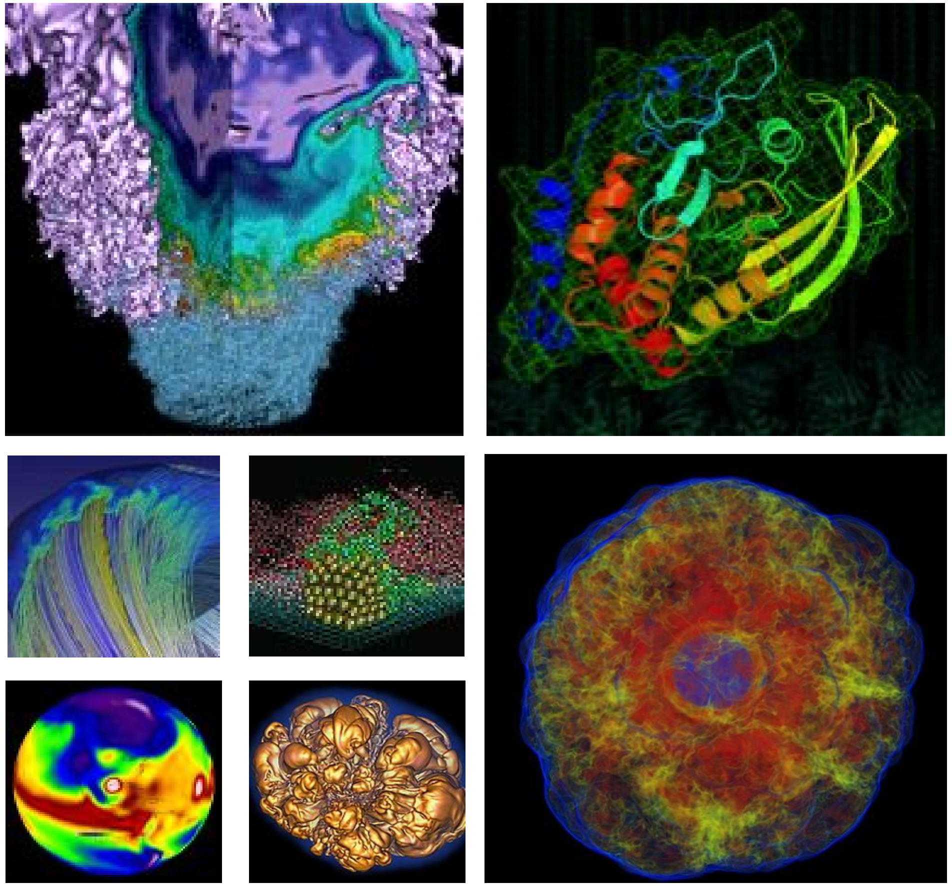
Optimizing Excited-State Electronic-Structure Codes for Intel Knights Landing



Jack Deslippe

Application Performance Group
NERSC

NERSC

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

What is GW

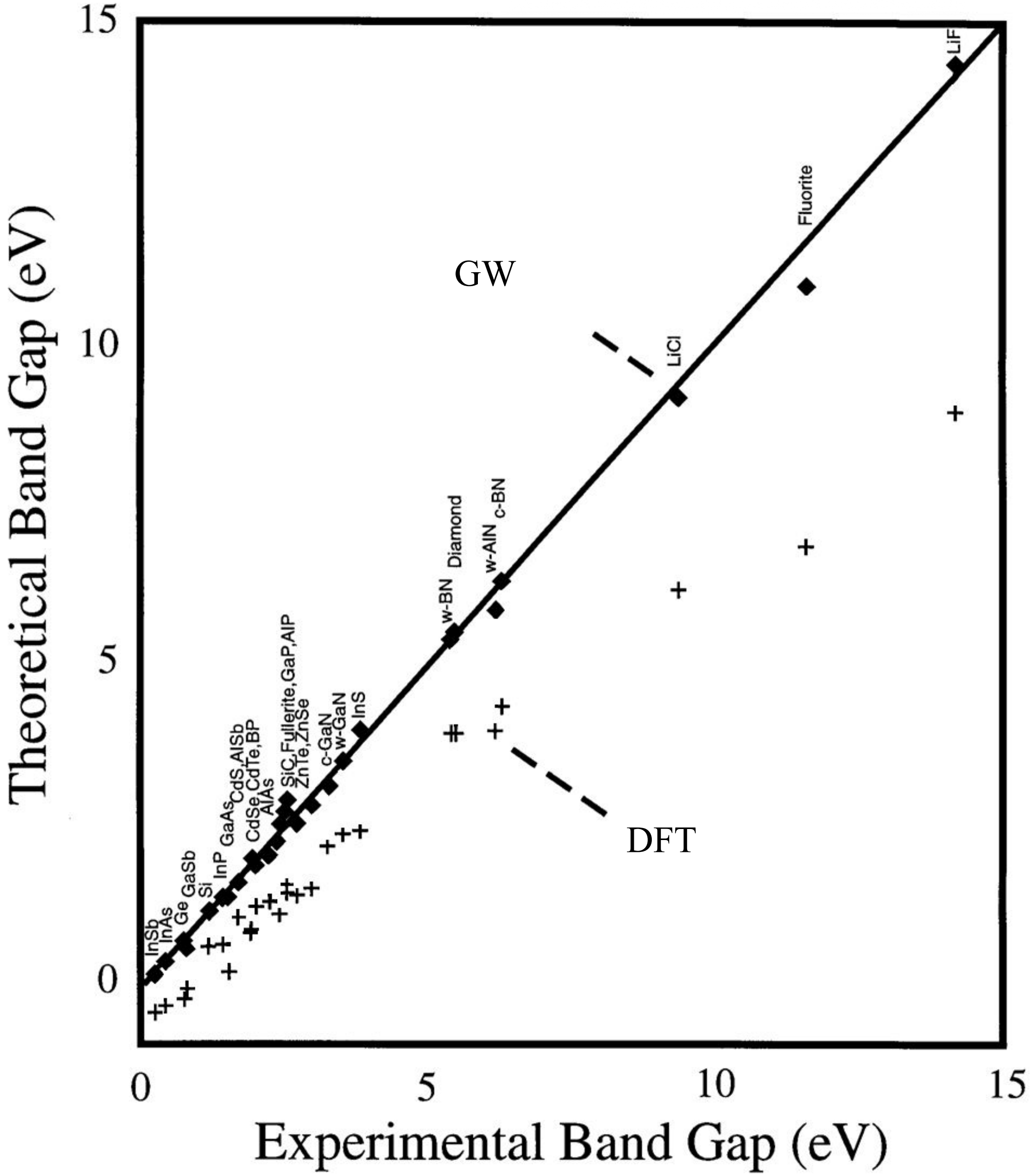
The “GW” method is an accurate approach for simulate the “excited state” properties of materials. Examples:

- What happens when you add or remove an electron from a system
- How do electrons behave when you apply a voltage
- How does the system respond to light or x-rays

GW is complementary to the widely used density functionally theory methods (DFT) which predict ground state properties of materials - i.e. the properties of the system associated with all particles in the lowest energy configuration.

Materials:

InSb, InAs
Ge
GaSb
Si
InP
GaAs
CdS
AlSb, AlAs
CdSe, CdTe
BP
SiC
C_{60}
GaP
AlP
ZnTe, ZnSe
c-GaN, w-GaN
InS
w-BN, c-BN
diamond
w-AlN
LiCl
Fluorite
LiF

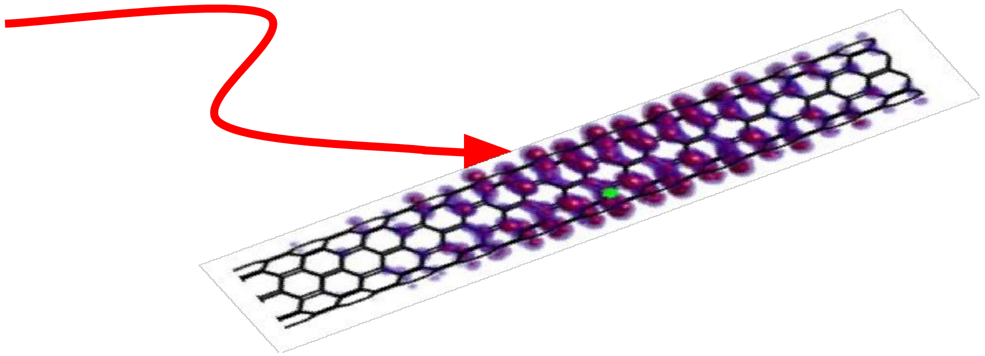


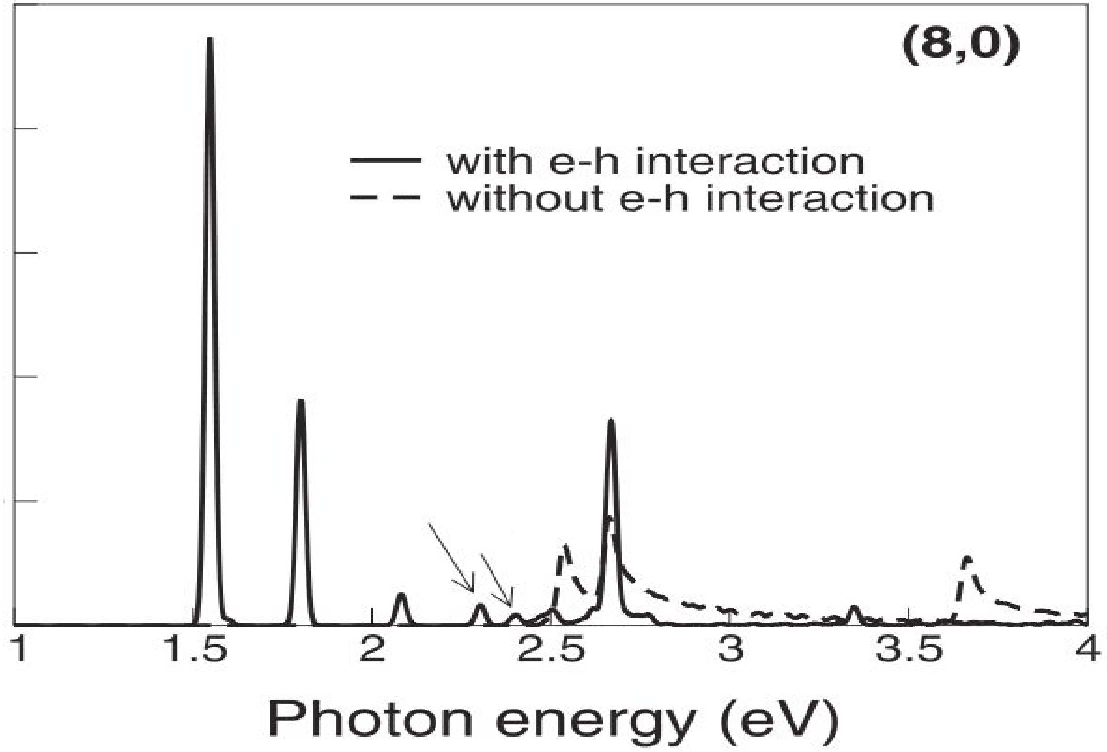
What is GW

Many-body effects extremely important in **Excited-State properties** of Complex Materials.

