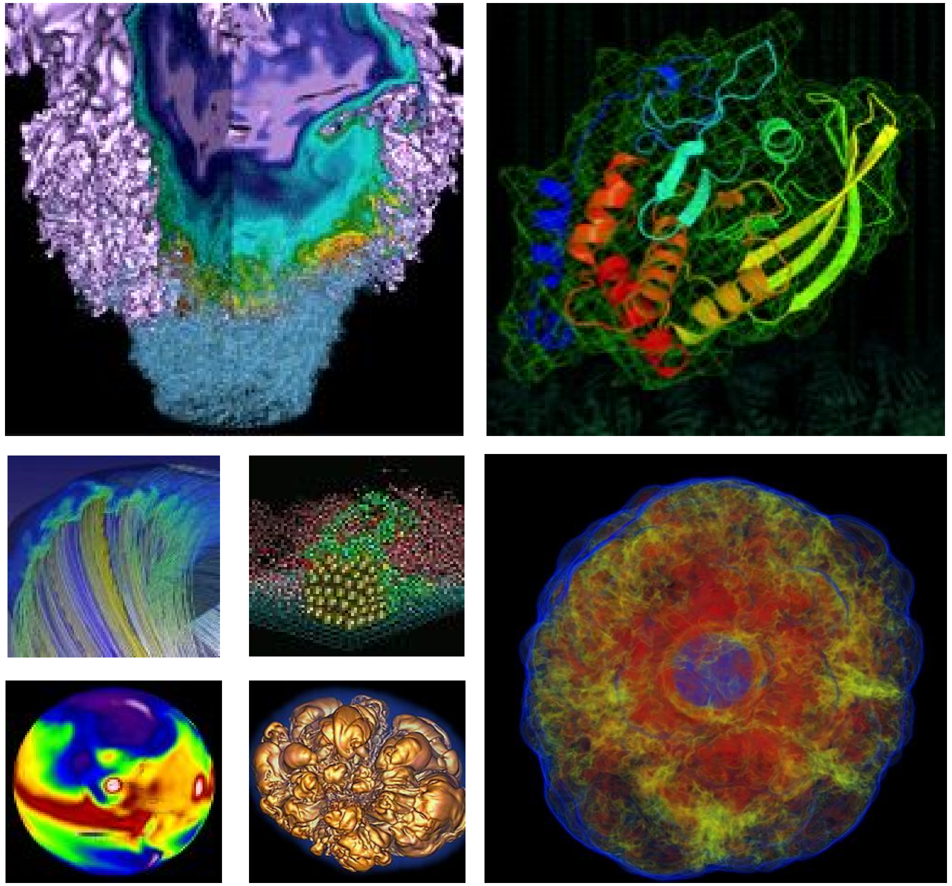
Optimizing Applications: an Ant Farm Approach

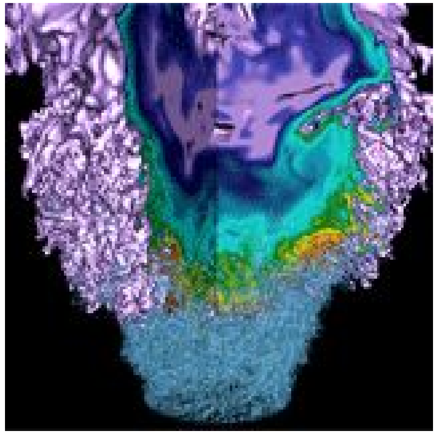


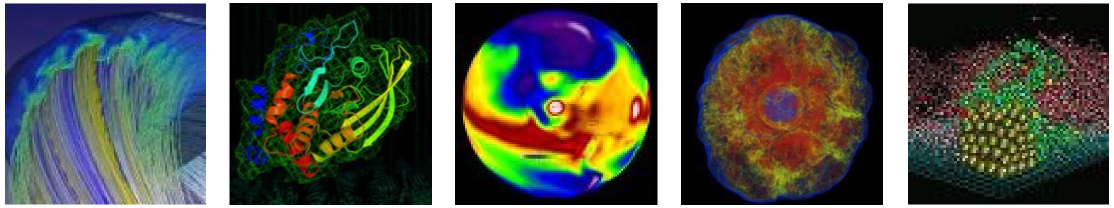
Jack Deslippe
August 2016

Agenda

1. Many Core vs Multi Core
2. Performance Optimization Concepts for Many Core
3. Performance Optimization Strategy for Many Core
4. Example Case Studies

Many Core HPC Systems





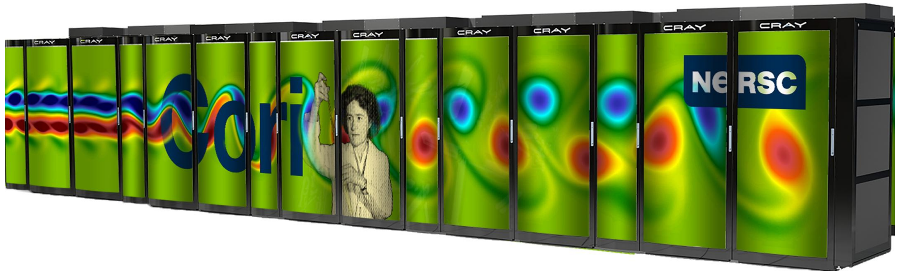
NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

Many Core Systems Coming to NERSC, ALCF and More

- **NERSC’s Cori will begin to transition the workload to more energy efficient architectures**
- **Cray XC system with over 9300 Intel Knights Landing (Xeon-Phi) compute nodes**
  - Self-hosted, (not an accelerator) manycore processor with 68 cores per node
  - On-package high-bandwidth memory



System named after Gerty Cori, Biochemist and first American woman to receive the Nobel prize in science.

What is different about Many-Core

Edison (Multi-Core):

- 5000+ Ivy Bridge Nodes
- 12 Cores Per CPU
- 24 Virtual Cores Per CPU

- 2.4-3.2 GHz

- Can do 4 Double Precision Operations per Cycle (+ multiply/add)

- 2.5 GB of Memory Per Core

- ~100 GB/s Memory Bandwidth

Cori (Many-Core):

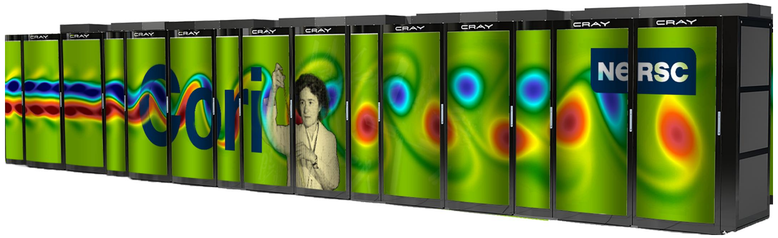
- 9000+ Knights Landing Nodes
- 68 Physical Cores Per CPU
- Up to 272 Virtual Cores Per CPU

- Much slower GHz

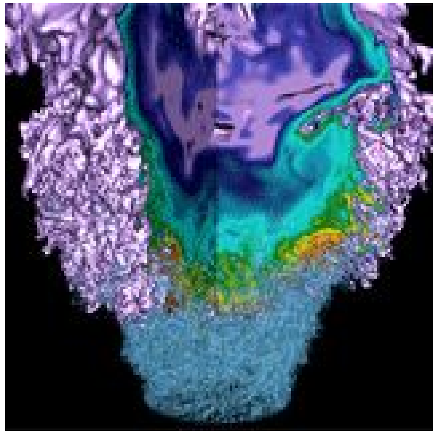
- Can do 8 Double Precision Operations per Cycle (+ multiply/add)

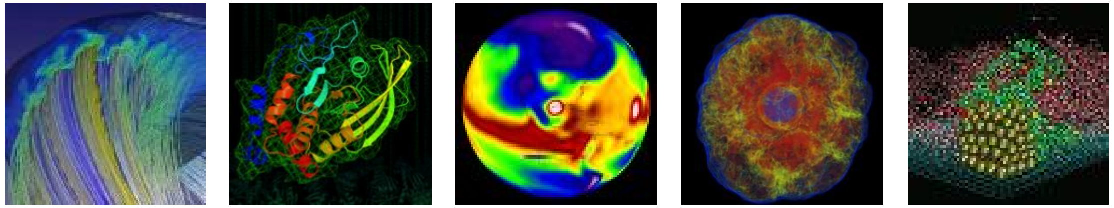
- < 0.3 GB of Fast Memory Per Core
  < 2 GB of Slow Memory Per Core

- Fast Memory has ~ 4-5x DDR4 Bandwidth



Basic Optimization Concepts





NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

MPI + X?

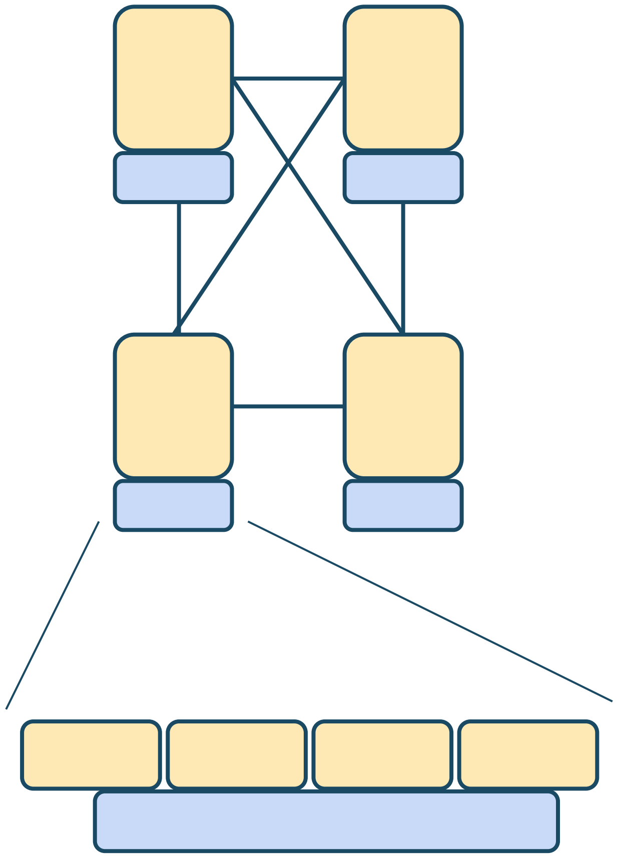
Need to explicitly consider both inter and on-node parallelism in application.

Existing applications may suffer from:

- Memory overhead due to duplicated data in traditional MPI tasks
- Lack of SIMD/Vectorization expressiveness in app.
- Potential MPI latency in all-to-all communication patterns

Possible Solutions:

MPI+MPI, MPI+OpenMP, PGAS (MPI+PGAS), Task Based Programming



PARATEC Use Case For OpenMP

PARATEC computes parallel FFTs across all processors.

Involves MPI all-to-all communication (small messages, latency bound).

Reducing the number of MPI tasks in favor OpenMP threads makes large improvement in overall runtime.

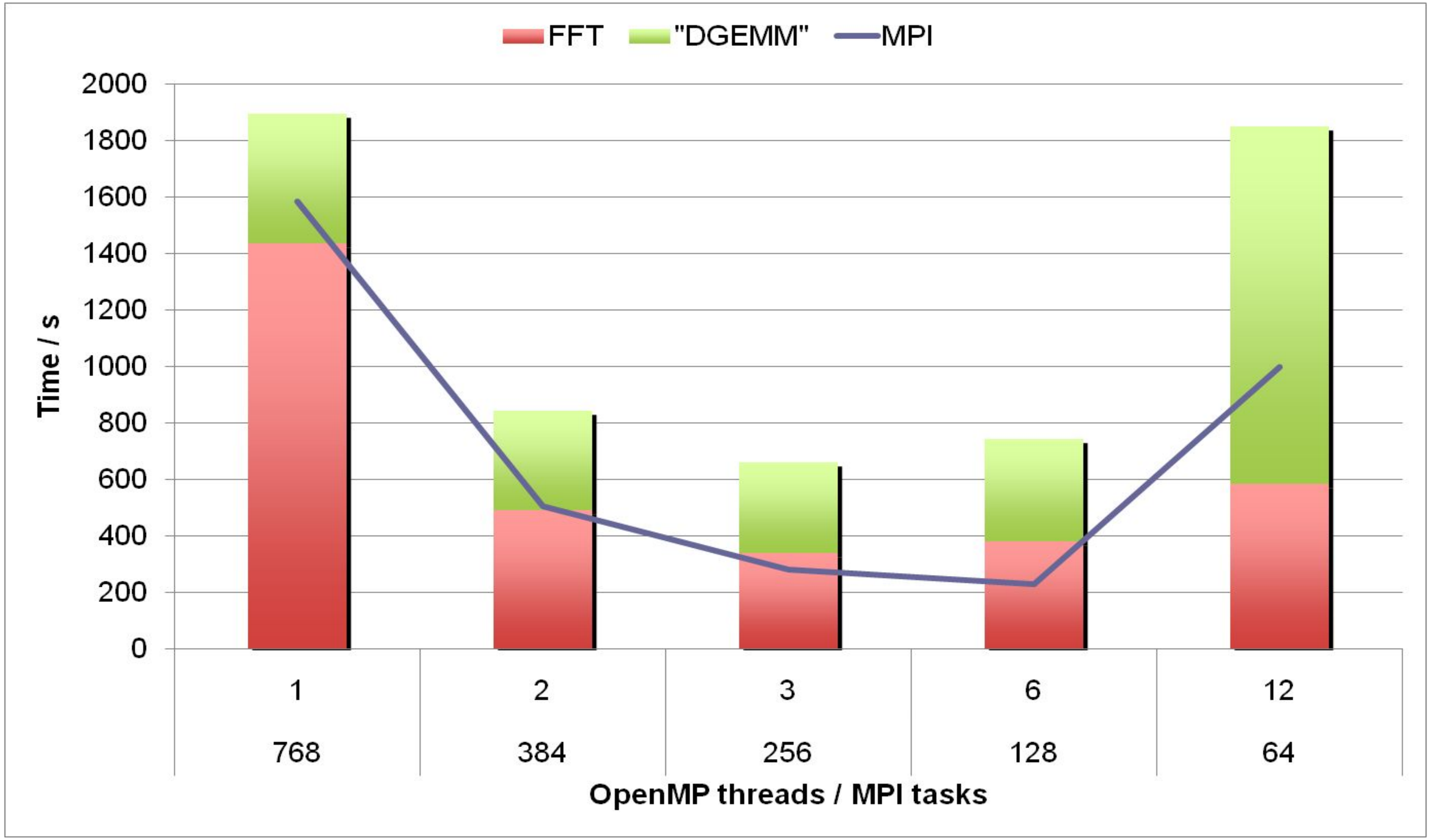


Figure Courtesy of Andrew Canning

Vectorization

There is a another important form of on-node parallelism

```
do i = 1, n
     a(i) = b(i) + c(i)
enddo
```

$$\downarrow$$

$$\begin{pmatrix} a_1 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \dots \\ b_n \end{pmatrix} + \begin{pmatrix} c_1 \\ \dots \\ c_n \end{pmatrix}$$

Vectorization: CPU does identical operations on different data; e.g., multiple iterations of the above loop can be done concurrently. Works best with long/aligned vectors.

