioning

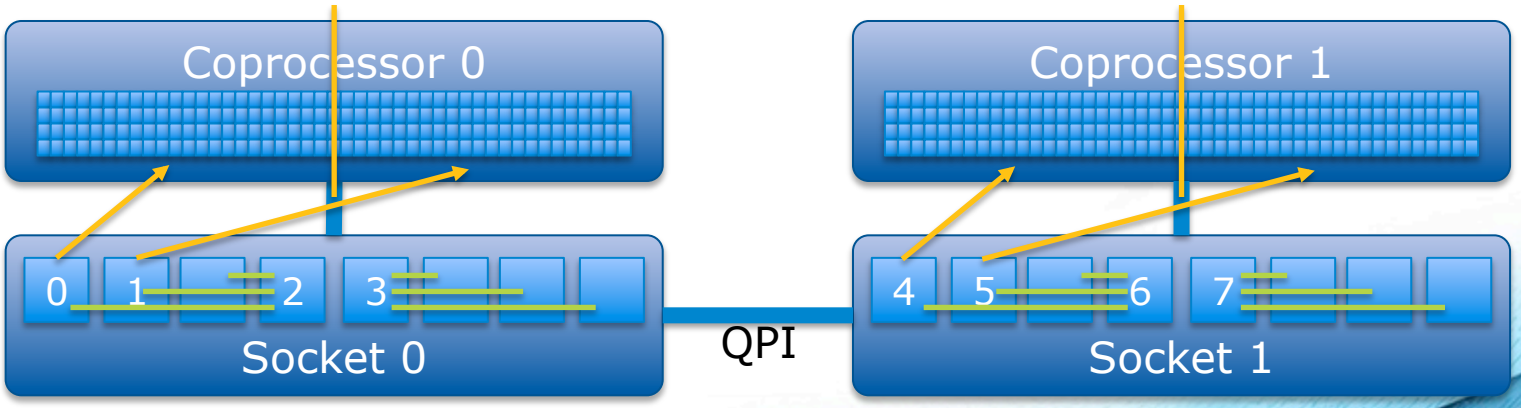
```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                     h7d,triplesx,t2sub,v2sub)
      implicit none
      integer h3d,h2d,h1d,p6d,p5d,p4d,h7d
      integer h3,h2,h1,p6,p5,p4,h7
      integer rmndr
      double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d*h2d,p6d,h7d)
      rmndr = mod(h3d,8) + mod(h2d,8) + mod(h1d,8) +
     1        mod(p6d,8) + mod(p5d,8) + mod(p4d,8) +
     2        mod(h7d,8)
      if (rmndr.eq.0) then
!$omp parallel do collapse(3)
        do p4=1,p4d
        do p5=1,p5d
        do p6=1,p6d
        do h1=1,h1d
        do h7=1,h7d
!dec$ vector aligned
        do h2h3=1,h2d*h3d
          triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1    - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
        enddo
        enddo
        enddo
        enddo
        enddo
        enddo
!$omp end parallel do
```

```
c     continued from left column
      else
!$omp parallel do collapse(3)
        do p4=1,p4d
        do p5=1,p5d
        do p6=1,p6d
        do h1=1,h1d
        do h7=1,h7d
        do h2h3=1,h2d*h3d
          triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1     - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
        enddo
        enddo
        enddo
        enddo
        enddo
        enddo
        enddo
!$omp end parallel do
      endif
      end
```