Accurately describes properties important for:

- Photovoltaics
- LEDs
- Junctions / Interfaces
- Defect Energy Levels
- ….





*C.D. Spataru, S. Ismail-Beigi, L.X. Benedict, S.G. Louie. PRL 077402 (2004)

*J. Deslippe, C.D. Spataru, D. Prendergast, S.G Louie. Nano Letters. 7 1626 (2007)

Original BerkeleyGW Code - Massively Parallel. But MPI Only

Original code was MPI only. But, unlike DFT there are many layers over which you can exploit parallelism:

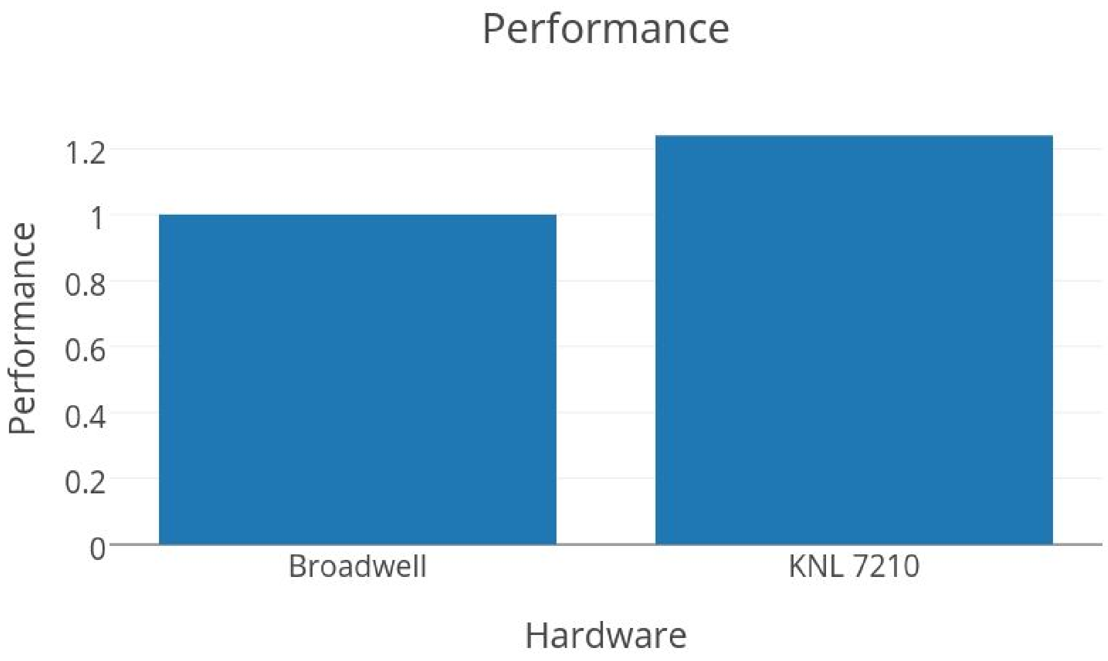
$$\chi_{\mathbf{G}\mathbf{G}'}(E) = \sum_{n}^{\mathrm{occ}} \sum_{n'}^{\mathrm{emp}} M^{*}_{nn'}(\mathbf{G}) M_{nn'}(\mathbf{G}') \frac{1}{E_n - E_{n'} - E}$$

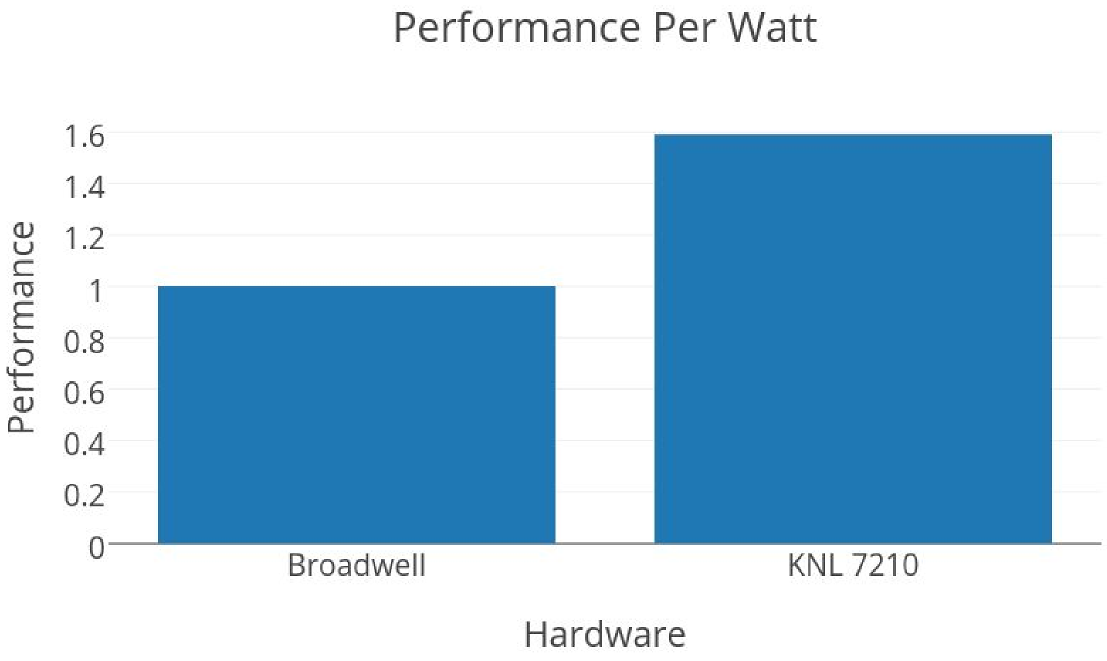
Band-pairs: (n,n') Millions
Energies: E Tens-Hundreds
Plane-Wave Basis Elements: $(\mathbf{G},\mathbf{G}')$ - Millions

Much better suited exploiting levels of parallelism on HPC system like Cori. ~10,000 Nodes, 250 Threads per Node, 8 Wide Vectors.

Don’t Bury the Lead

BerkeleyGW Running Well on KNL.





Data From Sigma Benzene Runs on Single Node - Excluding IO.

Computational Bottlenecks

3 Main Computational Bottlenecks. Use roughly equal time for 500 atom systems.

A. Compute transition probabilities (matrix-elements) for electrons from occupied to empty orbitals

B. Sum matrix-elements to form the overall material response function (polarizability)

C. Calculate the interacting electron energy from the polarizability

Kernel A

Compute the transition probability between two electron states (orbitals):

$$M_{nn'}(\mathbf{G}) = \langle n| \, e^{i\mathbf{G}\cdot\mathbf{r}} \, |n'\rangle$$

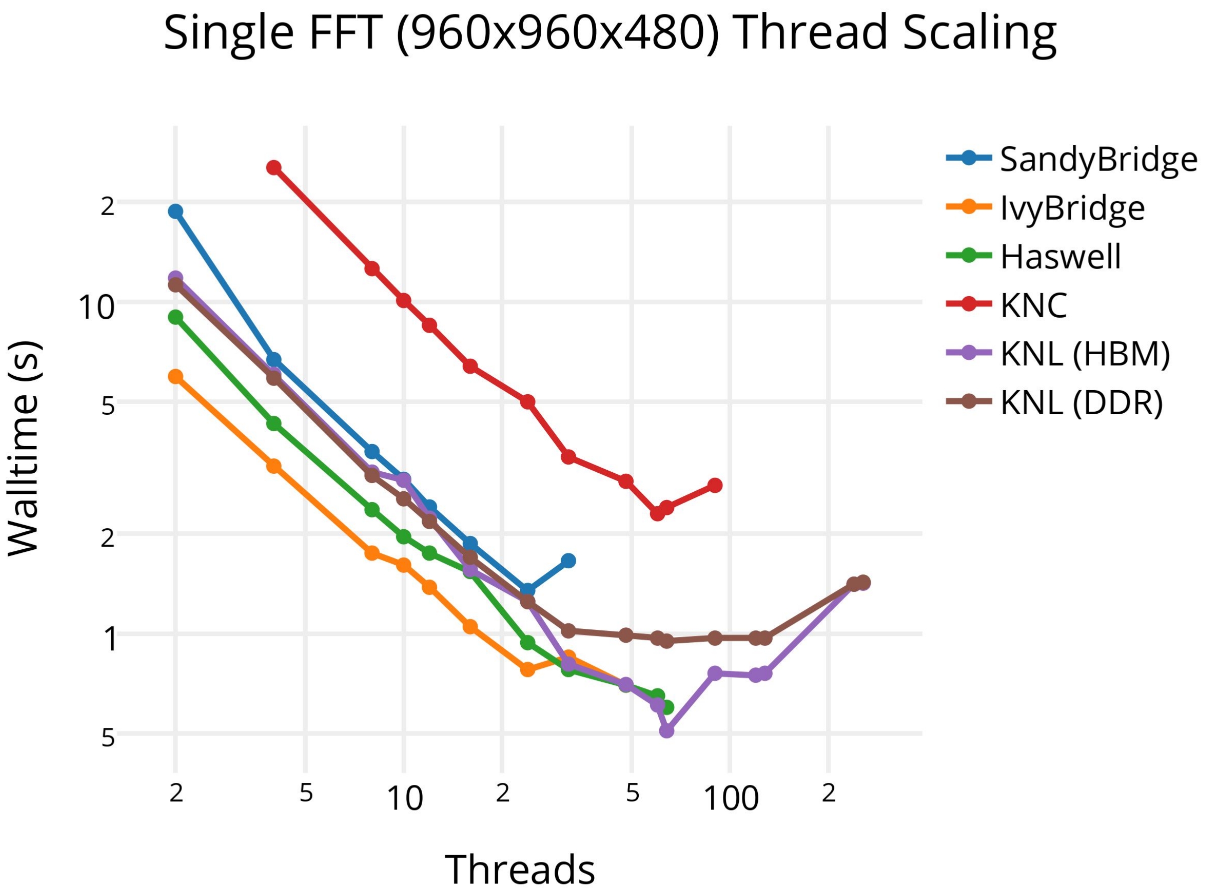
Typically computed by FFT:

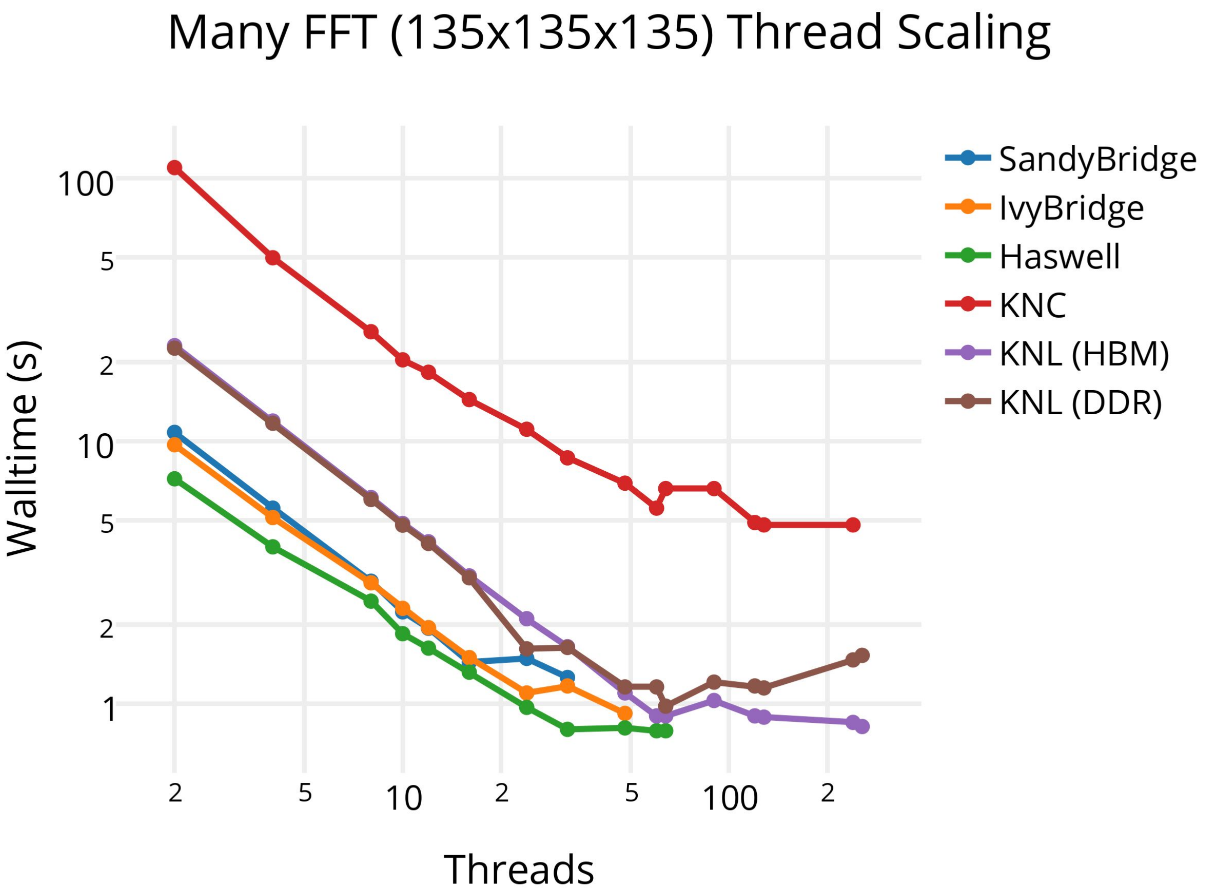
$$M_{nn'}(\{\mathbf{G}\}) = \mathrm{FFT}^{-1}\left(\phi_n(\mathbf{r})\phi^*_{n'}(\mathbf{r})\right)$$

Must be done for all pairs of orbitals n, n'. Since the number of orbitals considered is proportional to number of atoms in calculation. The complexity of this step is O(N^{3}logN).

We distribute (n, n') via MPI and call threaded 3D FFT libraries (MKL) in the app. (Note, significantly more parallelism than local DFT)

Kernel A Performance (MKL)





Related Improvements in Quantum ESPRESSO

When performing Hybrid Functional calculations within DFT, like within GW, you need to perform an FFT for each pair of orbitals.

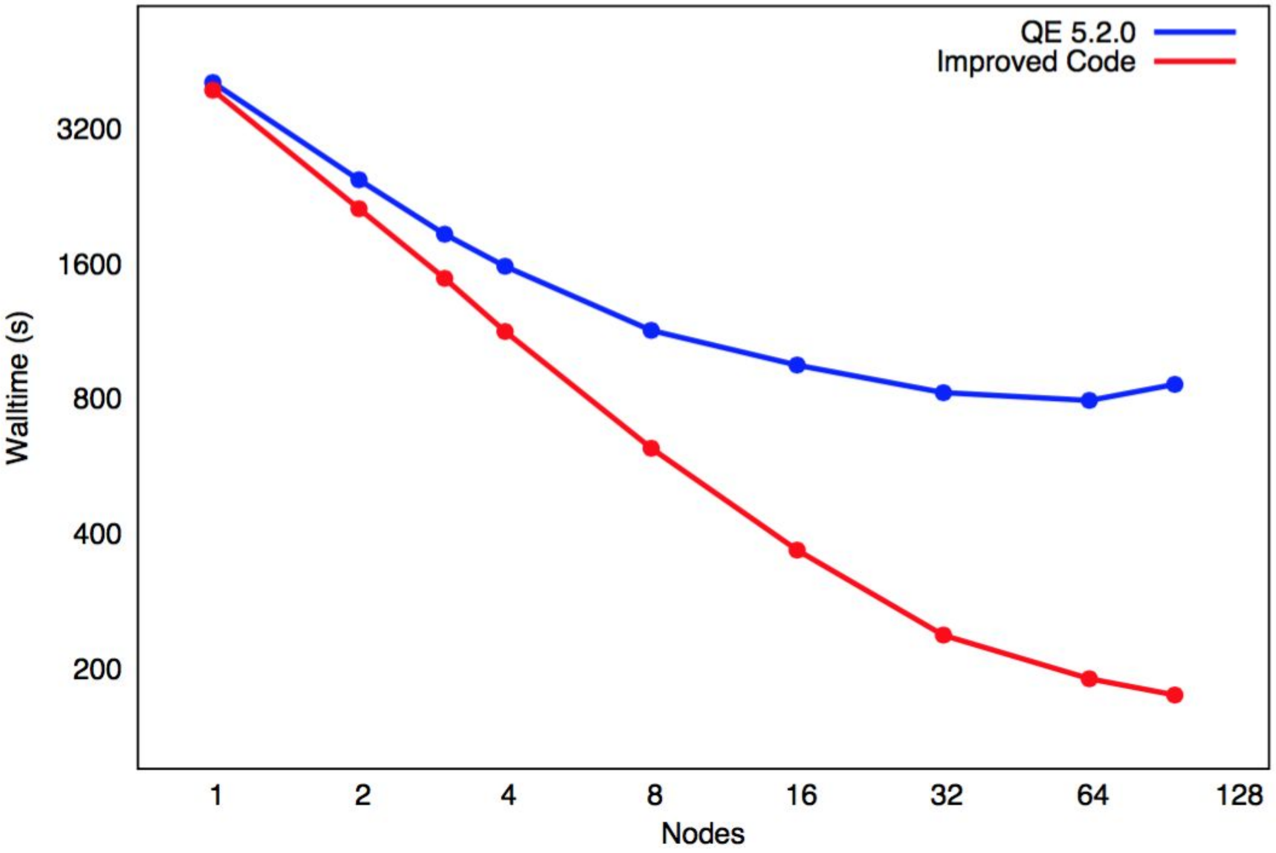
By default, code parallelizes each individual FFT.

We improve code to:

1. Parallelize over pairs of orbitals, before parallelizing individual FFTs
2. Allow simultaneous parallelism over orbitals for Hybrid calculation and other parameters for local calculation.

This leads to less communication, more work (complete 3D-FFT) on-node to parallelize.

Optimized QE on NERSC Edison



Kernel B

We want to compute the electronic polarizability of the system:

$$\chi_{\mathbf{G}\mathbf{G}'}(E) = \sum_{n}^{\mathrm{occ}} \sum_{n'}^{\mathrm{emp}} M^*_{nn'}(\mathbf{G}) M_{nn'}(\mathbf{G}') \frac{1}{E_n - E_{n'} - E}$$

We can write this as a number of ZGEMM operations (one for each E):

$$\chi_{\mathbf{G}\mathbf{G}'}(E) = \mathbf{M}^*(\mathbf{G}, (n, n'), E) \cdot \mathbf{M}^{\mathrm{T}}(\mathbf{G}', (n, n'), E)$$

Where $\mathbf{M}$ is:

$$\mathbf{M}(\mathbf{G}, (n, n'), E) = M_{nn'}(\mathbf{G}) \cdot \frac{1}{\sqrt{E_n - E_{n'} - E}}$$

There are two steps. First, constructing $\mathbf{M}$, and second, performing the complex double-precision ZGEMM.

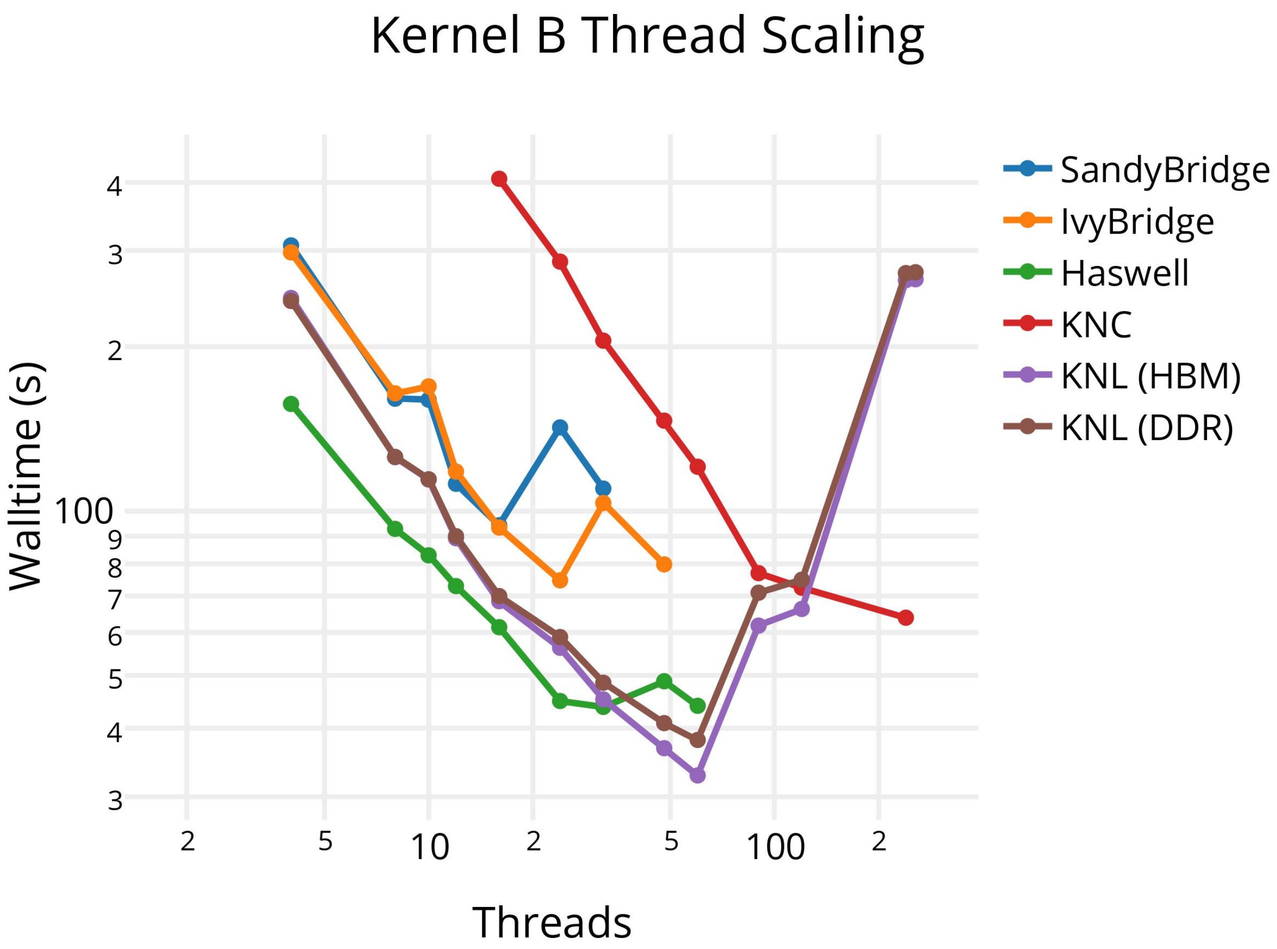
The complexity of this step is $O(N^4)$.