Vectorization

There is a another important form of on-node parallelism

```
do i = 1, n
    a(i) = b(i) + c(i)
enddo
```

$$\begin{pmatrix} a_1 \\ \dots \end{pmatrix} = \begin{pmatrix} b_1 \\ \dots \end{pmatrix} + \begin{pmatrix} c_1 \\ \dots \end{pmatrix}$$

Intel Xeon Sandy-Bridge/Ivy-Bridge:	4 Double Precision Ops Concurrently
Intel Xeon Phi:	8 Double Precision Ops Concurrently

Vectori of the
above l

Things that prevent vectorization in your code

Compilers want to “vectorize” your loops whenever possible. But sometimes they get stumped. Here are a few things that prevent your code from vectorizing:

Loop dependency:

```
do i = 1, n
     a(i) = a(i-1) + b(i)
enddo
```

Task forking:

```
do i = 1, n
     if (a(i) < x) cycle
     if (a(i) > x) …
enddo
```

Things that prevent vectorization in your code

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)

```
for (many iterations) {
   … many flops …
   et = exp(outcome1)
   tt = pow(outcome2,3)
   IN = IN * et +tt
}
```

Things that prevent vectorization in your code

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)

```
for (many iterations) {
   … many flops …
   et = exp(outcome1)
   tt = pow(outcome2,3)
   IN = IN * et +tt
}
```

```
for (many iterations) {
   … many flops …
   et(i) = exp(outcome1)
   tt(i) = pow(outcome2,3)
}
for (many iterations) {
    IN = IN * et(i) + tt(i)
}
```

Things that prevent vectorization in your code

Example From NERSC User Group Hackathon - (Astrophysics Transport Code)

```
for (many iterations) {
  … many flops …
  et = exp(outcome1)
  tt = pow(outcome2,3)
  IN = IN * et +tt
}
```

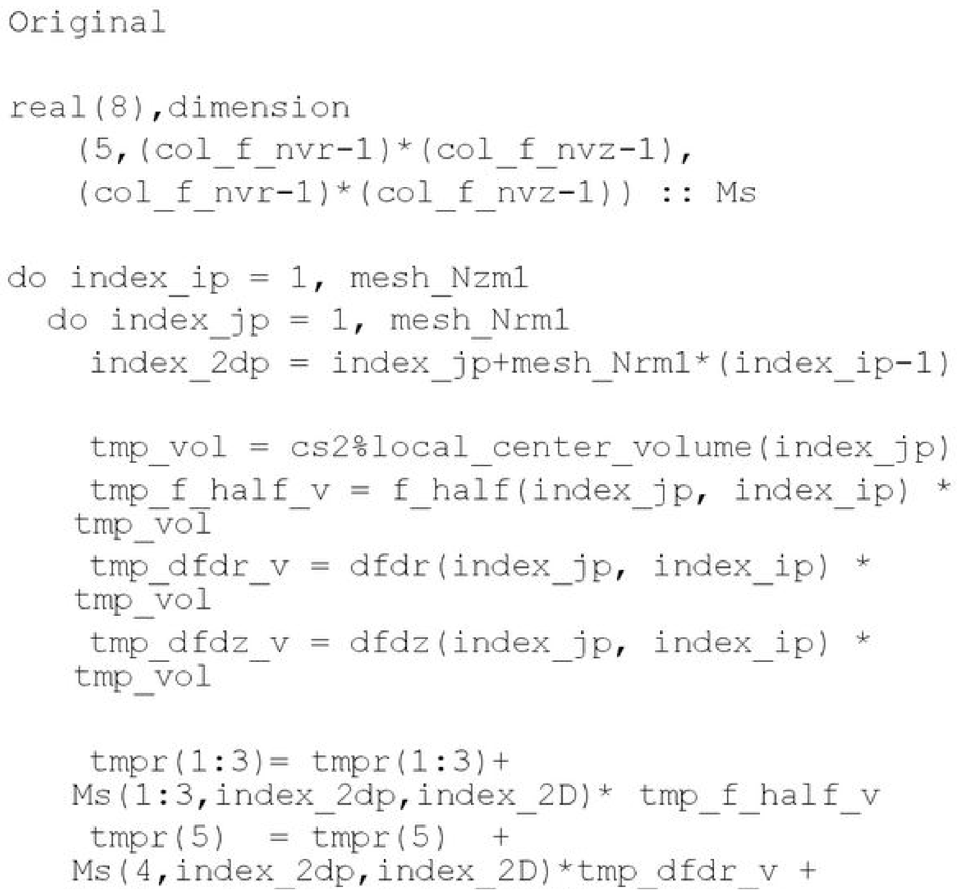
→

```
for (many iterations) {
  … many flops …
  et(i) = exp(outcome1)
  tt(i) = pow(outcome2,3)
}
for (many iterations) {
   IN = IN * et(i) + tt(i)
}
```

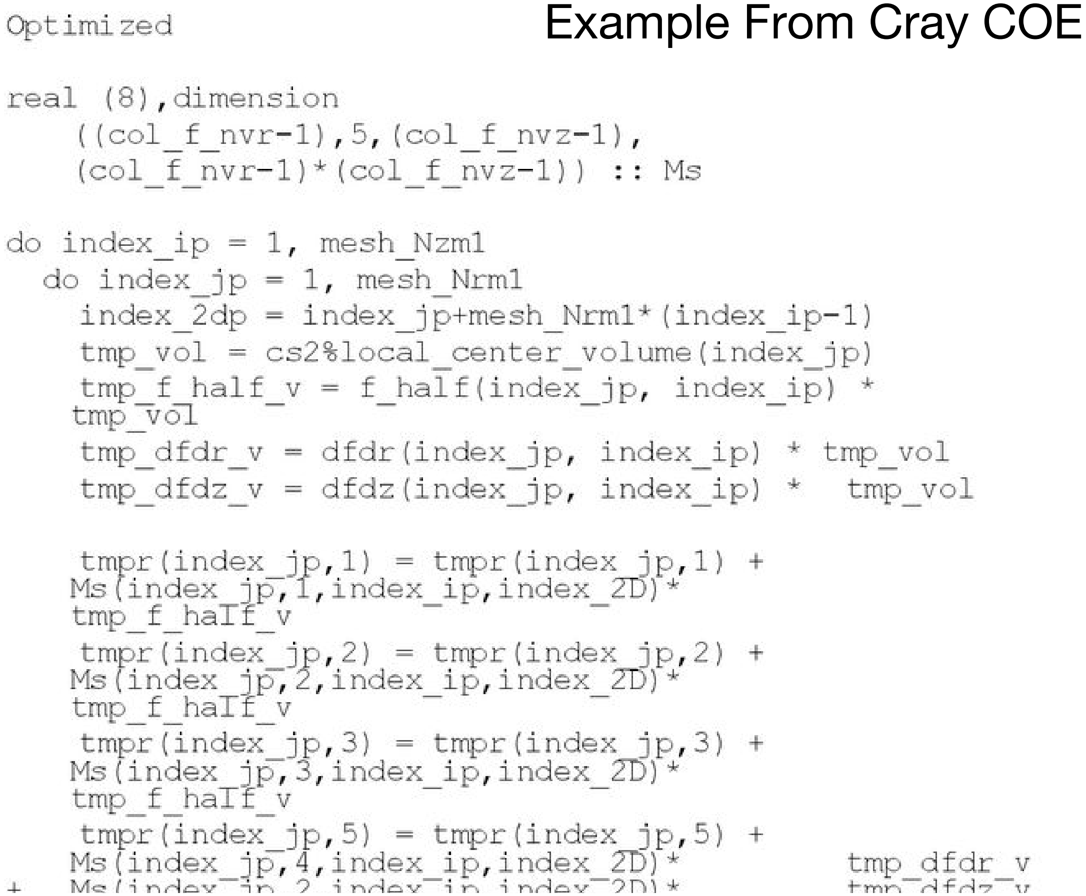
30% speed up for entire application!

Things that prevent vectorization in your code

Example From Cray COE Work on XGC1



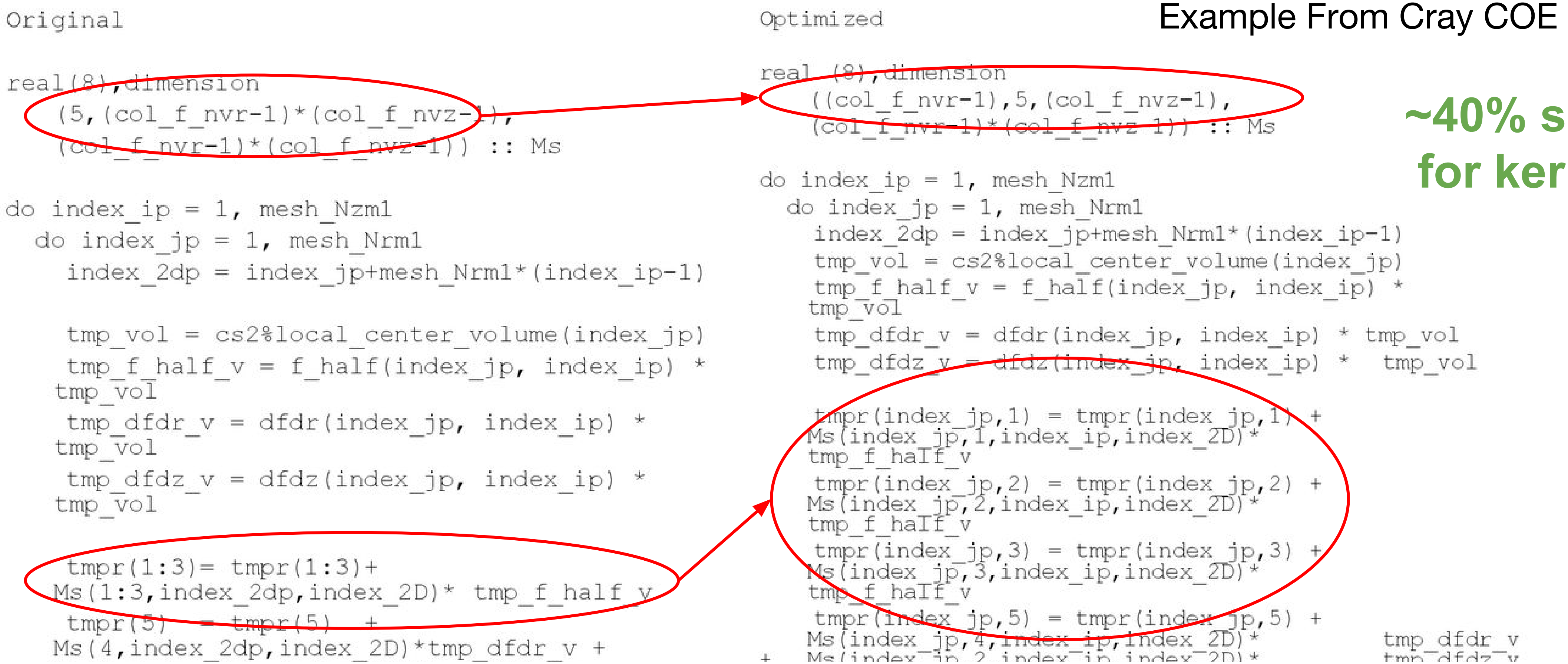
```
Original

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 +
```



```
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
```

Things that prevent vectorization in your code

Example From Cray COE Work on XGC1



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 +
```

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
```

~40% speed up for kernel

Memory Bandwidth

Consider the following loop:

```
do i = 1, n
     do j = 1, m
          c = c + a(i) * b(j)
     enddo
enddo
```

Assume, n & m are very large such that a & b don't fit into cache.

Then,

During execution, the **number of loads From DRAM** is

n*m + n

Memory Bandwidth

Consider the following loop:

```
do i = 1, n
    do j = 1, m
        c = c + a(i) * b(j)
    enddo
enddo
```

Assume, n & m are very large such that a & b don't fit into cache.

Then,

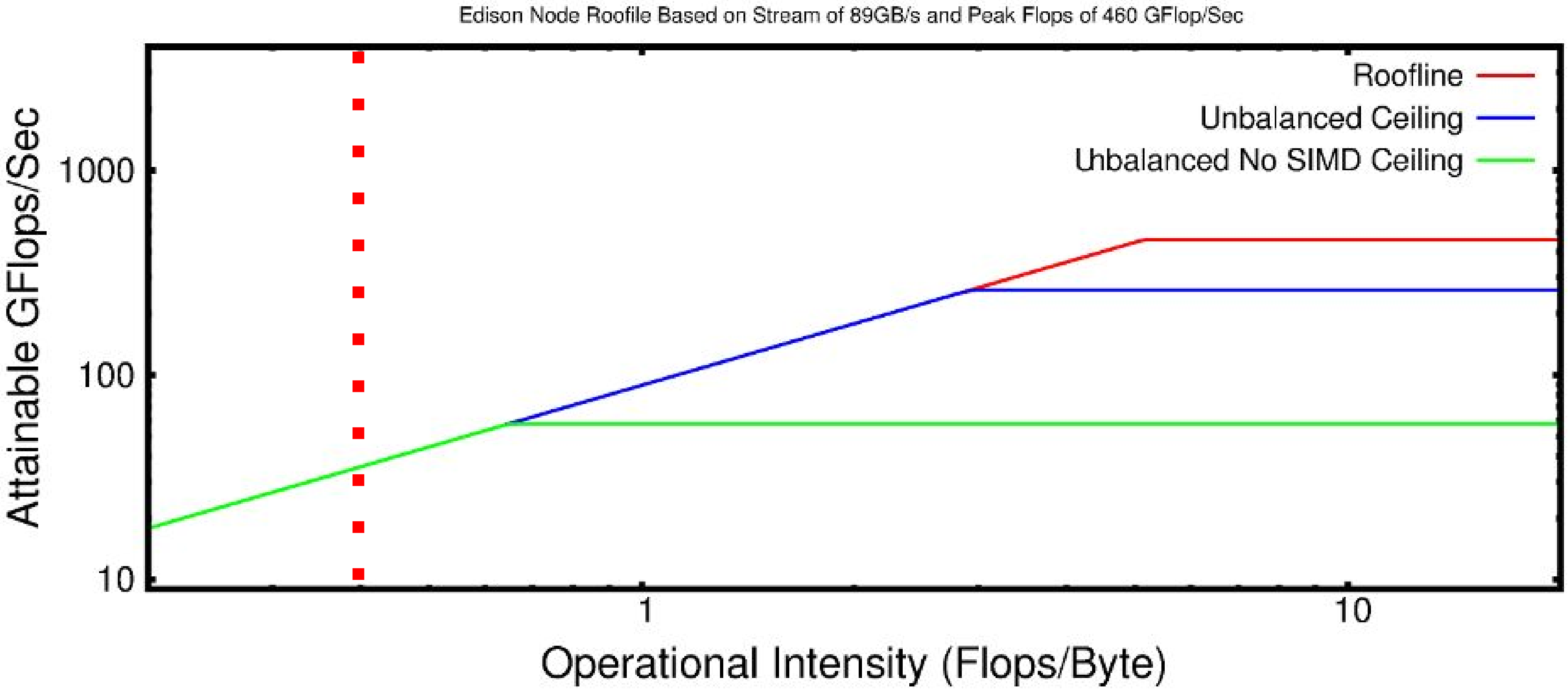
During execution, the **number of loads From DRAM** is

n*m + n

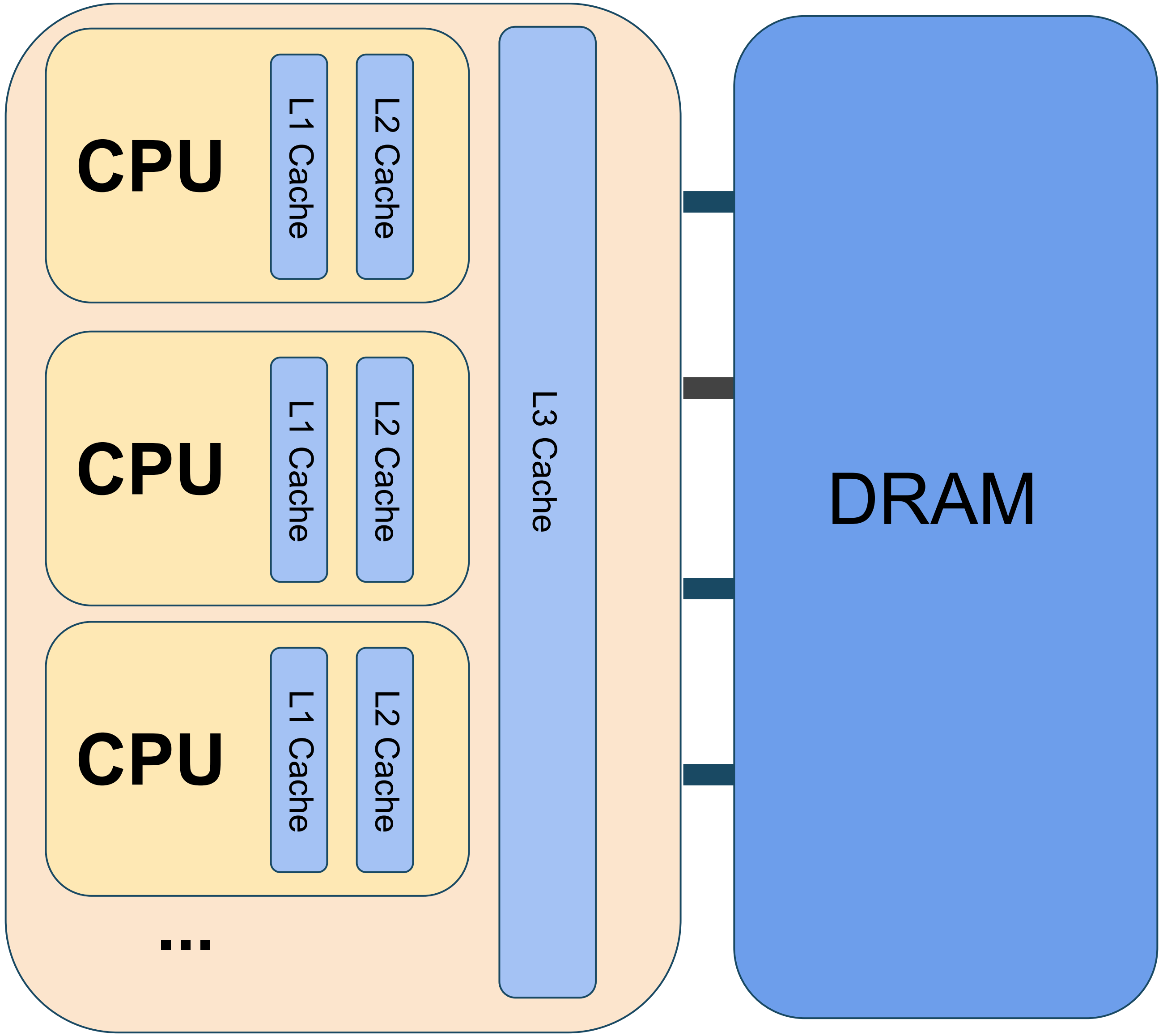
Requires 8 bytes loaded from DRAM per FMA (if supported). Assuming 100 GB/s bandwidth on Edison, we can at most achieve 25 GFlops/second (2 Flops per FMA)

Much lower than 460 GFlops/second peak on Edison node. Loop is memory bandwidth bound.