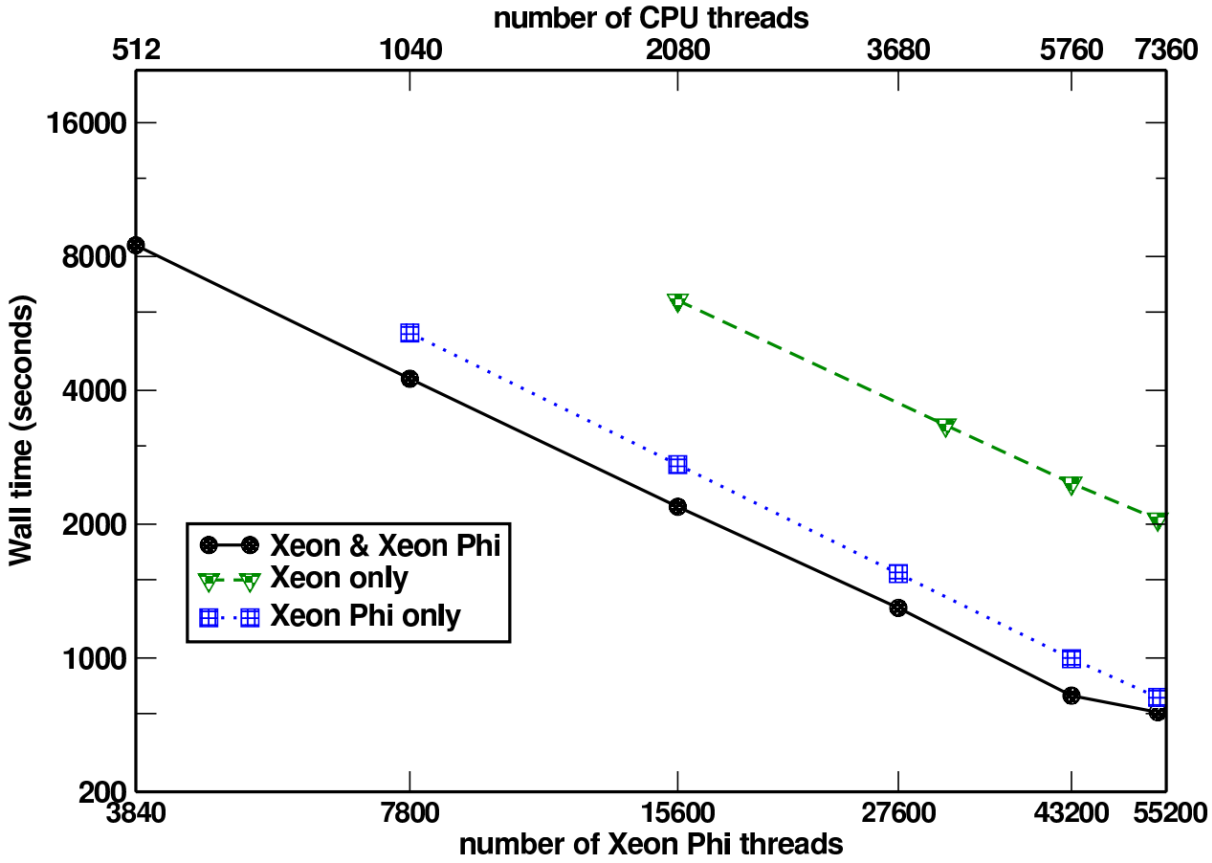
about 15% speed-up

Device Partitioning

- Host executes several MPI ranks
  - Utilize coprocessor from several host processes concurrently
  - Utilize host CPUs for increased performance
- Partition coprocessors through OpenMP* runtime
  - Less threading overhead, better overall system utilization
  - Rank 0: OFFLOAD_DEVICES=0 KMP_PLACE_THREADS=30c,4t,0o
  - Rank 1: OFFLOAD_DEVICES=0 KMP_PLACE_THREADS=30c,4t,30o
  - Rank 4: OFFLOAD_DEVICES=1 KMP_PLACE_THREADS=30c,4t,0o
  - Rank 5: OFFLOAD_DEVICES=1 KMP_PLACE_THREADS=30c,4t,30o



Performance Results



Performance tests are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. System configuration: Atipa Visione vf442 server with two Intel Xeon E5-2670 8-core processors at 2.6 GHz (128 GB DDR3 with 1333 MHz, Scientific Linux release 6.5) and Intel C600 IOH, two Intel Xeon Phi coprocessors 5110P (GDDR5 with 3.6 GT/sec, driver v3.1.2-1, flash image/micro OS 2.1.02.0390, Intel Composer XE 14.0.1.106). Benchmark perturbative triples correction to the CCSD(T) correlation energy of the 1,3,4,5-tetrasilylimidazol-2-ylidene molecule (formula $Si_4C_3N_2H_{12}$) in its triplet state.