Kernel B Performance

Little difference between MCDRAM and HBM performance - Only in the initialization/prep stage.

KNL overall performing 20% faster than Haswell.

No advantage of Hyper-Threading on Xeon or KNL seen.



Kernel C

Compute the electronic energy as:

$$\Sigma_n = \sum_{n'} \sum_{\mathbf{GG}'} M^*_{n'n}(-\mathbf{G}) M_{n'n}(-\mathbf{G}') \frac{\Omega^2_{\mathbf{GG}'}}{\tilde{\omega}_{\mathbf{GG}'}\left(E - E_{n'} - \tilde{\omega}_{\mathbf{GG}'}\right)} v(\mathbf{G}')$$

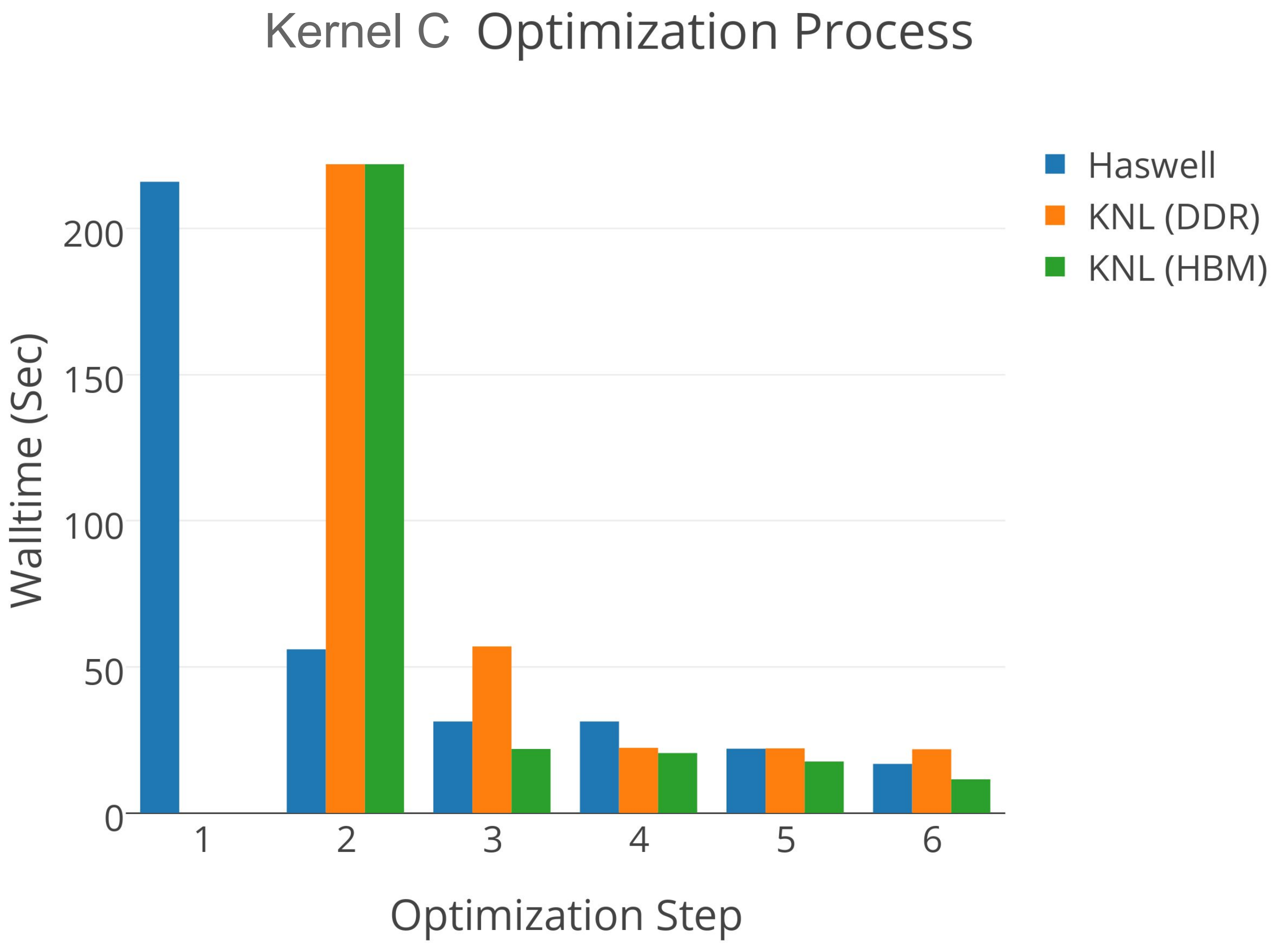
Here $\tilde{\omega}$ and Ω are complex double precision arrays derived from the polarizability. This is a tensor-contraction, matrix reduction type operation - performed by hand tuned code.

The complexity of this step for all n is O(N^4).