Roofline Model For Edison



Processor Memory Hierarchy on MultiCore



Improving Memory Locality

Improving Memory Locality. Reducing bandwidth required.

```
do i = 1, n
     do j = 1, m
          c = c + a(i) * b(j)
     enddo
enddo
```

Loads From DRAM:

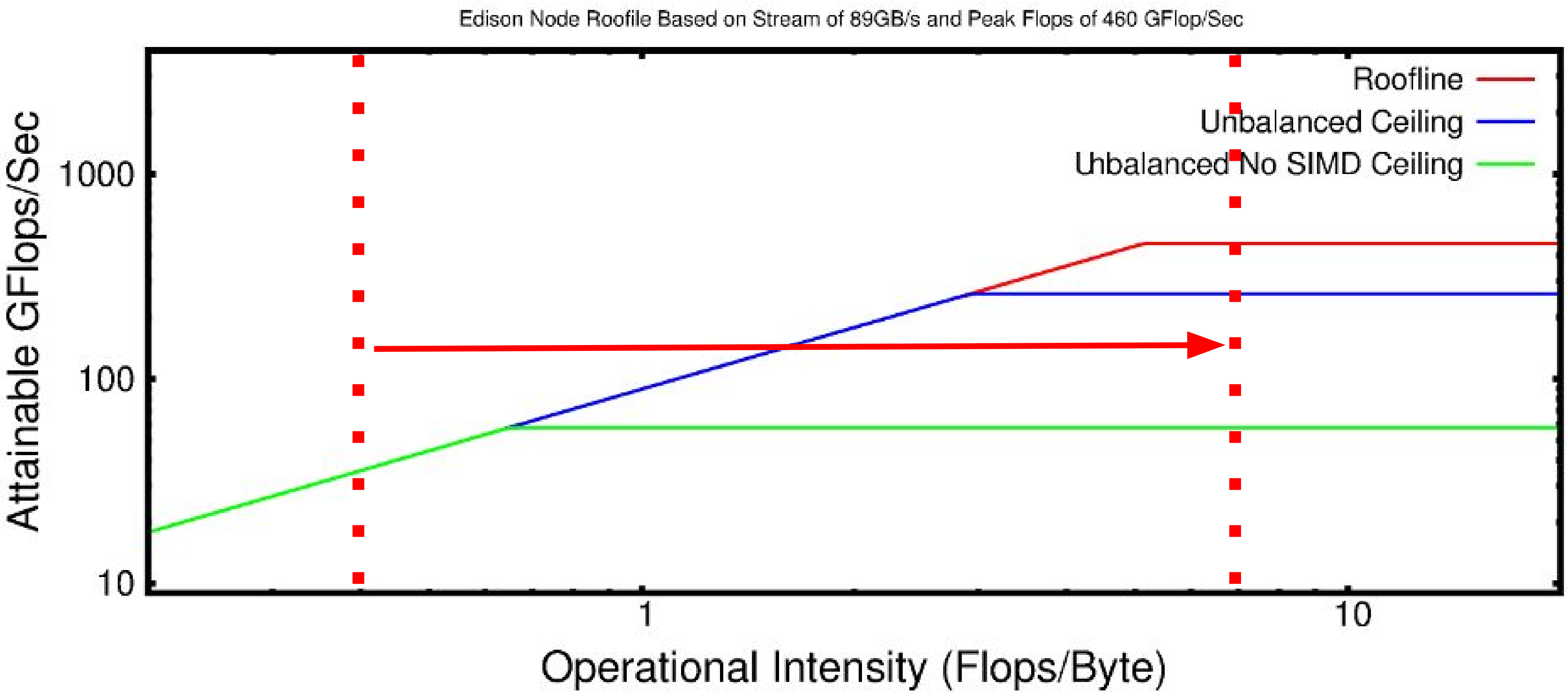
n*m + n

```
do jout = 1, m, block
     do i = 1, n
          do j = jout, jout+block
               c = c + a(i) * b(j)
           enddo
     enddo
enddo
```

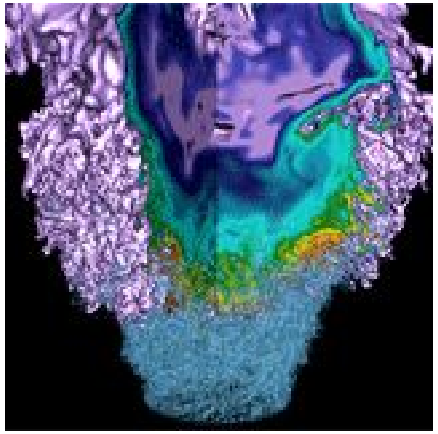
Loads From DRAM:

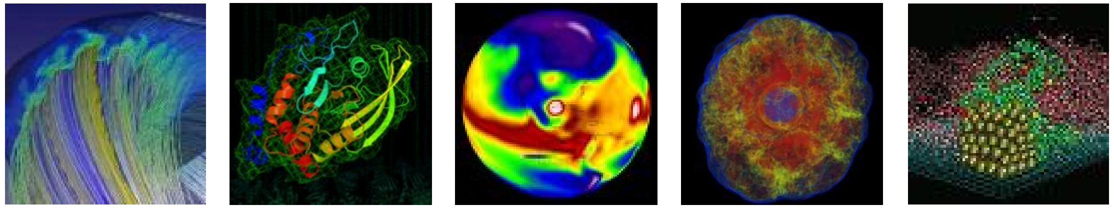
m/block * (n+block)
= **n*m/block** + m

Improving Memory Locality Moves you to the Right on the Roofline

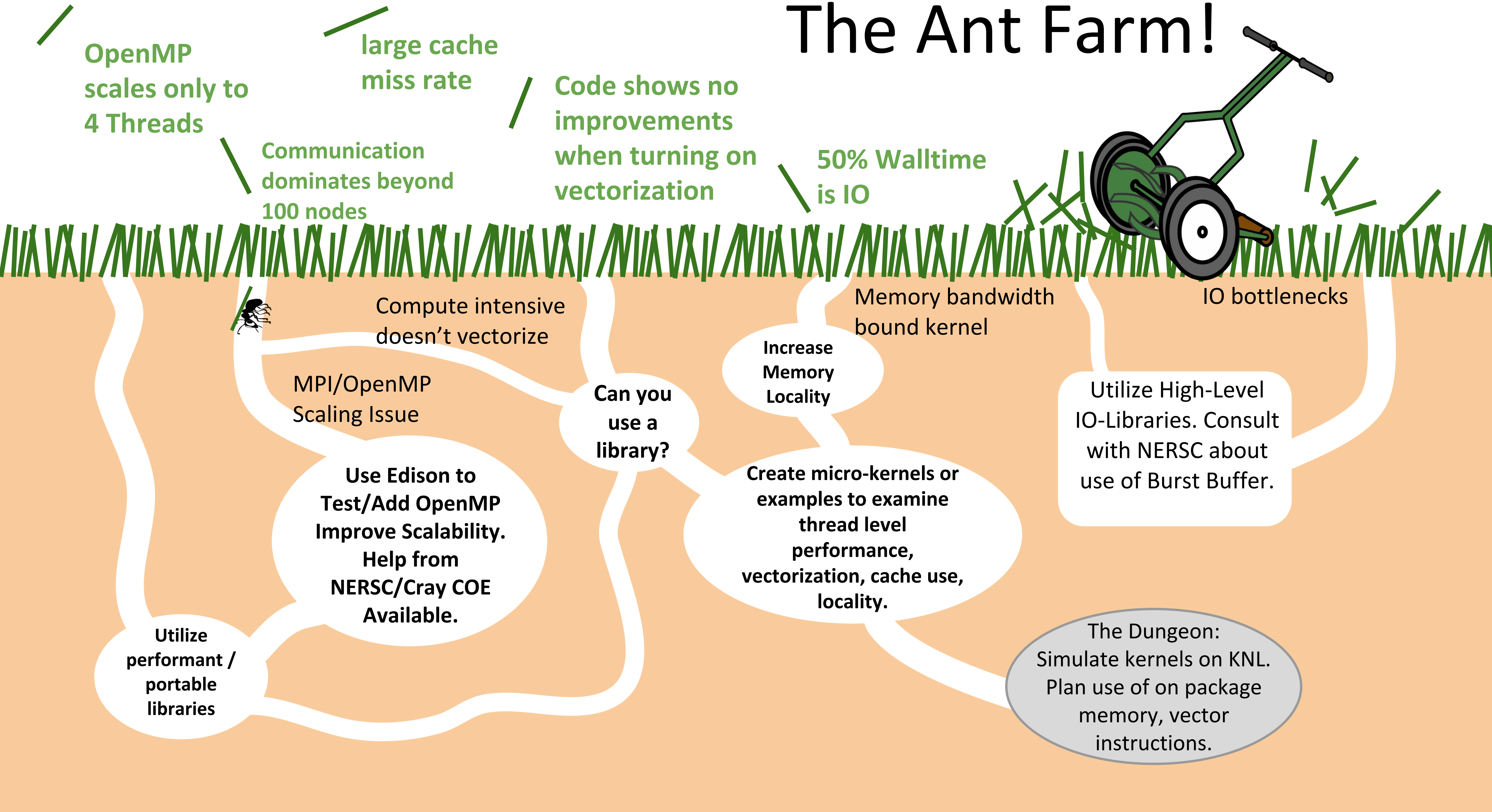


Optimization Strategy





The Ant Farm!



OpenMP scales only to 4 Threads

large cache miss rate

Communication dominates beyond 100 nodes

Code shows no improvements when turning on vectorization

50% Walltime is IO

Compute intensive doesn't vectorize

MPI/OpenMP Scaling Issue

Use Edison to Test/Add OpenMP Improve Scalability. Help from NERSC/Cray COE Available.

Utilize performant / portable libraries

Can you use a library?

Memory bandwidth bound kernel

Increase Memory Locality

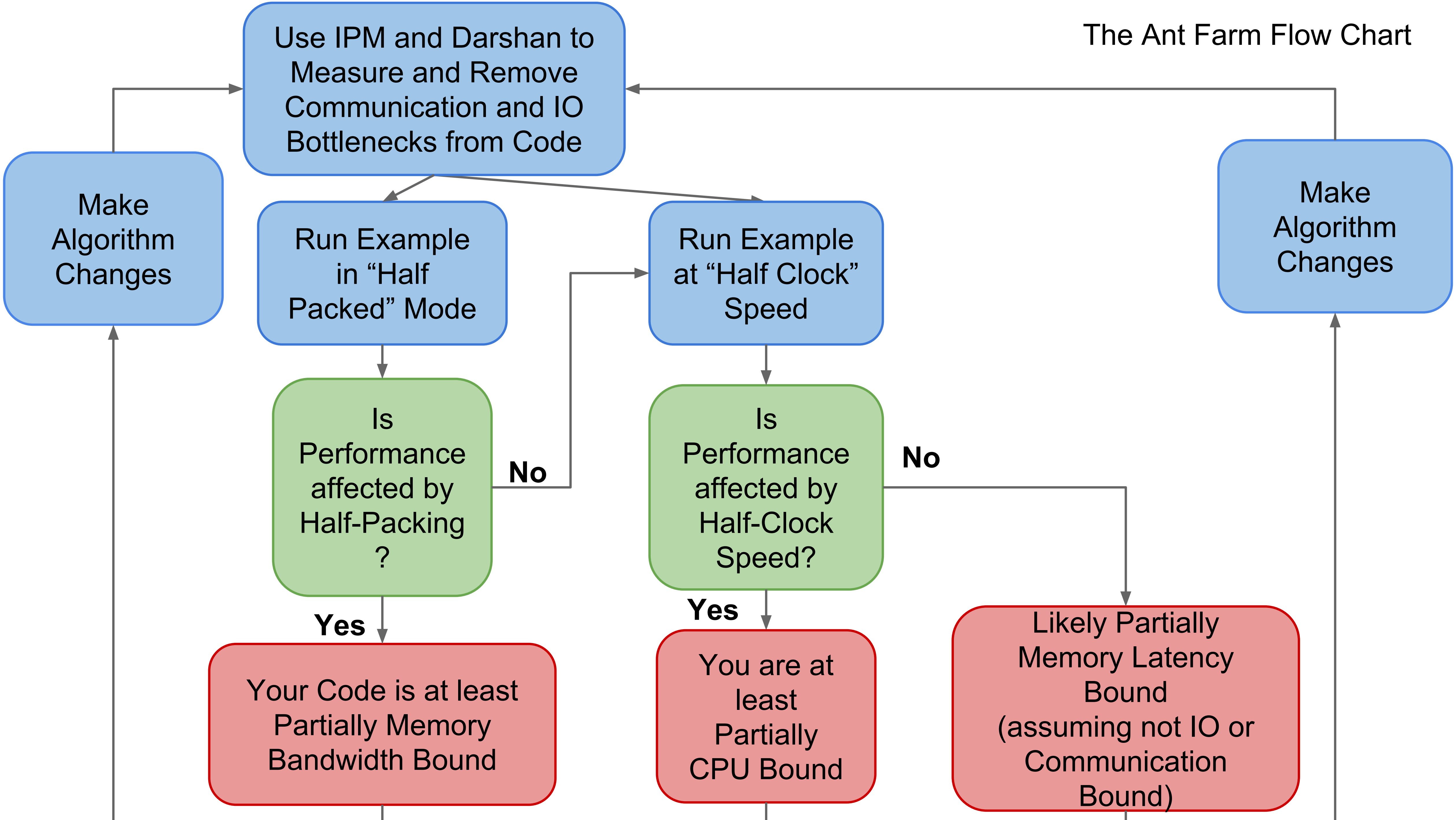
Create micro-kernels or examples to examine thread level performance, vectorization, cache use, locality.

IO bottlenecks

Utilize High-Level IO-Libraries. Consult with NERSC about use of Burst Buffer.

The Dungeon: Simulate kernels on KNL. Plan use of on package memory, vector instructions.

The Ant Farm Flow Chart



Use IPM and Darshan to Measure and Remove Communication and IO Bottlenecks from Code

Make Algorithm Changes

Run Example in “Half Packed” Mode

Is Performance affected by Half-Packing ?

No

Yes

Your Code is at least Partially Memory Bandwidth Bound

Run Example at “Half Clock” Speed

Is Performance affected by Half-Clock Speed?