Analysis

Profiling to identify optimization and offload opportunities is ubiquitous.

Kernel tuning involved:

1. Exposing fine-grain (loop) parallelism and OpenMP*.
2. Optimizing memory access and exposing SIMD.
3. Tweaking to deal with SIMD constraints.
4. Offloading kernels and partitioning resources.

The last item is merely coprocessor-specific. The only platform-specific bits pertain to SIMD width and alignment restrictions.

On the other hand, CUDA* requires >6500 lines of code that only runs on one platform.

On the subject of open standards…

- Porting ARMCI to new platforms is the overwhelming portability bottleneck for NWChem.
- Vendor-specific network APIs are not always supported and can be moving targets for ARMCI developers.
- MPI-3 is a popular, portable interface for HPC networks. Well over 90% of the machines on the Top500 support it.
- A native ARMCI port is not available on the #1 machine in the world but MPI-3 is (MPICH-GLEX).

ARMCI-MPI

Goal: Implement the ARMCI interface on top of MPI RMA (one-sided) features.

Challenge: MPI-2 one-sided was not a good match for ARMCI. In particular, atomics were missing.

Nonetheless, Jim Dinan and coworkers implemented ARMCI over MPI-2, which was called ARMCI-MPI.

We (Jim, Pavan, Jeff, etc.) helped fix one-sided in MPI-3.

http://wiki.mpich.org/armci-mpi/index.php/Main_Page

ARMCI-MPI Design

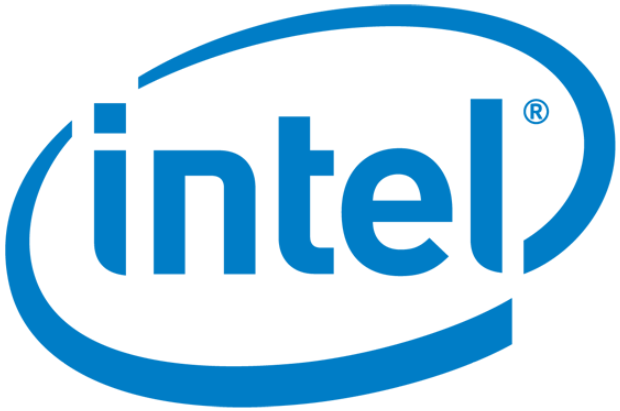
- ARMCI memory management routines mapping to MPI windows, which is tedious.
- ARMCI communication routines mapped to peer routines in MPI-3, including atomics (RMW, CAS).
- Noncontiguous data uses MPI datatypes or multiple injection.
- MPI-3 allows us to match ARMCI's location consistency (ordering) and direct local access semantics.
- Nonblocking support impossible using MPI-2, now well-optimized in ARMCI-MPI using MPI-3.

Portability, Scalability and Stability

- NWChem scaled to at least 38K processes of NERSC Edison.
- GTFOCK (IPDPS13 Best Paper) scaled to more than 8000 nodes of Tianhe-2.
- Running near the memory limit on InfiniBand*, which is not possible without ARMCI-MPI.
- Zero ARMCI-related failures except related to InfiniBand*, all of which are trivially solved with MVAPICH2* environment variables.
- Supports late-model MPICH* and derivatives (MVAPICH2*, Cray* MPI, etc.) as well as Open MPI*.

Summary

- NWChem supports Intel® Xeon PHI™ processor in HPC-oriented CCSD(T) module using portable directives. Work in progress for other methods.
- Intel® Xeon PHI™ processor optimizations pay off on Xeon and other CPU platforms: ditch the offload, keep the OpenMP* and SIMD.
- ARMCI-MPI3 provides portability, stability and scalability on essentially all HPC platforms.


intel