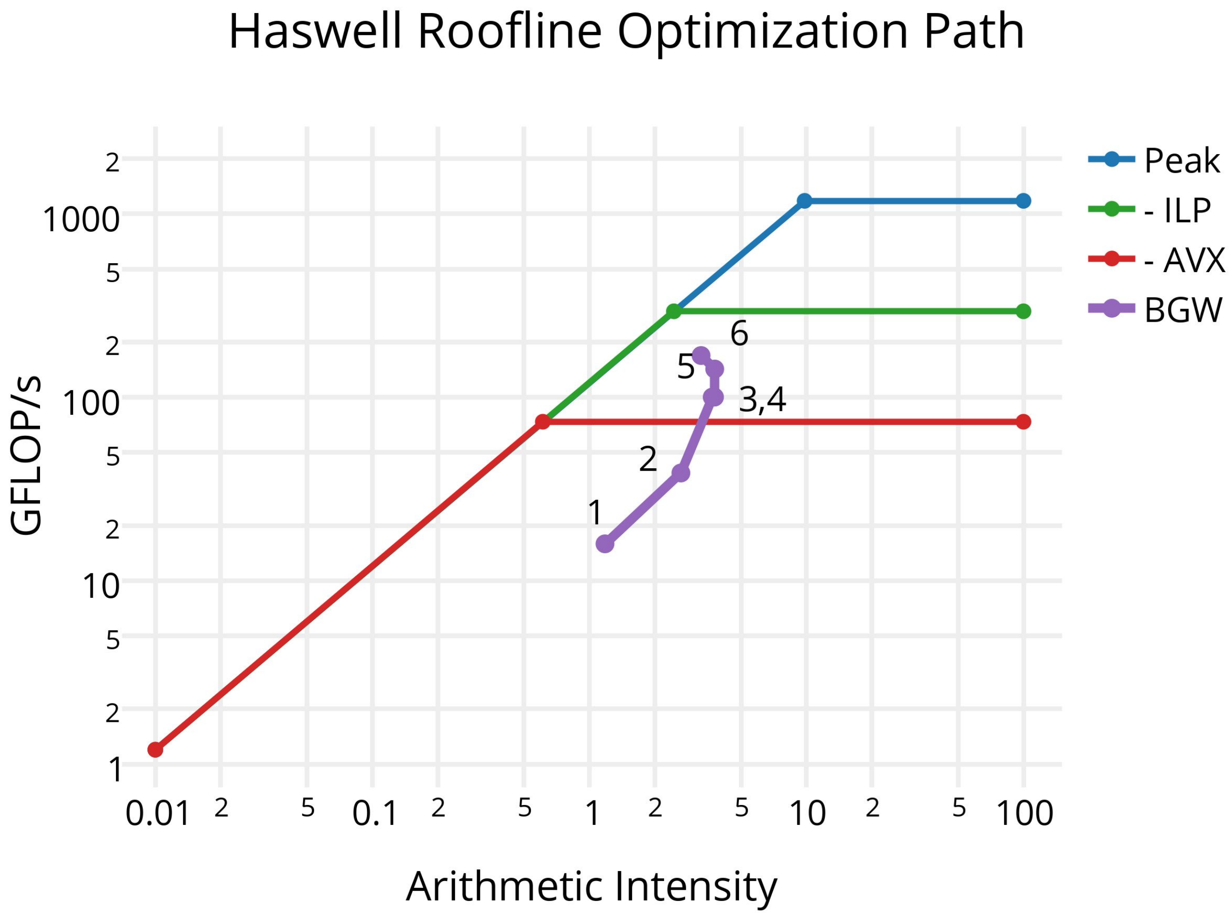
Kernel C Optimization

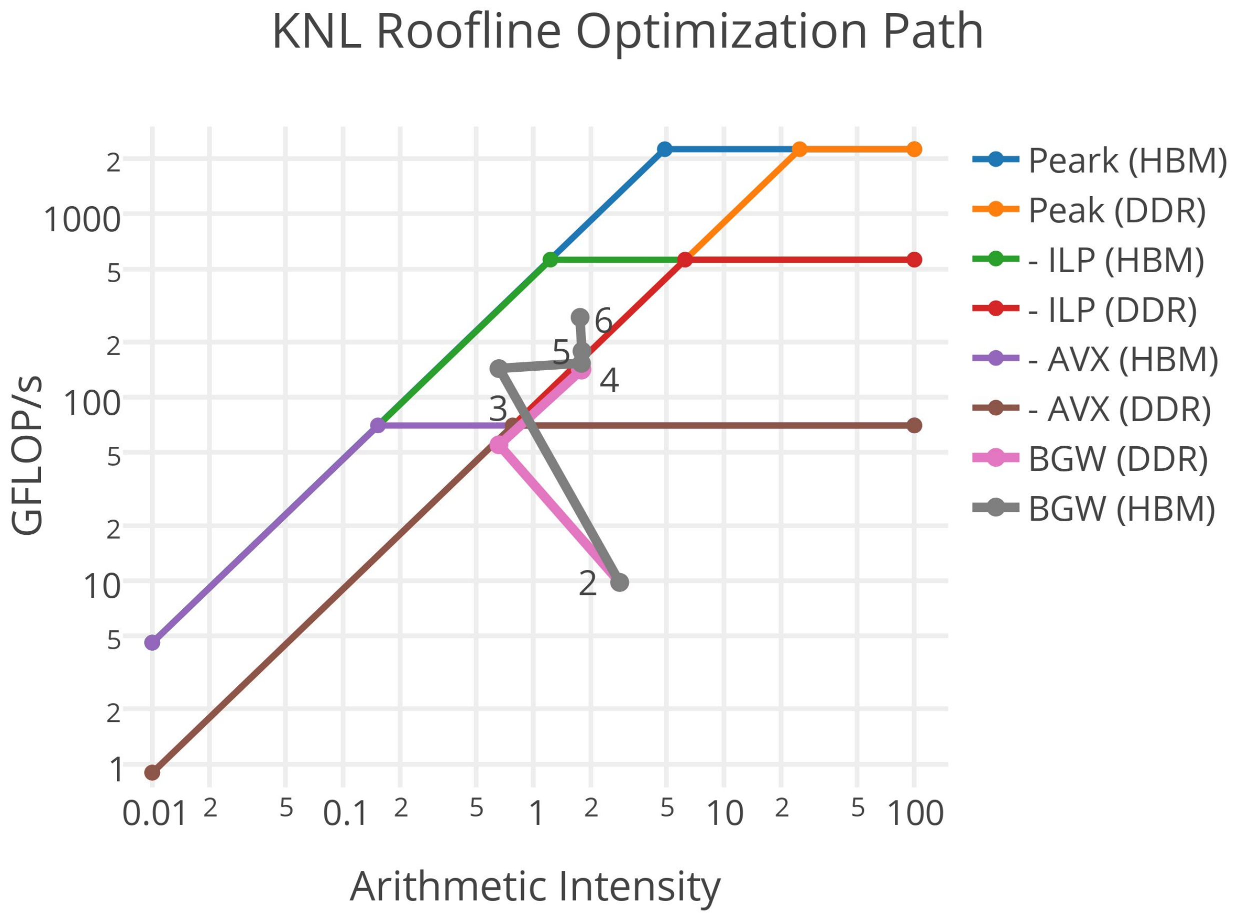
Optimization process for Kernel-C (Sigma code):

1. Refactor (3 Loops for MPI, OpenMP, Vectors)
2. Add OpenMP
3. Initial Vectorization (loop reordering, conditional removal)
4. Cache-Blocking
5. Improved Vectorization
6. Hyper-threading

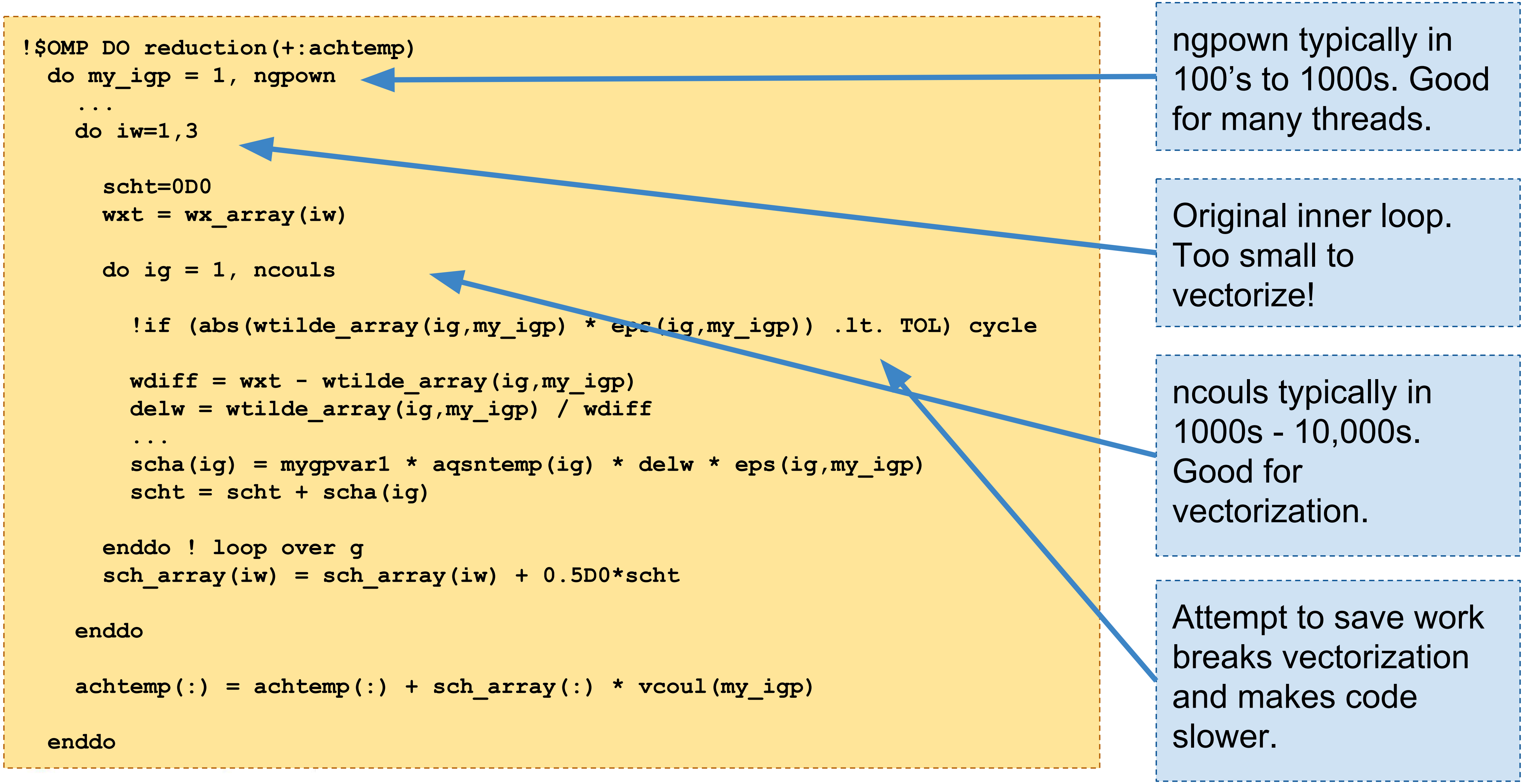


Kernel C Optimization





Steps 2-3



```
!$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!

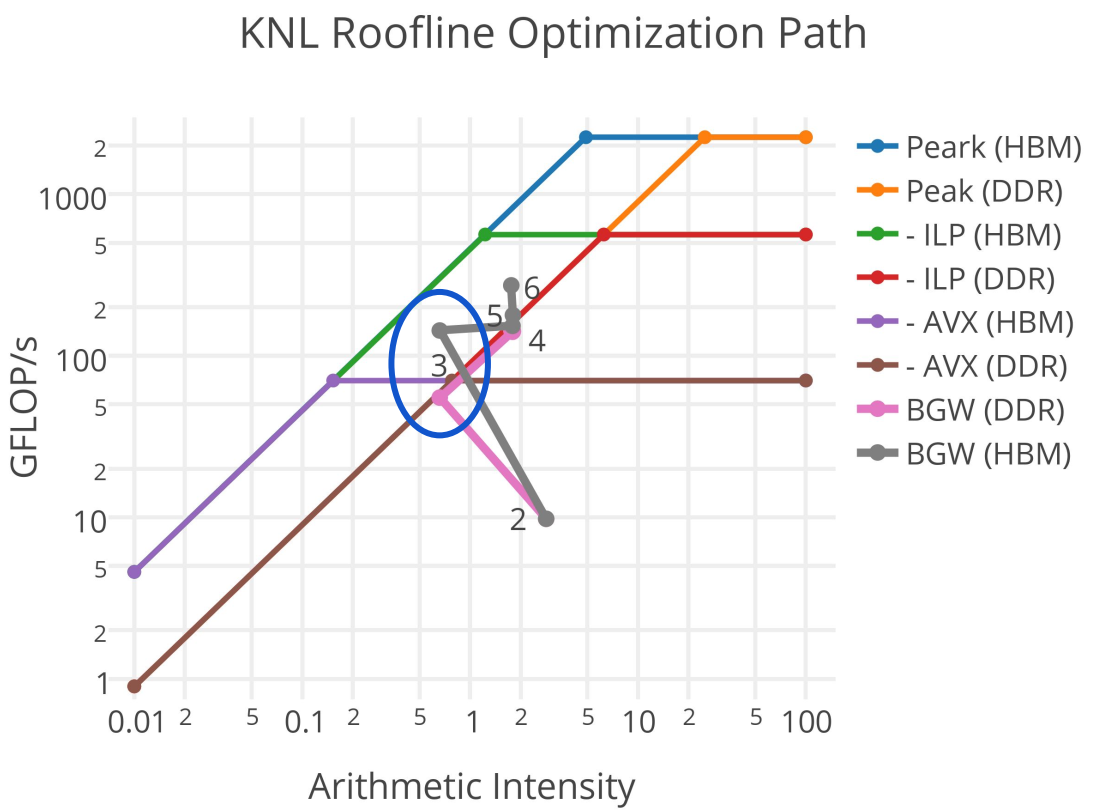
ncouls typically in 1000s - 10,000s. Good for vectorization.

Attempt to save work breaks vectorization and makes code slower.

Data Locality Worse Between Step 2 - 3 on KNL/KNC

Optimization process for Kernel-C (Sigma code):

1. Refactor (3 Loops for MPI, OpenMP, Vectors)
2. Add OpenMP
3. Initial Vectorization (loop reordering, conditional removal)
4. Cache-Blocking
5. Improved Vectorization
6. Hyper-threading



KNL Roofline Optimization Path

The loss of L3 on MIC makes locality more important.

Why AI Lower on KNL?

```
!$OMP DO
do my_igp = 1, ngpown
      do iw = 1 , 3
         do ig = 1, igmax
            load wtilde_array(ig,my_igp) 819 MB, 512KB per row
            load aqsntemp(ig,n1) 256 MB, 512KB per row
            load I_eps_array(ig,my_igp) 819 MB, 512KB per row
            do work (including divide)
```

Required Cache size to reuse 3 times:

1536 KB

L2 on KNL is 512 KB per core
L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core

Without blocking we spill out of L2 on KNC and Haswell. But, Haswell has L3 to catch us.

Why AI Lower on KNL?