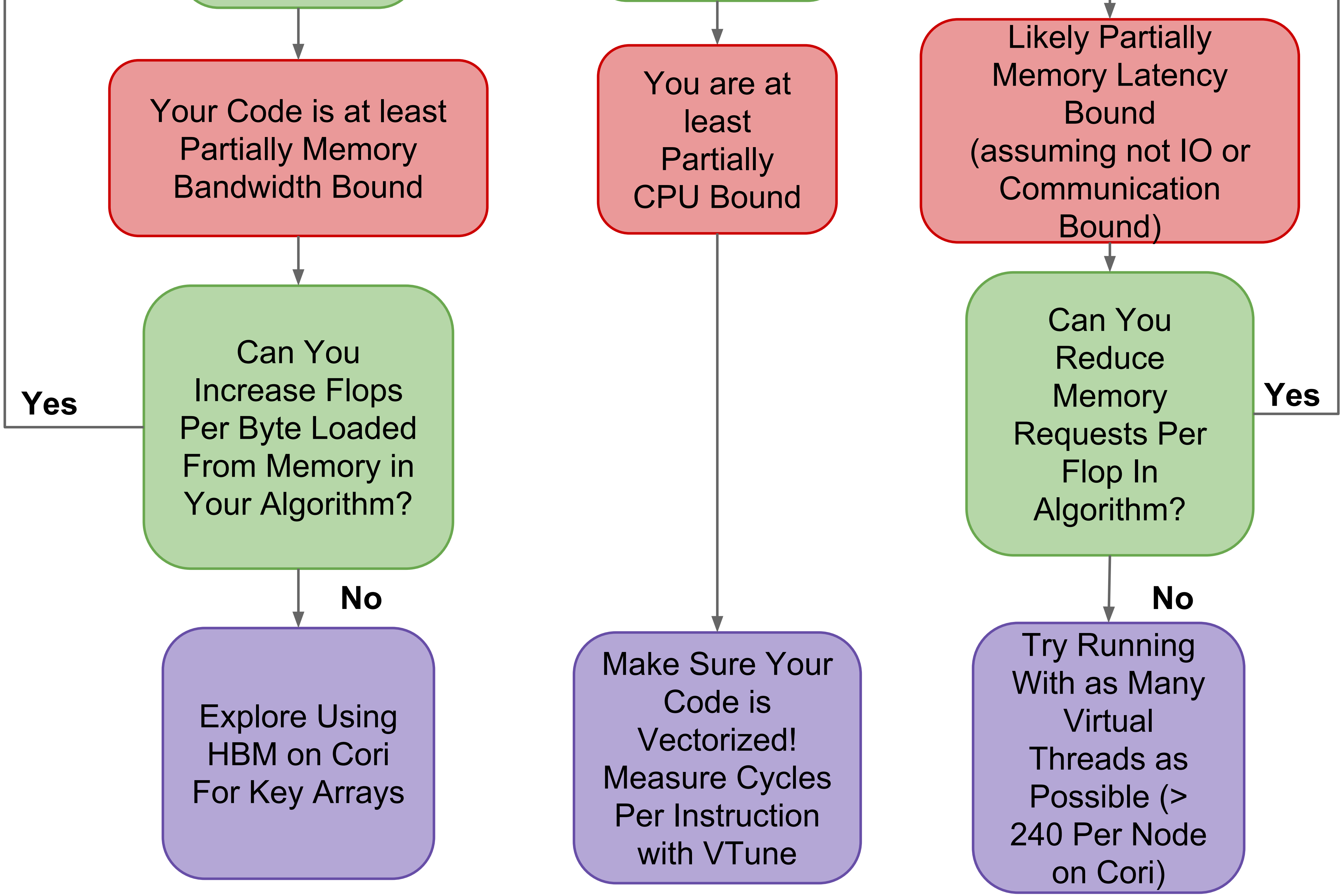
Yes

You are at least Partially CPU Bound

No

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)

Make Algorithm Changes



Your Code is at least Partially Memory Bandwidth Bound

Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?

Yes

No

Explore Using HBM on Cori For Key Arrays

You are at least Partially CPU Bound

Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune

Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)

Can You Reduce Memory Requests Per Flop In Algorithm?

Yes

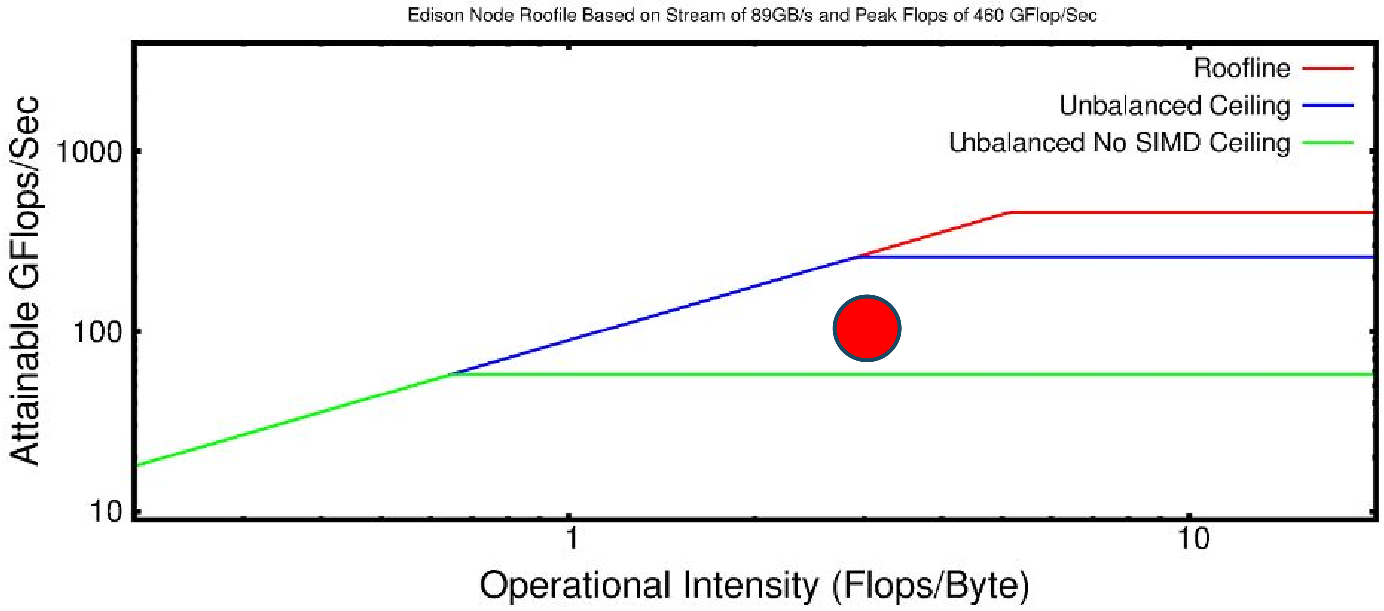
No

Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)

Are You Memory Bound? Compute Bound? Neither?

1. Determine your roofline position:

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/



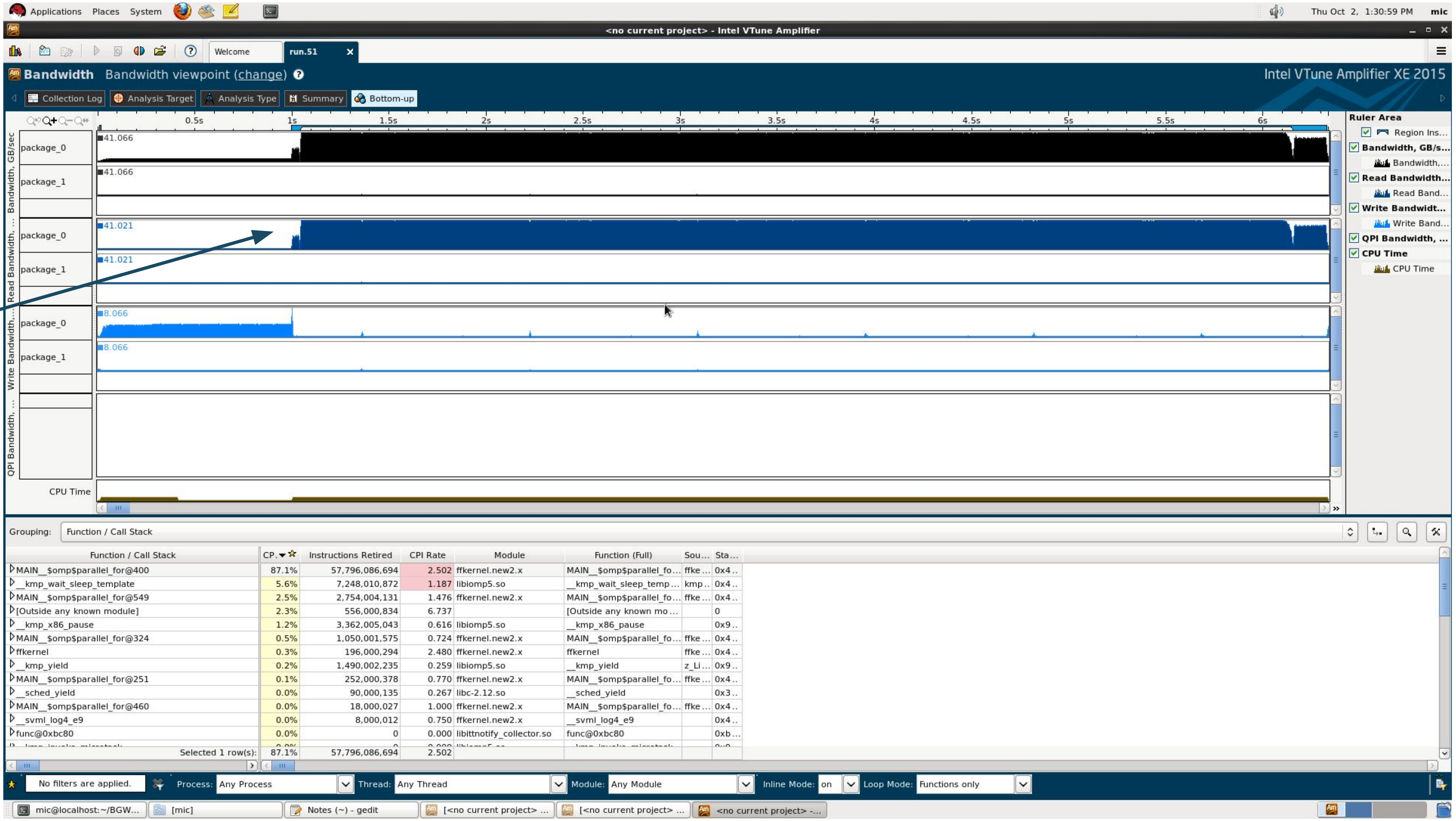
Measuring Your Memory Bandwidth Usage (VTune)

Measure memory bandwidth usage in VTune. (Next Talk)

Compare to Stream GB/s.

If 90% of stream, you are memory bandwidth bound.

If less, more tests need to be done.

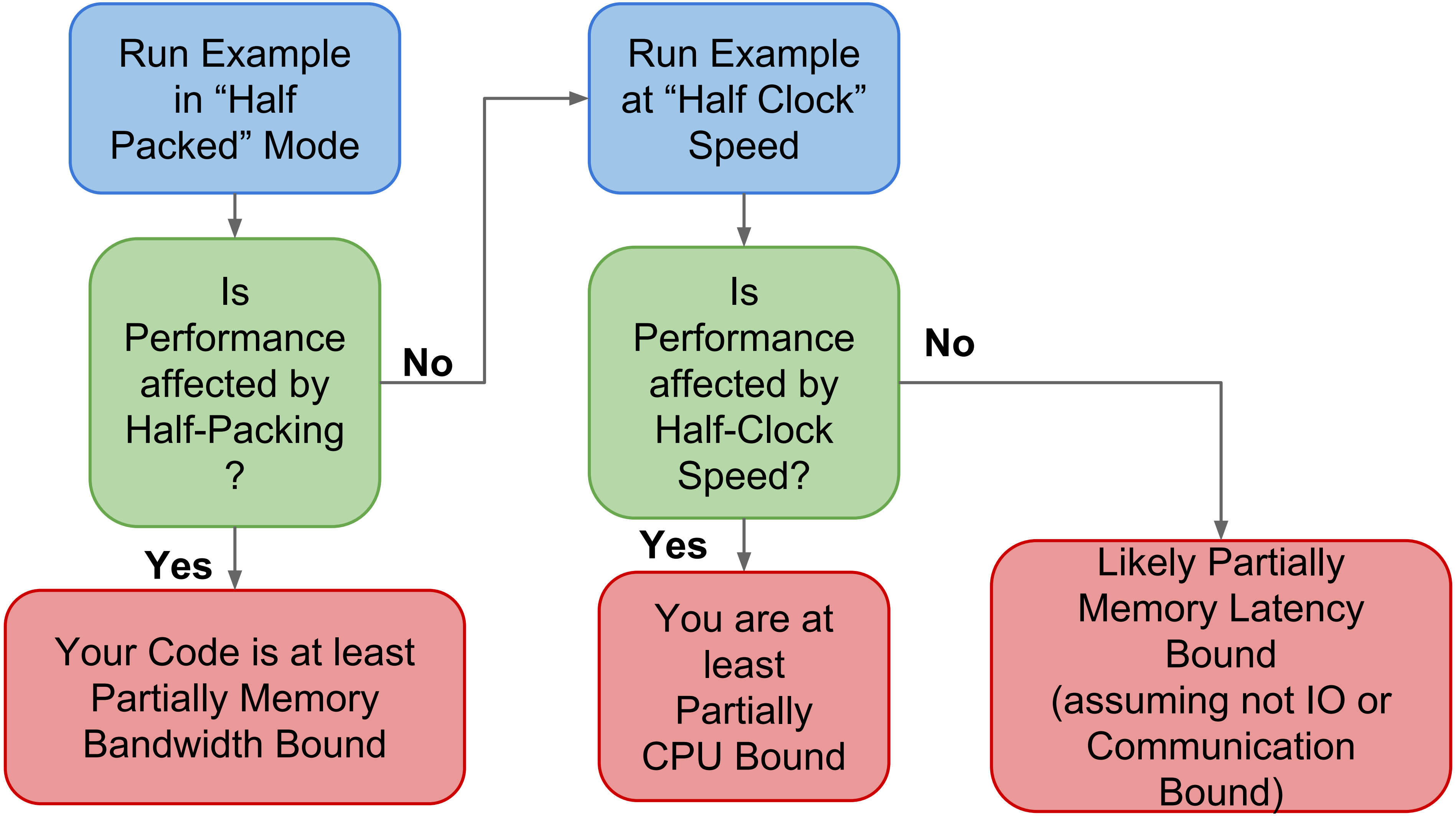


Grouping: Function / Call Stack

Function / Call Stack	CP.	Instructions Retired	CPI Rate	Module	Function (Full)	Sou...	Sta...
MAIN__ompparallel_for@400	87.1%	57,796,086,694	2.502	ffkernel.new2.x	MAIN__ompparallel_fo...	ffke...	0x4..
__kmp_wait_sleep_template	5.6%	7,248,010,872	1.187	libiomp5.so	__kmp_wait_sleep_temp...	kmp..	0x4..
MAIN__ompparallel_for@549	2.5%	2,754,004,131	1.476	ffkernel.new2.x	MAIN__ompparallel_fo...	ffke...	0x4..
[Outside any known module]	2.3%	556,000,834	6.737		[Outside any known mo...		0
__kmp_x86_pause	1.2%	3,362,005,043	0.616	libiomp5.so	__kmp_x86_pause		0x9..
MAIN__ompparallel_for@324	0.5%	1,050,001,575	0.724	ffkernel.new2.x	MAIN__ompparallel_fo...	ffke...	0x4..
ffkernel	0.3%	196,000,294	2.480	ffkernel.new2.x	ffkernel	ffke...	0x4..
__kmp_yield	0.2%	1,490,002,235	0.259	libiomp5.so	__kmp_yield	z_Li...	0x9..
MAIN__ompparallel_for@251	0.1%	252,000,378	0.770	ffkernel.new2.x	MAIN__ompparallel_fo...	ffke...	0x4..
__sched_yield	0.0%	90,000,135	0.267	libc-2.12.so	__sched_yield		0x3..
MAIN__ompparallel_for@460	0.0%	18,000,027	1.000	ffkernel.new2.x	MAIN__ompparallel_fo...	ffke...	0x4..
__svml_log4_e9	0.0%	8,000,012	0.750	ffkernel.new2.x	__svml_log4_e9		0x4..
func@0xbc80	0.0%	0	0.000	libittnotify_collector.so	func@0xbc80		0xb...
Selected 1 row(s):	87.1%	57,796,086,694	2.502				

No filters are applied. Process: Any Process Thread: Any Thread Module: Any Module Inline Mode: on Loop Mode: Functions only

Are you memory or compute bound? Or both?



Run Example in "Half Packed" Mode

Is Performance affected by Half-Packing ?

Yes → Your Code is at least Partially Memory Bandwidth Bound

No → Run Example at "Half Clock" Speed

Is Performance affected by Half-Clock Speed?

Yes → You are at least Partially CPU Bound

No → Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)

Are you memory or compute bound? Or both?

Run Example in “Half Packed” Mode

If you run on only half of the cores on a node, each core you do run has access to more bandwidth

```
aprun -n 24 -N 12 - S 6 ...
```

VS

```
aprun -n 24 -N 24 -S 12 ...
```

If your performance changes, you are at least partially memory bandwidth bound

Are you memory or compute bound? Or both?

Run Example in "Half Packed" Mode

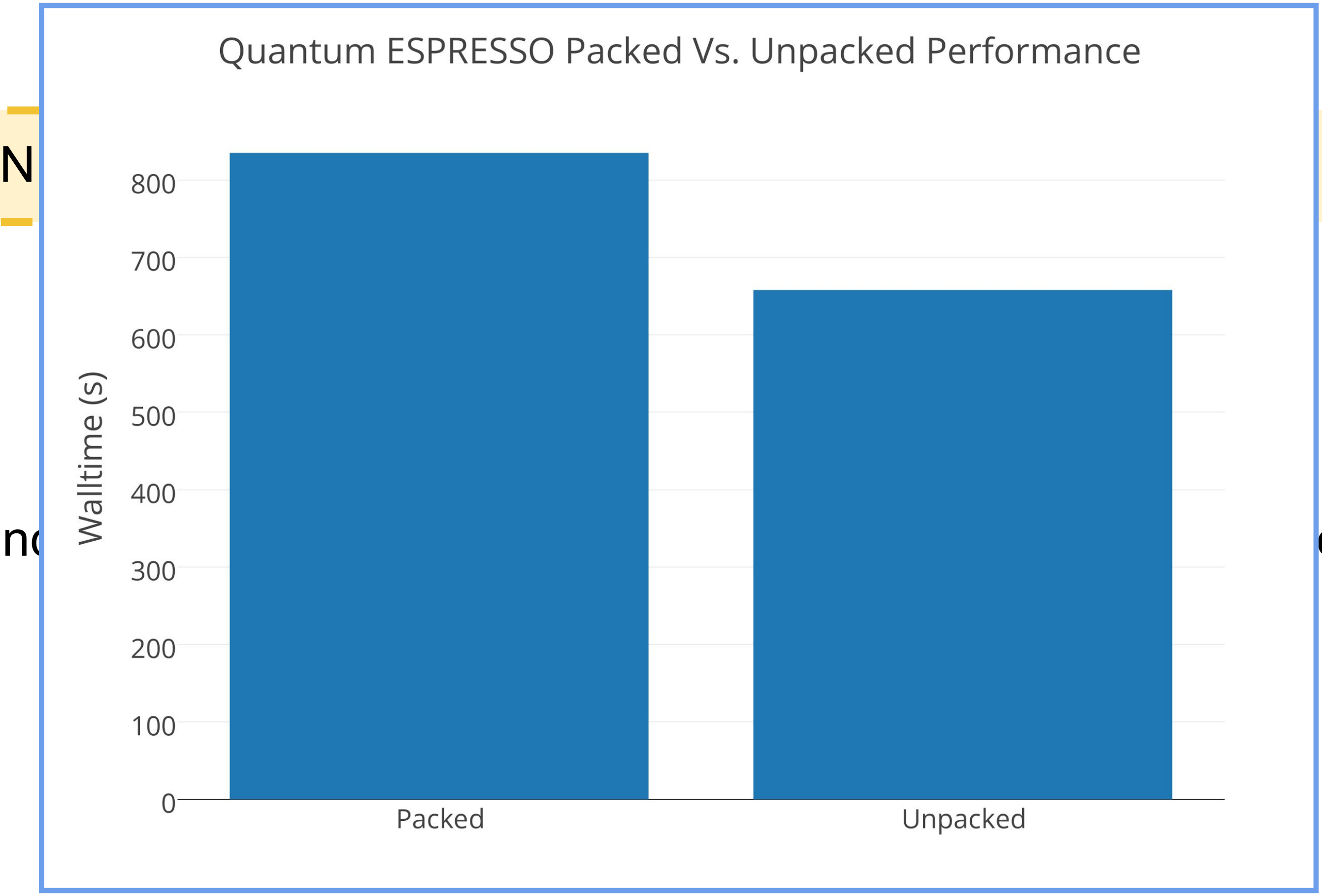
If you run on only half of the cores on a node, each core you do run has access to more bandwidth

aprun -n 24 -N

2 ...

If your performanc

ound



Are you memory or compute bound? Or both?

Run Example at “Half Clock” Speed

Reducing the CPU speed slows down computation, but doesn’t reduce memory bandwidth available.

```
aprun --p-state=2400000 ...
```

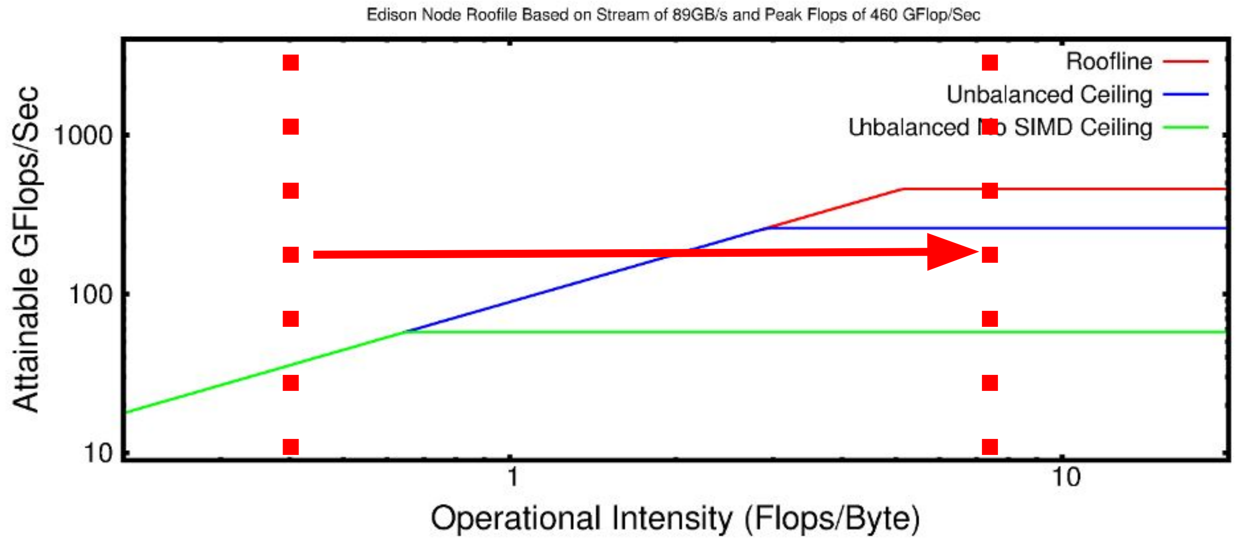
VS

```
aprun --p-state=1900000 ...
```

If your performance changes, you are at least partially compute bound

So, you are Memory Bandwidth Bound?

What to do?



1. Try to improve memory locality, cache reuse

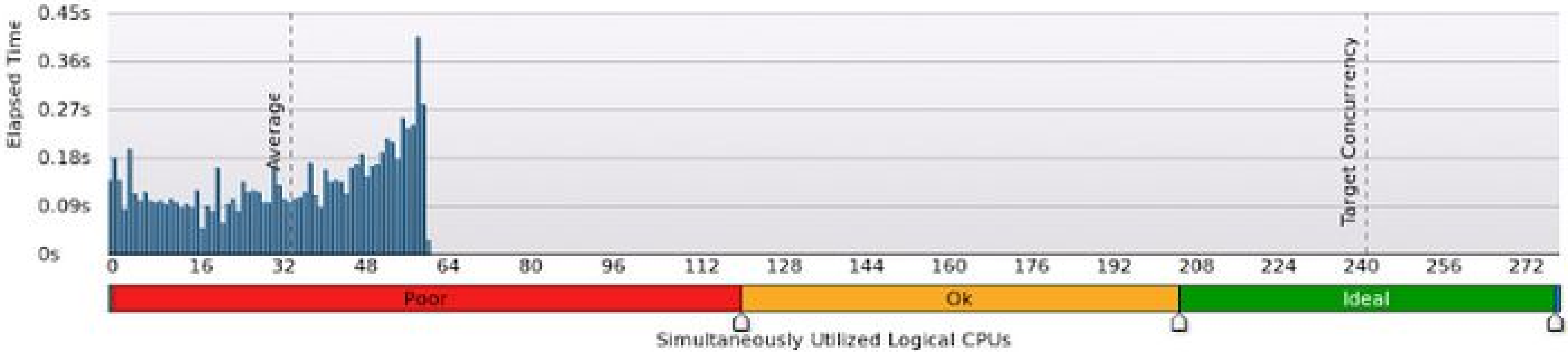
2. Identify the key arrays leading to high memory bandwidth usage and make sure they are/will-be allocated in HBM on Cori.

   Profit by getting ~ 5x more bandwidth GB/s.

So, you are Compute Bound?

What to do?

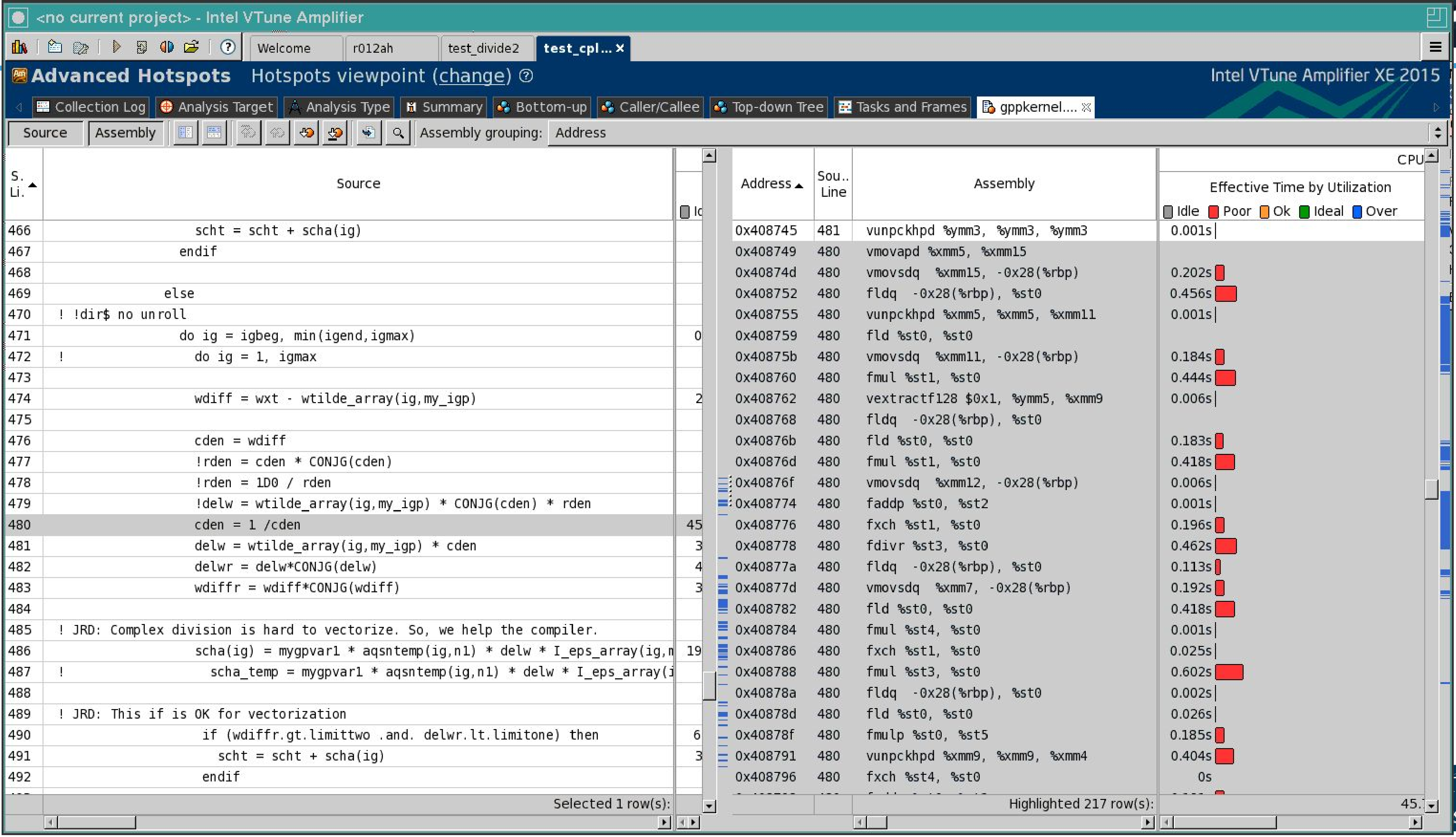
1. Make sure you have good OpenMP scalability. Look at VTune to see thread activity for major OpenMP regions.



2. Make sure your code is vectorizing. Look at Cycles per Instruction (CPI) and VPU utilization in vtune.

See whether intel compiler vectorized loop using compiler flag: -qopt-report=5

High latency instructions : Complex-Division (without -fp model fast=2)



S. Li.	Source	Ic
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,n	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	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.

Are you latency bound?

You may be memory latency bound (or you may be spending all your time in IO and Communication).

If running with hyper-threading on Edison improves performance, you *might* be latency bound:

```
aprun -j 2 -n 48 ….
```

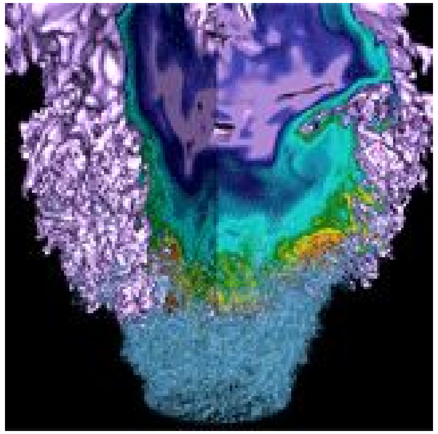
vs

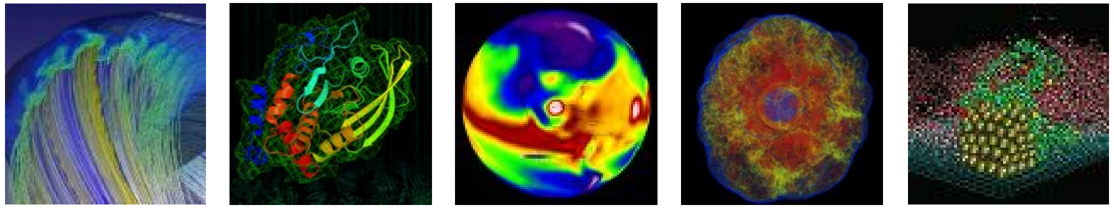
```
aprun -n 24 ….
```

If you can, try to reduce the number of memory requests per flop by accessing contiguous and predictable segments of memory and reusing variables in cache as much as possible.

On Cori, each core will support up to 4 threads. Use them all.