```
!$OMP DO
do my_igp = 1, ngpown
  do igbeg = 1, igmax, igblk
    do iw = 1 , 3
      do ig = igbeg, min(igbeg + igblk,igmax)
        load wtilde_array(ig,my_igp) 819 MB, 512KB per row
        load aqsntemp(ig,n1) 256 MB, 512KB per row
        load I_eps_array(ig,my_igp) 819 MB, 512KB per row
        do work (including divide)
```

Required Cache size to reuse 3 times:

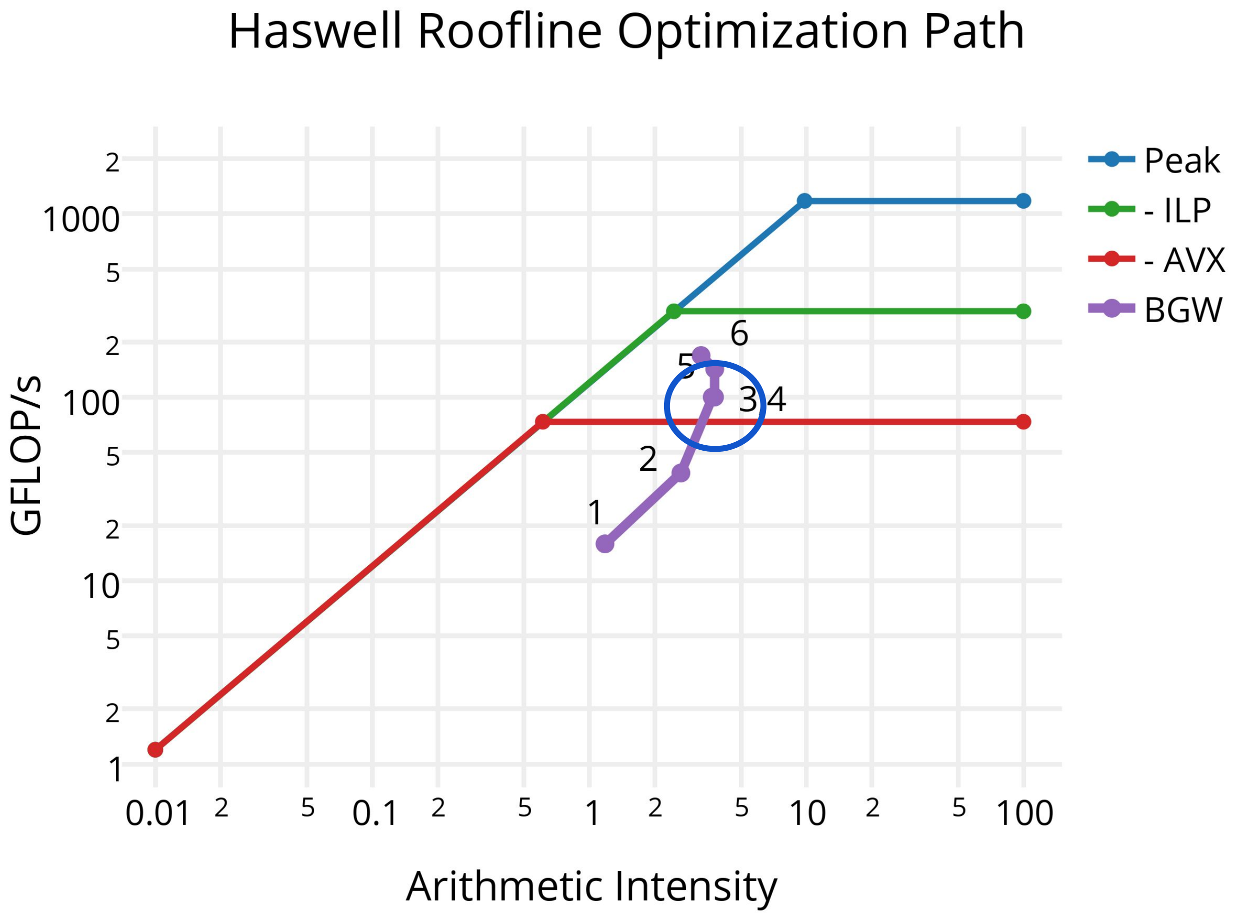
1536 KB

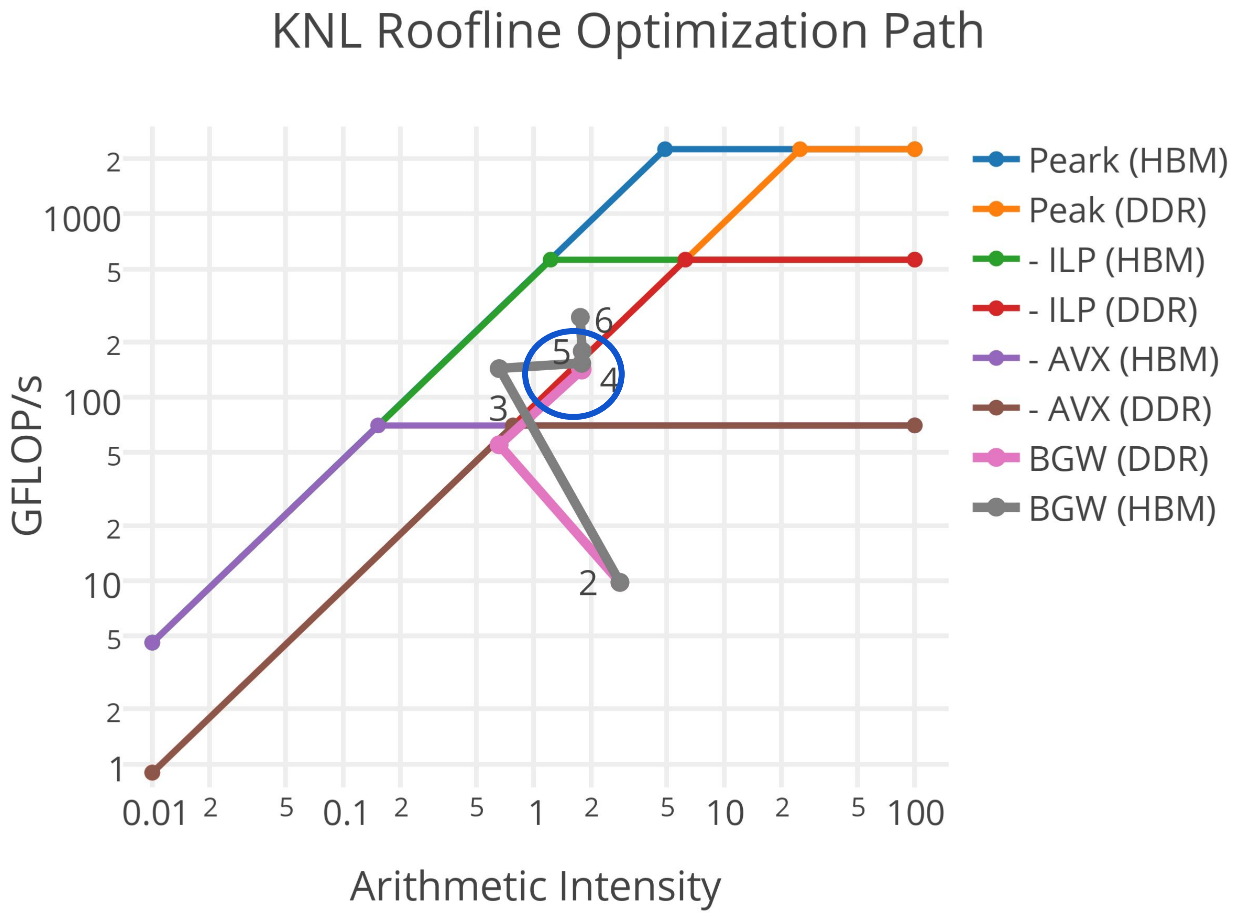
L2 on KNL is 512 KB per core
L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core

Without blocking we spill out of L2 on KNC and Haswell. But, Haswell has L3 to catch us.

Kernel C Optimization

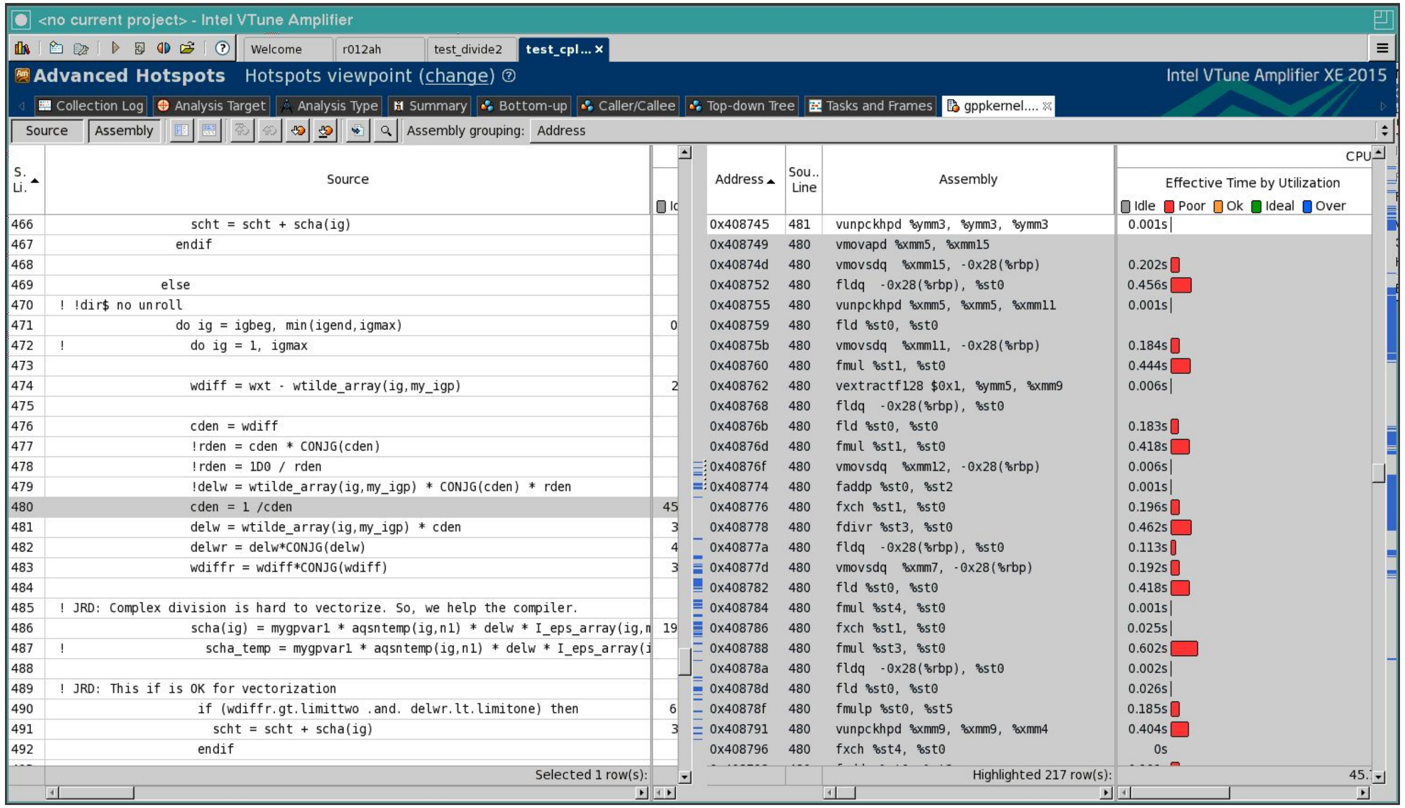




SDE + Vtune, Why Complex Divides so Slow?

Found ~ 2x Instruction reduction from AVX to AVX512

However, found significant x87 instructions from 1/complex_number



```
S.
Li.  Source                                                                       Id
466                 scht = scht + scha(ig)
467               endif
468
469             else
470   ! !dir$ no unroll
471               do ig = igbeg, min(igend,igmax)                                 0
472   !             do ig = 1, igmax
473
474                 wdiff = wxt - wtilde_array(ig,my_igp)                         2
475
476                 cden = wdiff
477                 !rden = cden * CONJG(cden)
478                 !rden = 1D0 / rden
479                 !delw = wtilde_array(ig,my_igp) * CONJG(cden) * rden
480                 cden = 1 /cden                                               45
481                 delw = wtilde_array(ig,my_igp) * cden                         3
482                 delwr = delw*CONJG(delw)                                      4
483                 wdiffr = wdiff*CONJG(wdiff)                                   3
484
485   ! JRD: Complex division is hard to vectorize. So, we help the compiler.
486                 scha(ig) = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,m 19
487   !               scha_temp = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(i
488
489   ! JRD: This if is OK for vectorization
490                  if (wdiffr.gt.limittwo .and. delwr.lt.limitone) then         6
491                    scht = scht + scha(ig)                                     3
492                  endif
                                                          Selected 1 row(s):
```

Address ▲	Sou.. Line	Assembly	CPU Effective Time by Utilization ▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over
0x408745	481	vunpckhpd %ymm3, %ymm3, %ymm3	0.001s
0x408749	480	vmovapd %xmm5, %xmm15	
0x40874d	480	vmovsdq %xmm15, -0x28(%rbp)	0.202s
0x408752	480	fldq -0x28(%rbp), %st0	0.456s
0x408755	480	vunpckhpd %xmm5, %xmm5, %xmm11	0.001s
0x408759	480	fld %st0, %st0	
0x40875b	480	vmovsdq %xmm11, -0x28(%rbp)	0.184s
0x408760	480	fmul %st1, %st0	0.444s
0x408762	480	vextractf128 $0x1, %ymm5, %xmm9	0.006s
0x408768	480	fldq -0x28(%rbp), %st0	
0x40876b	480	fld %st0, %st0	0.183s
0x40876d	480	fmul %st1, %st0	0.418s
0x40876f	480	vmovsdq %xmm12, -0x28(%rbp)	0.006s
0x408774	480	faddp %st0, %st2	0.001s
0x408776	480	fxch %st1, %st0	0.196s
0x408778	480	fdivr %st3, %st0	0.462s
0x40877a	480	fldq -0x28(%rbp), %st0	0.113s
0x40877d	480	vmovsdq %xmm7, -0x28(%rbp)	0.192s
0x408782	480	fld %st0, %st0	0.418s
0x408784	480	fmul %st4, %st0	0.001s
0x408786	480	fxch %st1, %st0	0.025s
0x408788	480	fmul %st3, %st0	0.602s
0x40878a	480	fldq -0x28(%rbp), %st0	0.002s
0x40878d	480	fld %st0, %st0	0.026s
0x40878f	480	fmulp %st0, %st5	0.185s
0x408791	480	vunpckhpd %xmm9, %xmm9, %xmm4	0.404s
0x408796	480	fxch %st4, %st0	0s
		Highlighted 217 row(s):	45.1

Can significantly speed up by

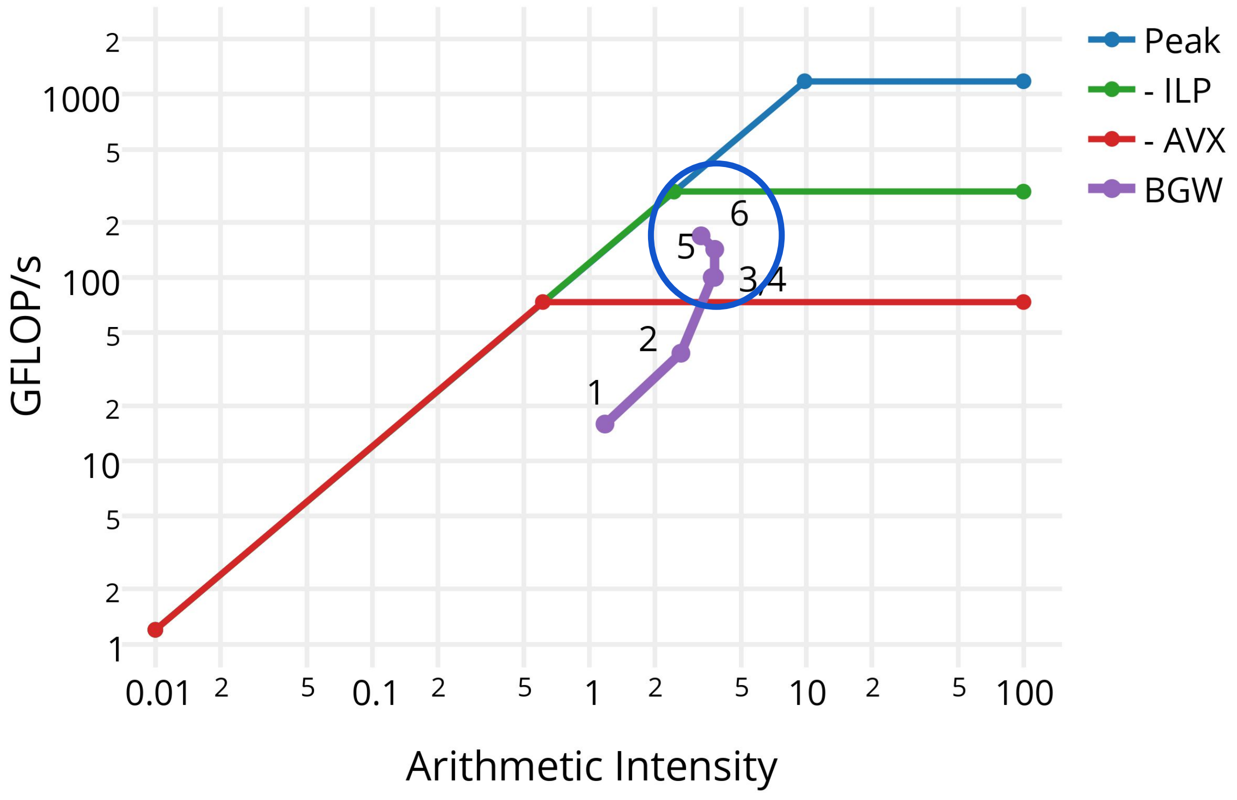
a) Doing complex divide manually

Or

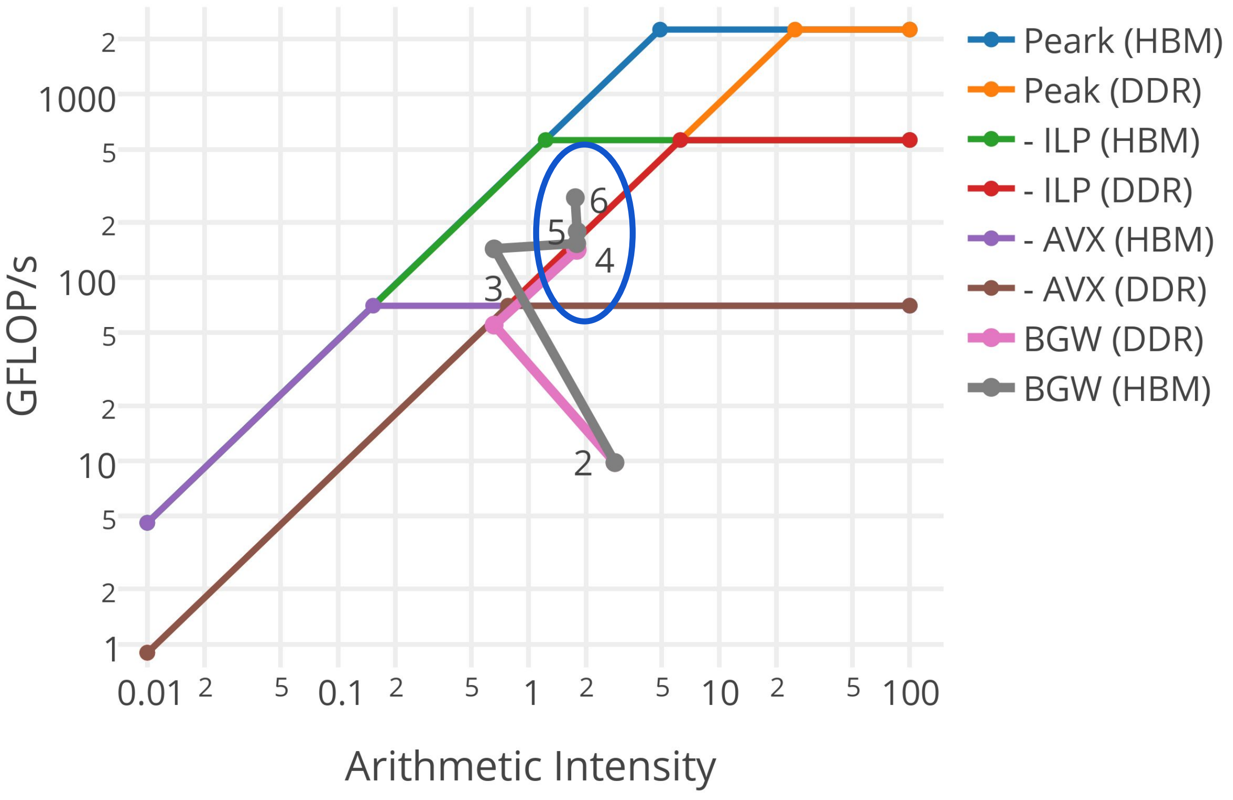
b) Using -fp-model fast=2

Additional Speedups from Hyperthreading