NESAP Case Study





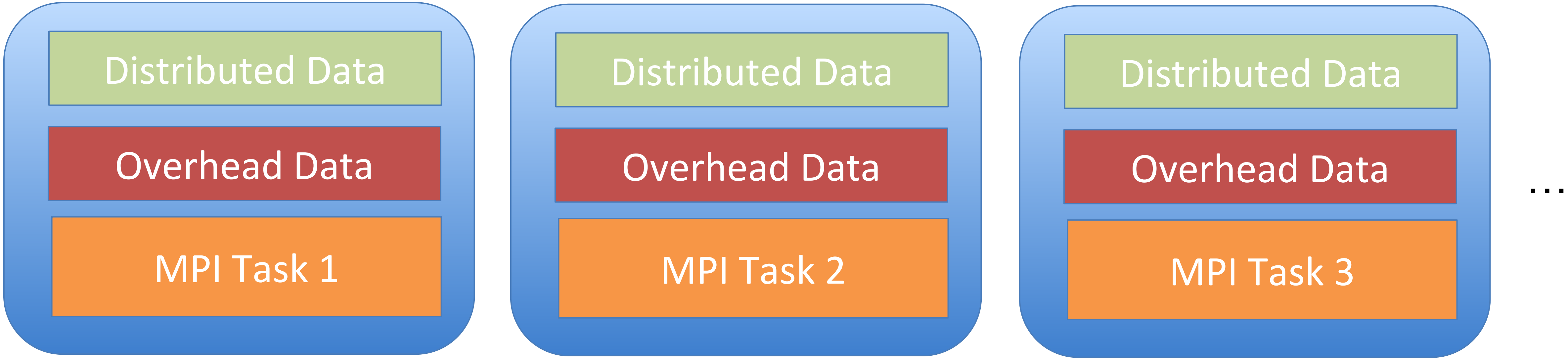
NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

BerkeleyGW Use Case

- Big systems require more memory. Cost scales as N_{atoms}^2 to store the data.
- 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

…

Computational Bottlenecks

In house code (I’m one of main developers). Use as “prototype” for App Readiness.

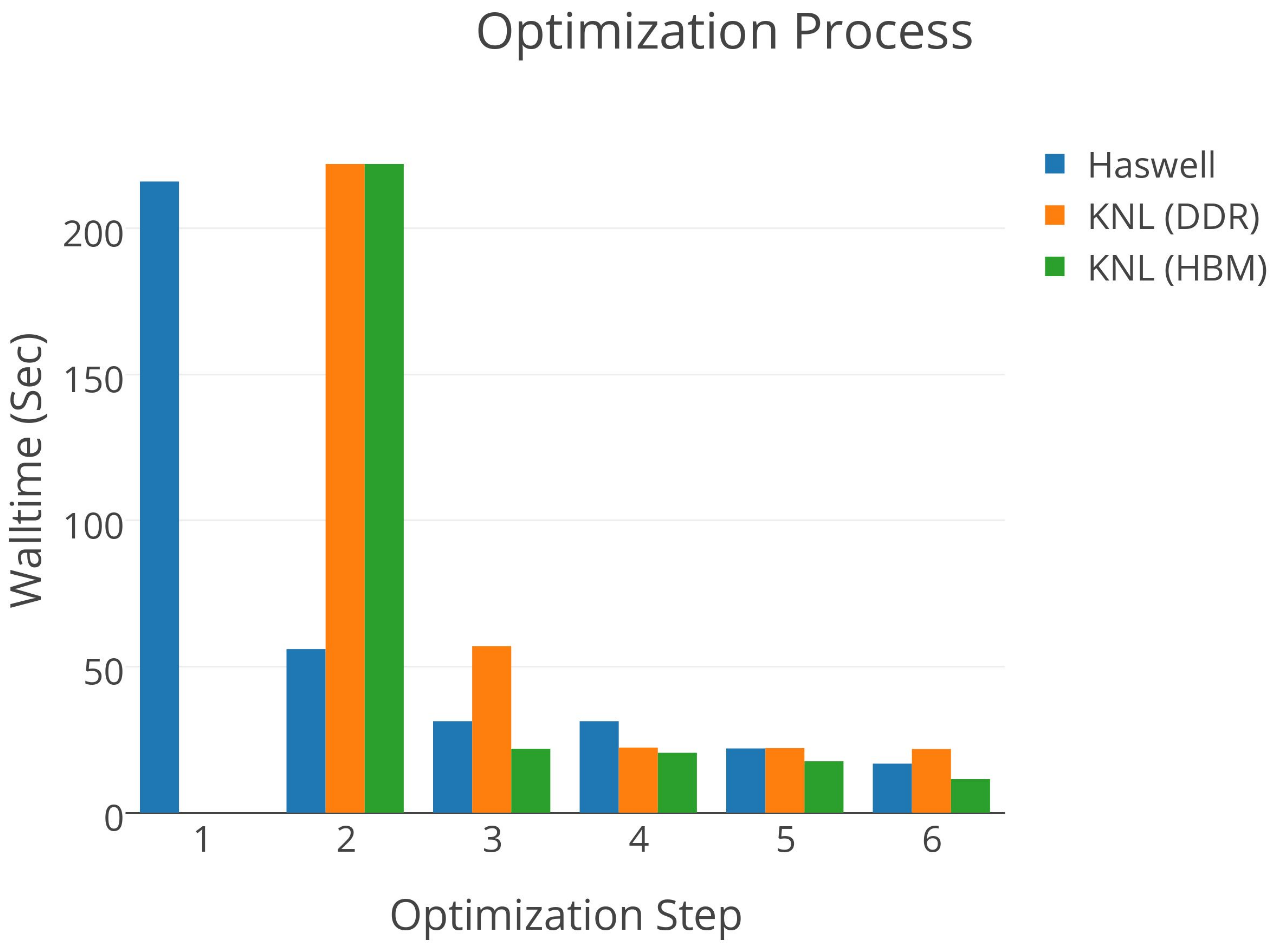
Significant Bottleneck is large matrix reduction like operations. Turning arrays into numbers.

$$\langle n\mathbf{k}|\, \Sigma_{\mathrm{CH}}(E)\, |n'\mathbf{k}\rangle = \frac{1}{2}\sum_{n''}\sum_{\mathbf{q}\mathbf{G}\mathbf{G}'} M^{*}_{n''n}(\mathbf{k},-\mathbf{q},-\mathbf{G})M_{n''n'}(\mathbf{k},-\mathbf{q},-\mathbf{G}') \times \frac{\Omega^{2}_{\mathbf{G}\mathbf{G}'}(\mathbf{q})\,(1-i\tan\phi_{\mathbf{G}\mathbf{G}'}(\mathbf{q}))}{\tilde{\omega}_{\mathbf{G}\mathbf{G}'}(\mathbf{q})\,(E-E_{n''\mathbf{k}-\mathbf{q}}-\tilde{\omega}_{\mathbf{G}\mathbf{G}'}(\mathbf{q}))}\, v(\mathbf{q}+\mathbf{G}')$$

Optimization Path

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



Vectorization

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

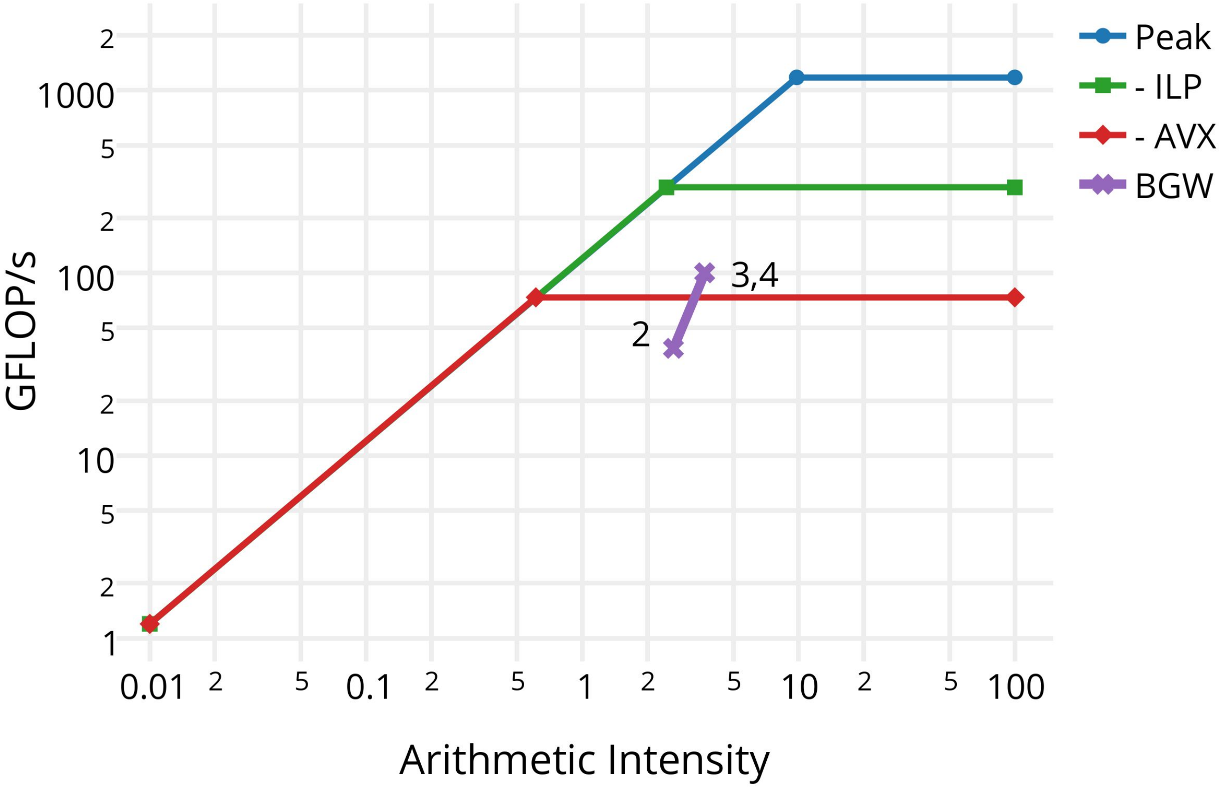
ngpown typically in 100’s to 1000s. Good for many threads.

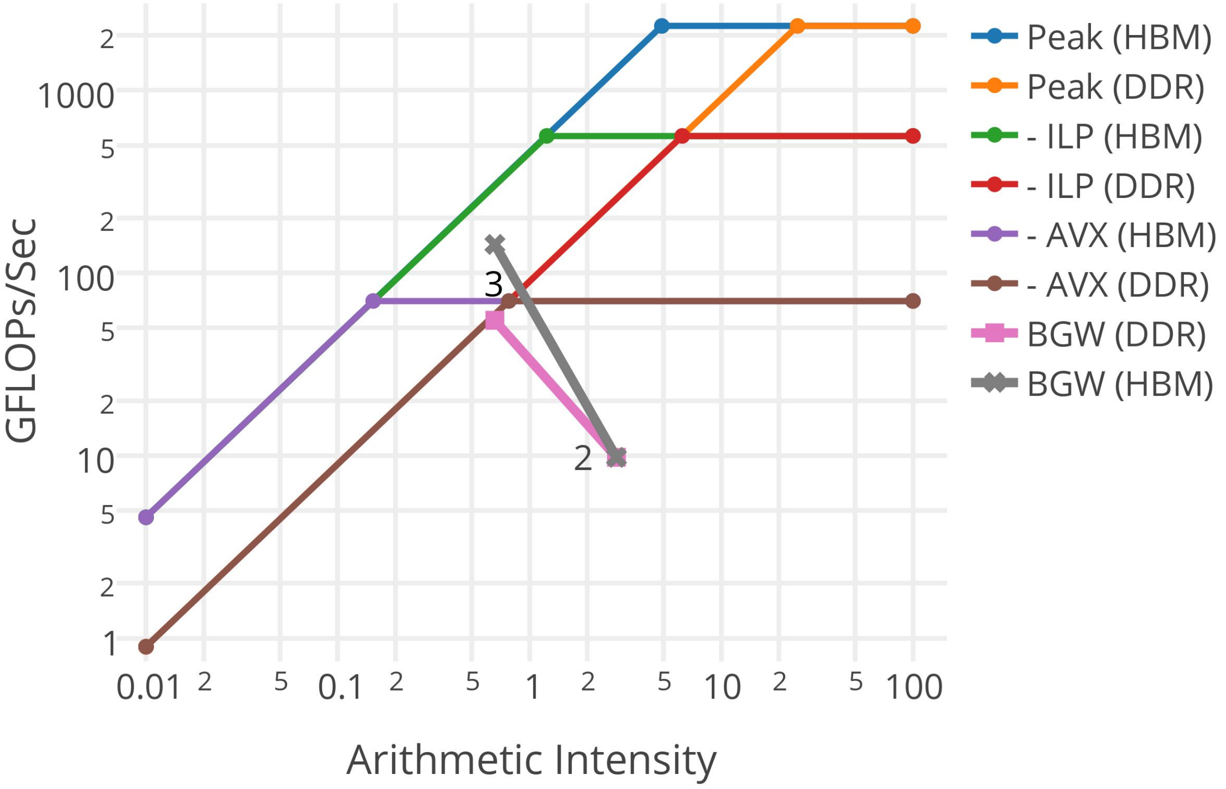
Original inner loop. Too small to vectorize!

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

Attempt to save work breaks vectorization and makes code slower.

Change in Roofline


Haswell Roofline Optimization Path


KNL Roofline Optimization Path

The loss of L3 on MIC makes locality more important.

Why KNC worse than Haswell for GPP Kernel?

- 2S Haswell 27.9s   KNC 39.9s   (Bandwidth bound on KNC but not on Haswell)

```
!$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 KNC worse than Haswell for GPP Kernel?

- 2S Haswell 27.9s KNC 39.9s (Bandwidth bound on KNC but not on Haswell)

```
!$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:

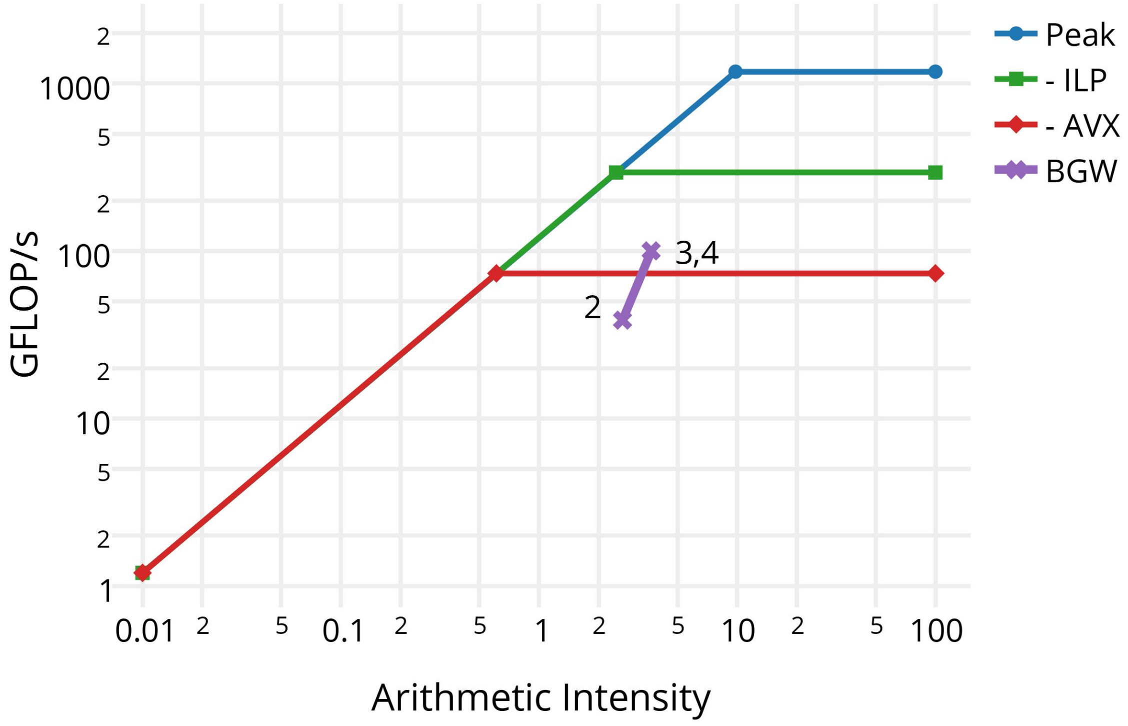
1536 KB

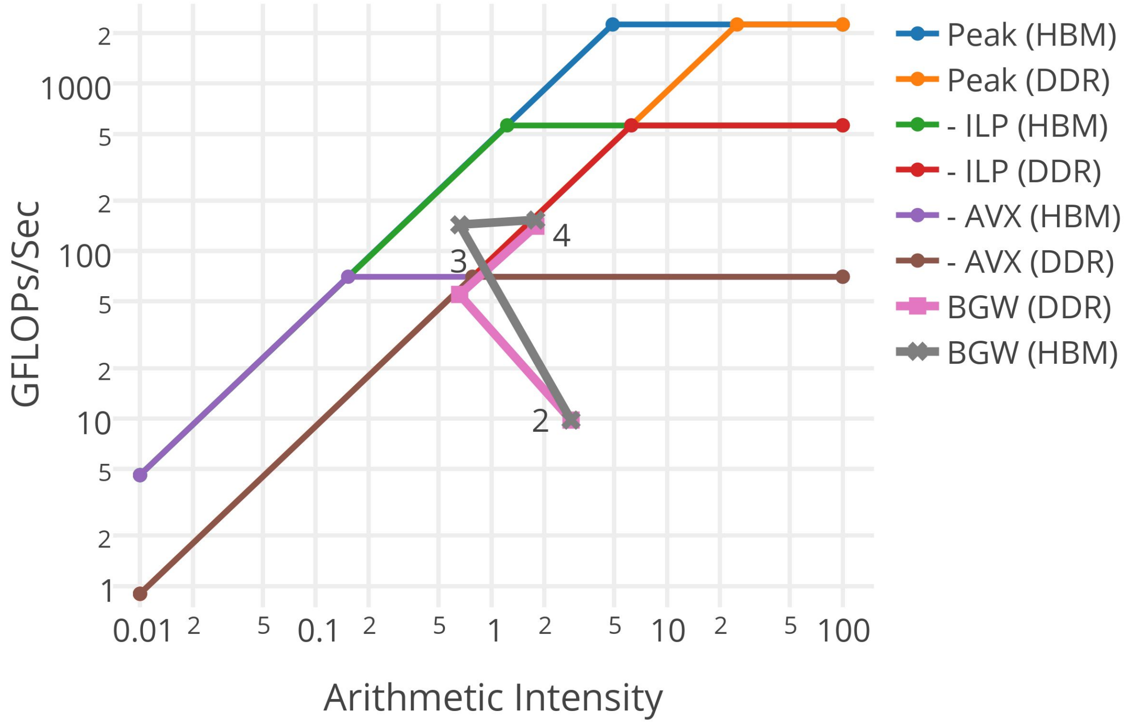
L2 on KNL is 512 KB per core
L2 on Has. is 256 KB per core

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.

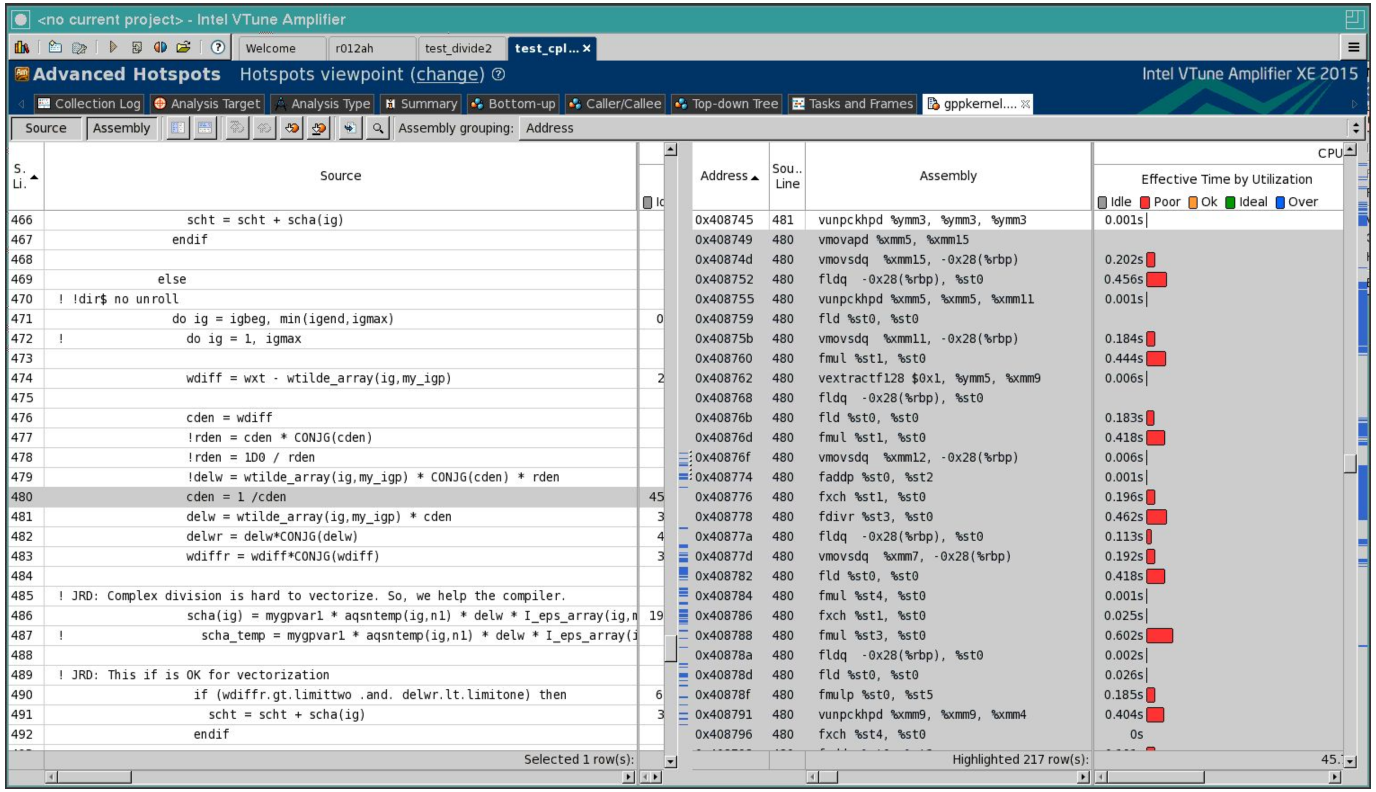
Cache Blocking Optimization


Haswell Roofline Optimization Path


KNL Roofline Optimization Path

Why Complex Divides so Slow?

Found significant x87 instructions from 1/complex_number instead of AVX/AVX-512



```
S.
Li.   Source
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):
```

```
                                                    CPU
Address   Sou..                                     Effective Time by Utilization
          Line   Assembly                           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.
```

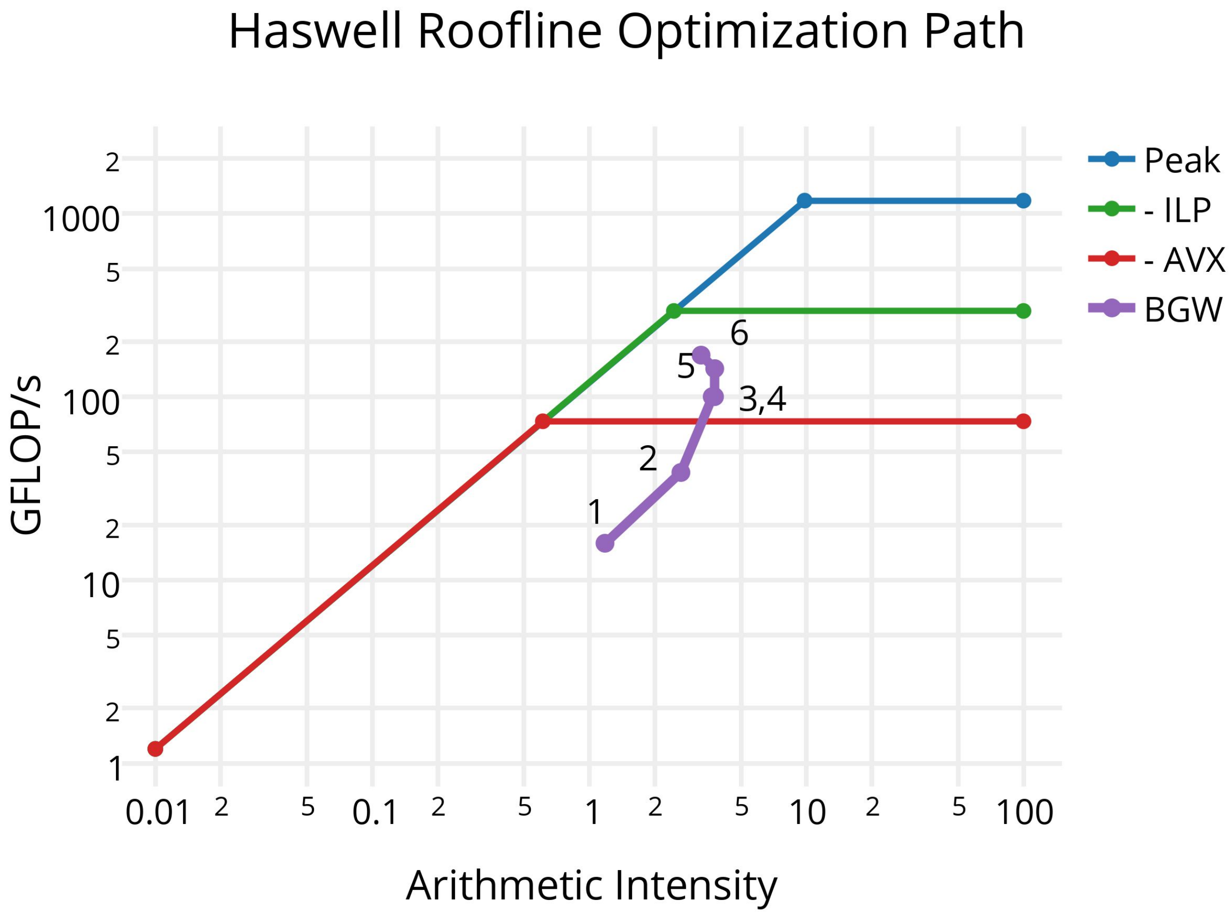
Can significantly speed up by

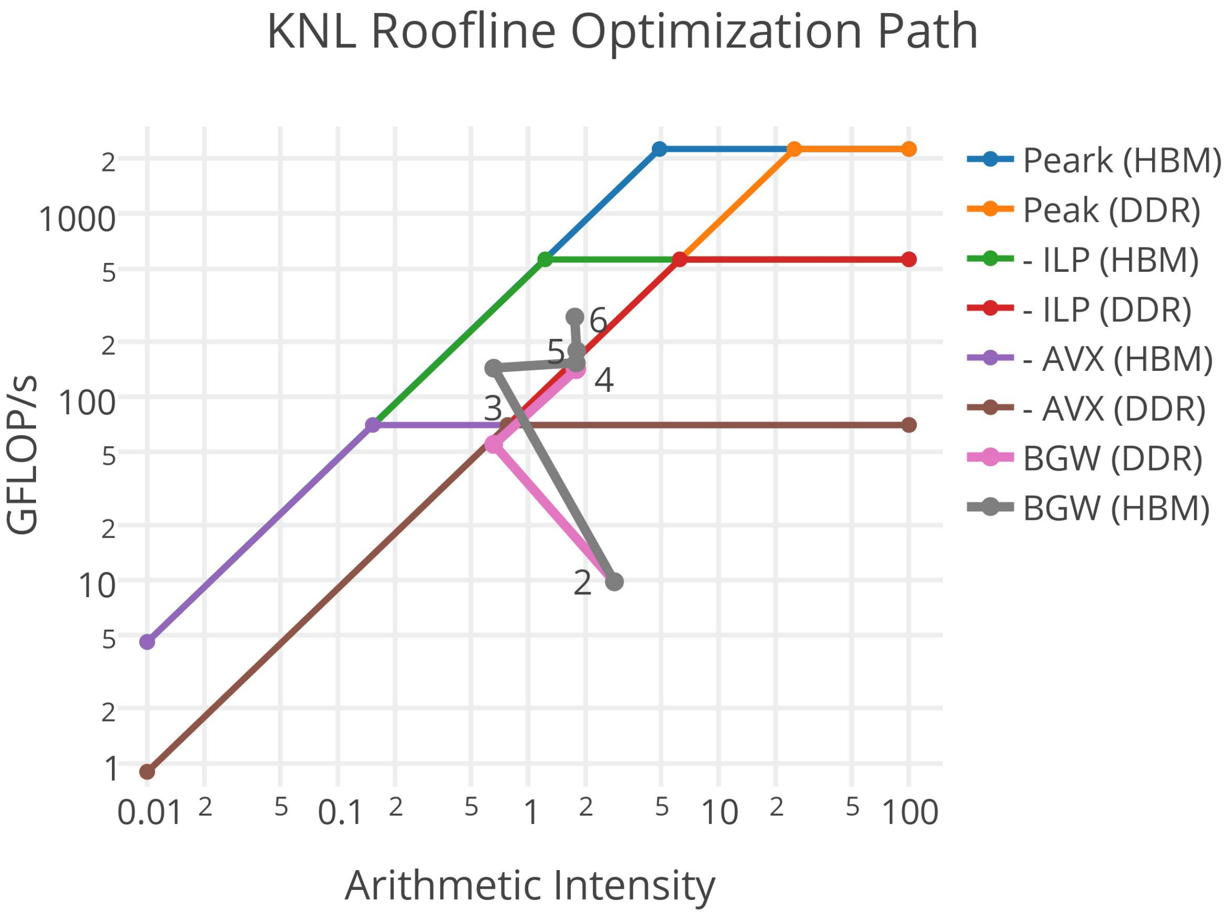
a) Doing complex divide manually

Or

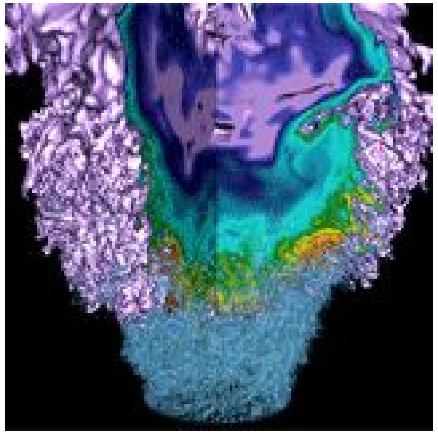
b) Using -fp-model fast=2

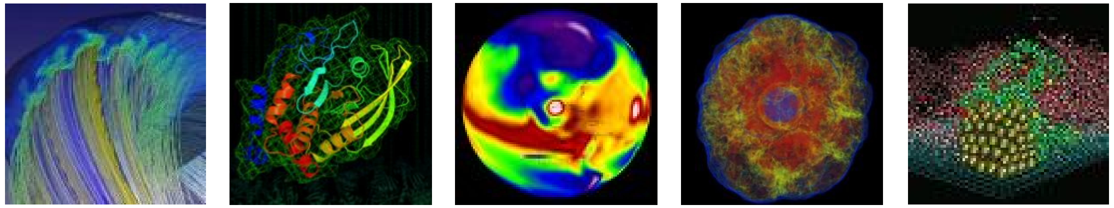
Additional Speedups from Hyperthreading





Conclusions





NERSC 40 YEARS at the FOREFRONT

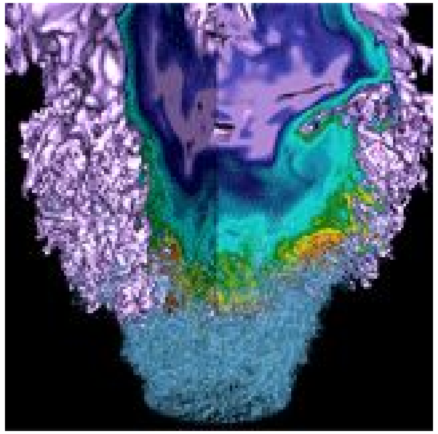
High Level Lessons

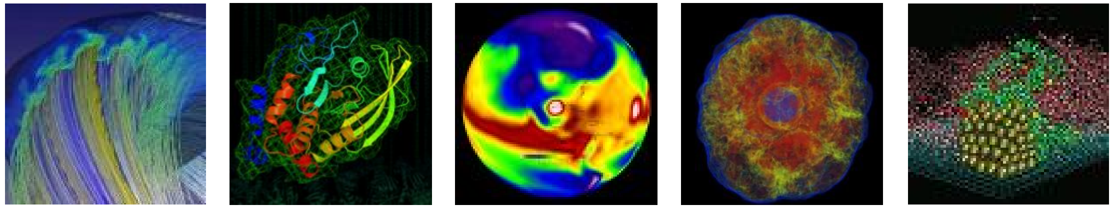
1. Optimizing code is not always straightforward. It is a continual discovery process that involves many sequential and coupled changes.