Haswell Roofline Optimization Path



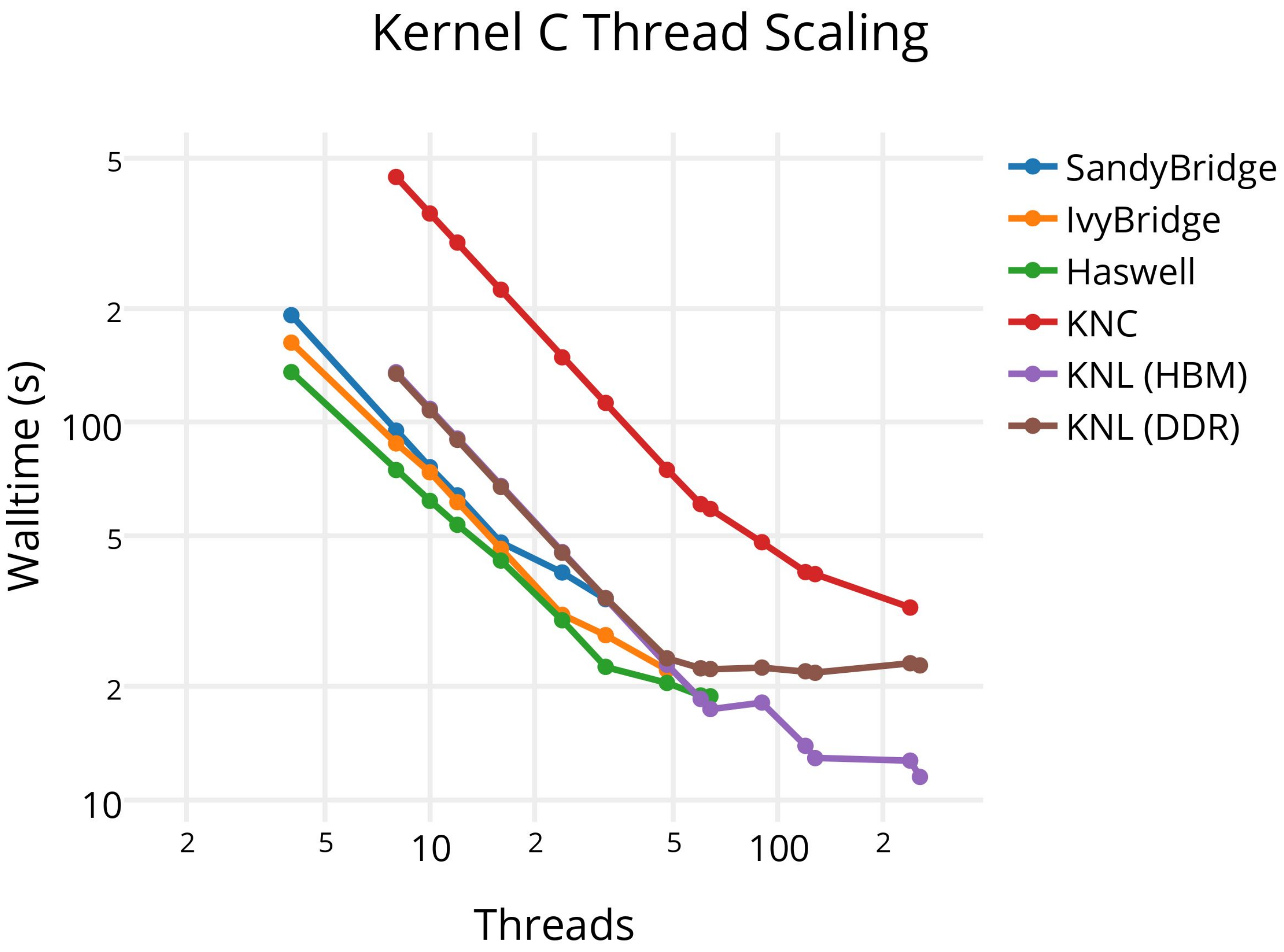
KNL Roofline Optimization Path



Kernel C Thread Scaling

KNL DDR performance saturates at around 50 threads, becomes memory bandwidth limited.

KNL MCDRAM performance beats dual socket Haswell by 63%.



Conclusions

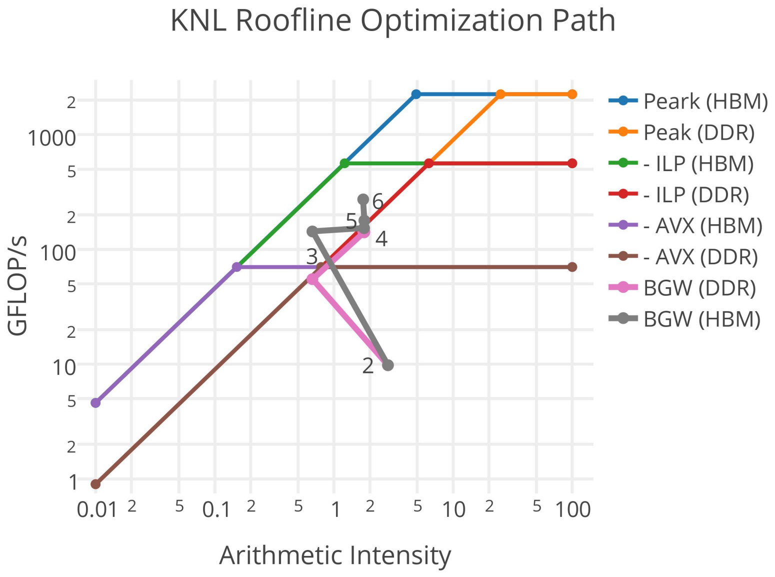
Kernel A - FFTs show moderate speedups over dual-socket Haswell. Kernel B - ZGEMM and stream like operations show big speedups over Haswell

Kernel C - Hand tuned matrix reduction operations show 60% speedup over haswell.

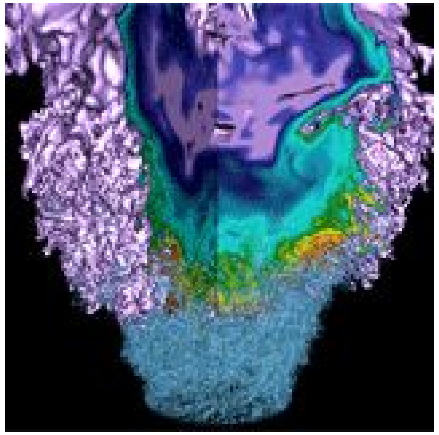
For algorithms with AI near roofline APEX (1-10), there is a rich optimization space that needs to be explored. Need all of:

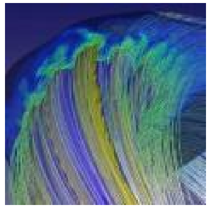
- Thread scaling
- Vectorization
- Cache-Reuse
- Effective use of MCDRAM

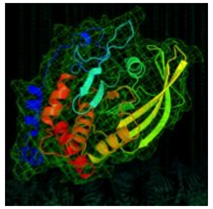
Targeting Many-Core greatly helps performance back on Xeon.

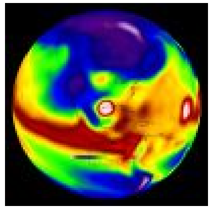

KNL Roofline Optimization Path

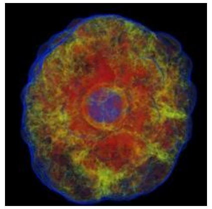
The End (Extra Slides)

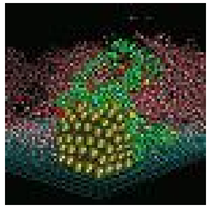












NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

GW Reputation (Motivation Behind BerkeleyGW)

$$[E_{n\mathbf{k}} - H_0(\mathbf{r}) - V_H(\mathbf{r})]\,\psi_{n\mathbf{k}}(\mathbf{r}) - \int \Sigma(\mathbf{r},\mathbf{r}',E_{n,\mathbf{k}})\psi_{n\mathbf{k}}(\mathbf{r}')d\mathbf{r}' = 0$$

The Good:

Quantitatively accurate for quasiparticle properties in a wide variety of systems.

Accurately describes dielectric screening important in excited state properties.

The Bad:

Prohibitively slow for large systems. Usually thought to cost orders of magnitude more time that DFT.

Memory intensive and scales badly. Exhausted by storage of the dielectric matrix and wavefunctions. Limited ~50 atoms.

GW Reputation (Motivation Behind BerkeleyGW)

$$\left[E_{n\mathbf{k}} - H_0(\mathbf{r}) - V_H(\mathbf{r})\right]\psi_{n\mathbf{k}}(\mathbf{r}) - \int \Sigma(\mathbf{r},\mathbf{r}',E_{n,\mathbf{k}})\psi_{n\mathbf{k}}(\mathbf{r}')d\mathbf{r}' = 0$$

The Good:

Quantitatively accurate for quasiparticle properties in a wide variety of systems.

Accurately describes dielectric screening important in excited state properties.

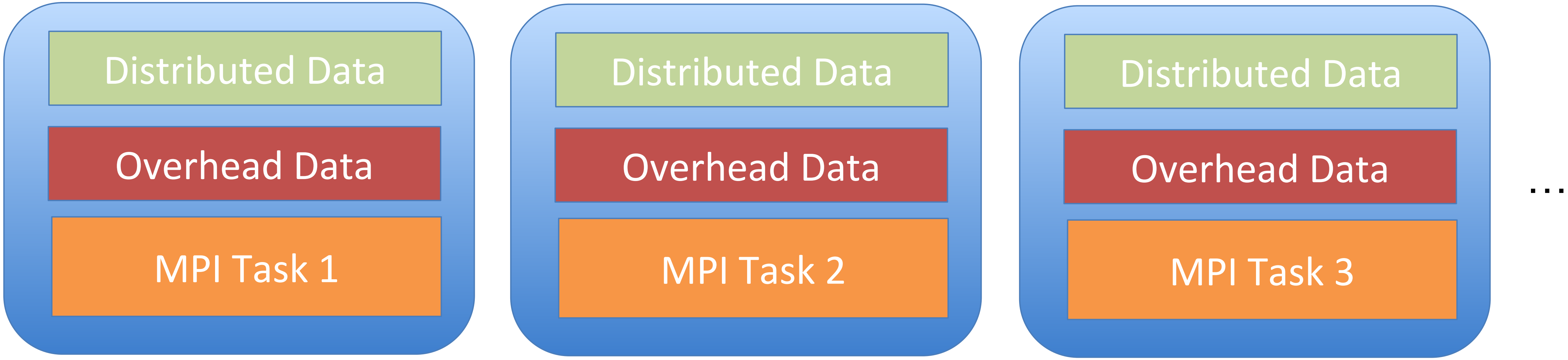
The Bad:

Prohibitively slow for large systems. Usually thought to cost orders of magnitude more time that DFT.

Memory intensive and scales badly. Exhausted by storage of the dielectric matrix and wavefunctions. Limited ~50 atoms.

BerkeleyGW Towards Many-Core

- ★ In an MPI GW implementation, in practice, to avoid communication, data is duplicated and **each MPI task has a memory overhead**.
- ★ Users sometimes forced to use 1 of 24 available cores, in order to provide MPI tasks with enough memory. **90% of the computing capability is lost**.


Distributed Data
Overhead Data
MPI Task 1
Distributed Data
Overhead Data
MPI Task 2
Distributed Data
Overhead Data
MPI Task 3
…