2. Use profiling tools to find and characterize hotspots.

3. Understanding bandwidth and compute limitations of hotspots are key to deciding how to improve code.

The End (Extra Slides)





NERSC 40 YEARS at the FOREFRONT

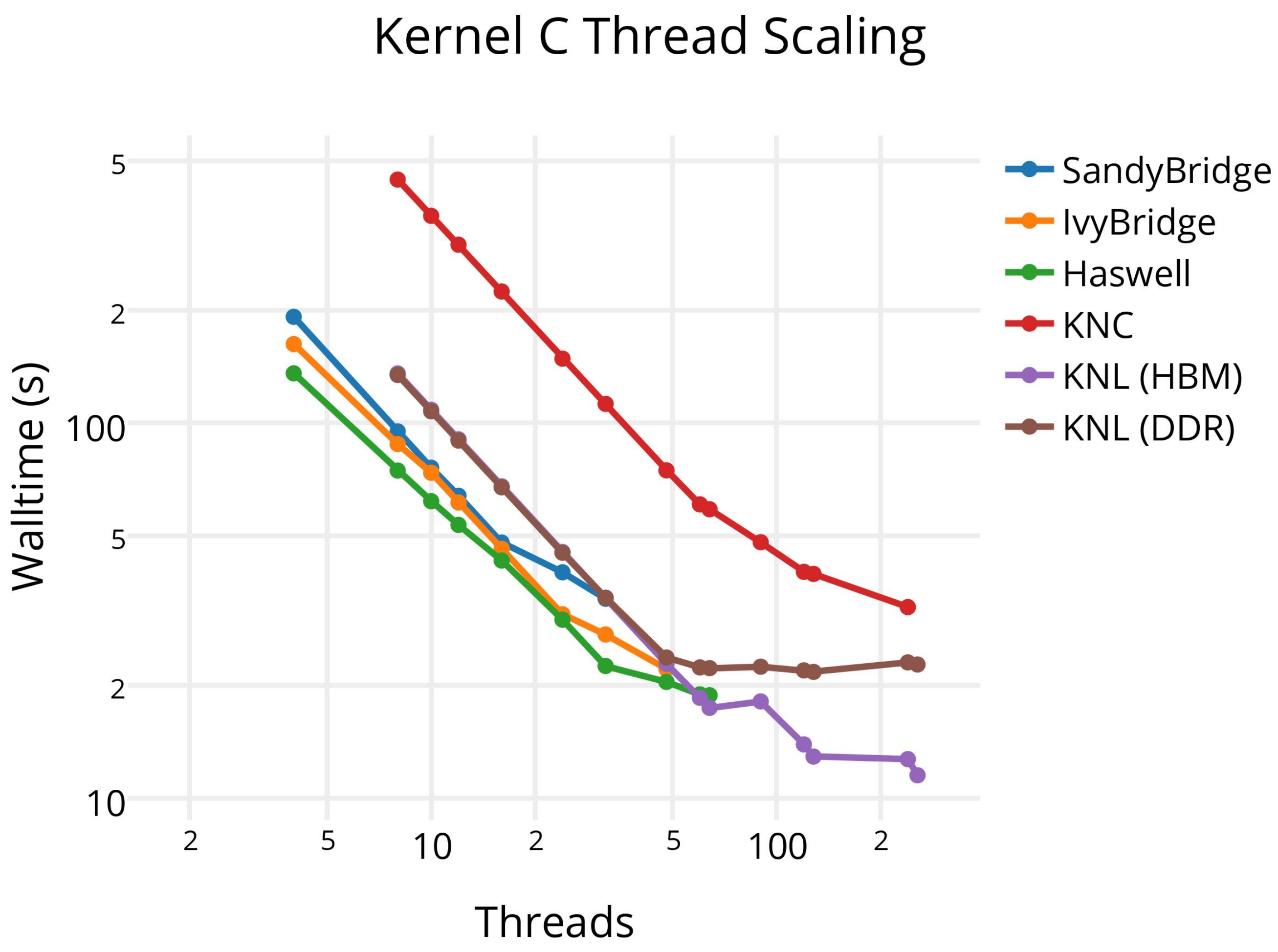
U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Thread Scaling

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

KNL MCDRAM performance beats dual socket Haswell by 63%.



Why Complex Divides so Slow?

Code performance now limited by complex divides

why??

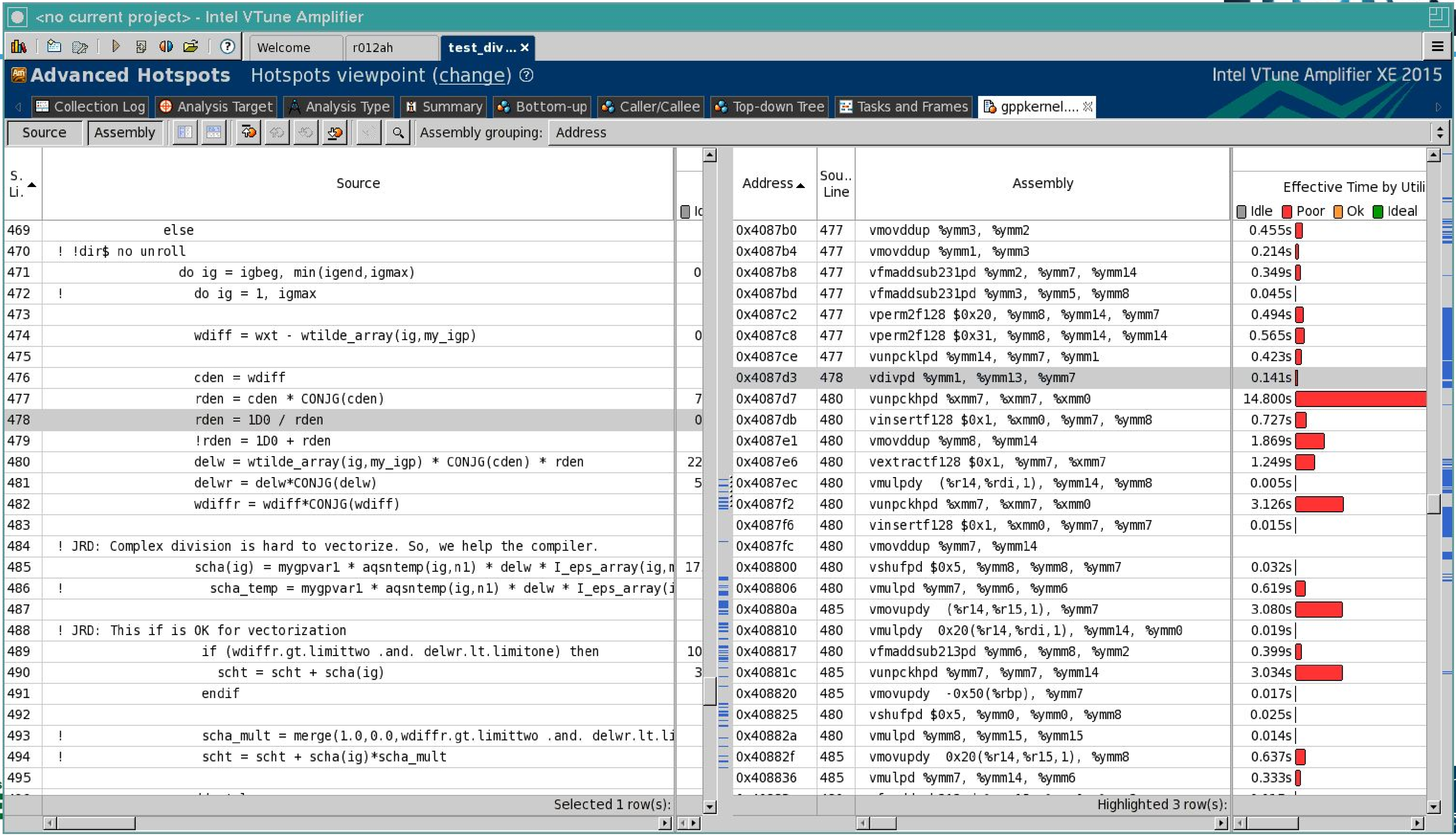
For complex division in performance critical loop, I had already removed the explicit complex divide but what is faster?

a) `c =1 / c` vs. b)

```
r = c * conjg(c)
r = 1 / r
c = conjg(c) * r
```

c/d) Compiling with/without -fp-model fast=2

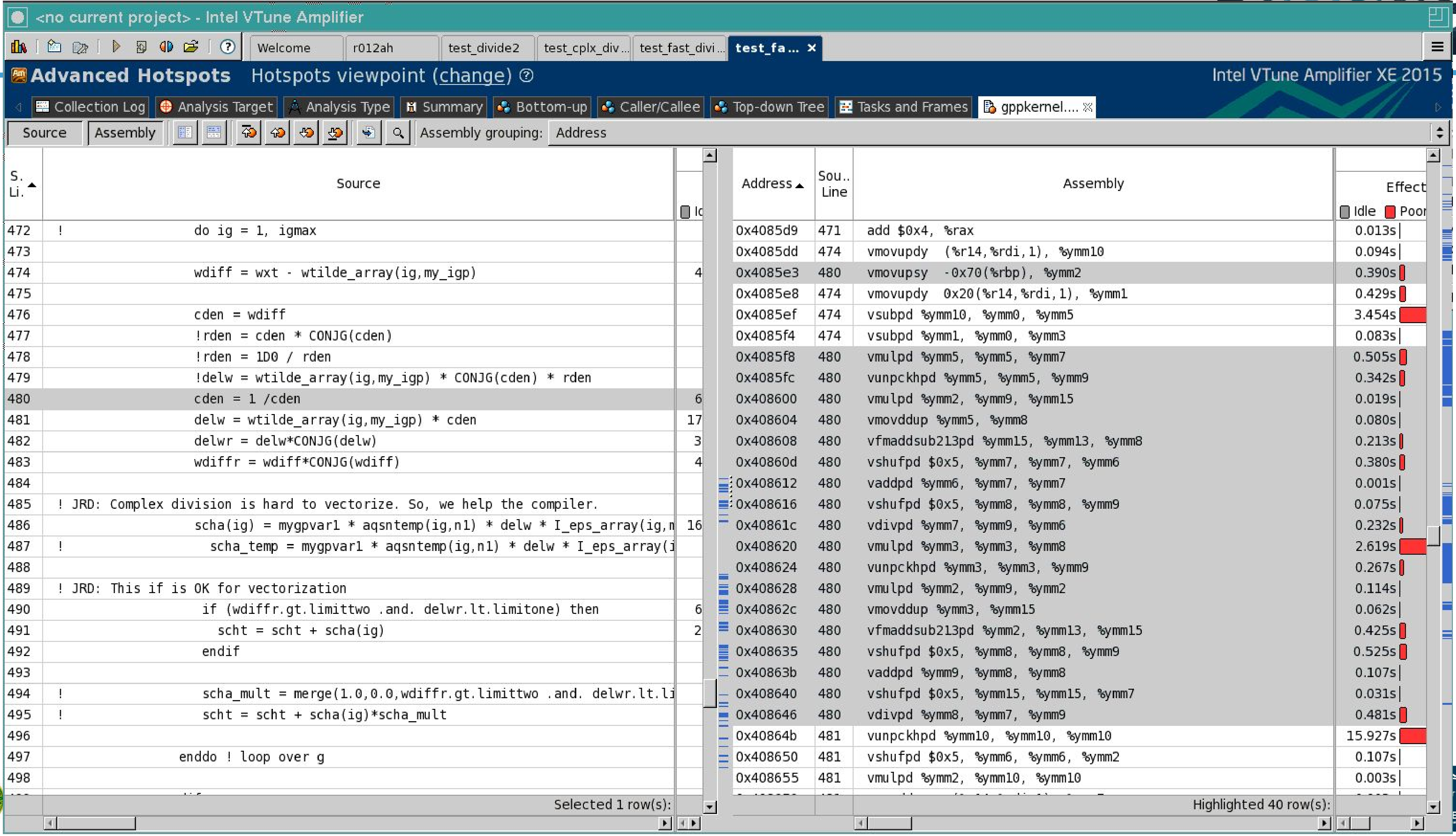
Real-Division (with or without -fp model fast=2)



```
S.   Source                                                                     Id
Li.
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)                 0
475
476                       cden = wdiff
477                       rden = cden * CONJG(cden)                             7
478                       rden = 1D0 / rden                                     0
479                       !rden = 1D0 + rden
480                       delw = wtilde_array(ig,my_igp) * CONJG(cden) * rden   22
481                       delwr = delw*CONJG(delw)                              5
482                       wdiffr = wdiff*CONJG(wdiff)
483
484   ! JRD: Complex division is hard to vectorize. So, we help the compiler.
485                       scha(ig) = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,n 17.
486   !                     scha_temp = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(i
487
488   ! JRD: This if is OK for vectorization
489                        if (wdiffr.gt.limittwo .and. delwr.lt.limitone) then  10
490                          scht = scht + scha(ig)                             3
491                        endif
492
493   !                    scha_mult = merge(1.0,0.0,wdiffr.gt.limittwo .and. delwr.lt.li
494   !                    scht = scht + scha(ig)*scha_mult
495
                                                           Selected 1 row(s):
```

Address	Sou.. Line	Assembly	Effective Time by Utili ▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal
0x4087b0	477	vmovddup %ymm3, %ymm2	0.455s
0x4087b4	477	vmovddup %ymm1, %ymm3	0.214s
0x4087b8	477	vfmaddsub231pd %ymm2, %ymm7, %ymm14	0.349s
0x4087bd	477	vfmaddsub231pd %ymm3, %ymm5, %ymm8	0.045s
0x4087c2	477	vperm2f128 $0x20, %ymm8, %ymm14, %ymm7	0.494s
0x4087c8	477	vperm2f128 $0x31, %ymm8, %ymm14, %ymm14	0.565s
0x4087ce	477	vunpcklpd %ymm14, %ymm7, %ymm1	0.423s
0x4087d3	478	vdivpd %ymm1, %ymm13, %ymm7	0.141s
0x4087d7	480	vunpckhpd %xmm7, %xmm7, %xmm0	14.800s
0x4087db	480	vinsertf128 $0x1, %xmm0, %ymm7, %ymm8	0.727s
0x4087e1	480	vmovddup %ymm8, %ymm14	1.869s
0x4087e6	480	vextractf128 $0x1, %ymm7, %xmm7	1.249s
0x4087ec	480	vmulpdy (%r14,%rdi,1), %ymm14, %ymm8	0.005s
0x4087f2	480	vunpckhpd %xmm7, %xmm7, %xmm0	3.126s
0x4087f6	480	vinsertf128 $0x1, %xmm0, %ymm7, %ymm7	0.015s
0x4087fc	480	vmovddup %ymm7, %ymm14	
0x408800	480	vshufpd $0x5, %ymm8, %ymm8, %ymm7	0.032s
0x408806	480	vmulpd %ymm7, %ymm6, %ymm6	0.619s
0x40880a	485	vmovupdy (%r14,%r15,1), %ymm7	3.080s
0x408810	480	vmulpdy 0x20(%r14,%rdi,1), %ymm14, %ymm0	0.019s
0x408817	480	vfmaddsub213pd %ymm6, %ymm8, %ymm2	0.399s
0x40881c	485	vunpckhpd %ymm7, %ymm7, %ymm14	3.034s
0x408820	485	vmovupdy -0x50(%rbp), %ymm7	0.017s
0x408825	480	vshufpd $0x5, %ymm0, %ymm0, %ymm8	0.025s
0x40882a	480	vmulpd %ymm8, %ymm15, %ymm15	0.014s
0x40882f	485	vmovupdy 0x20(%r14,%r15,1), %ymm8	0.637s
0x408836	485	vmulpd %ymm7, %ymm14, %ymm6	0.333s
		Highlighted 3 row(s):	

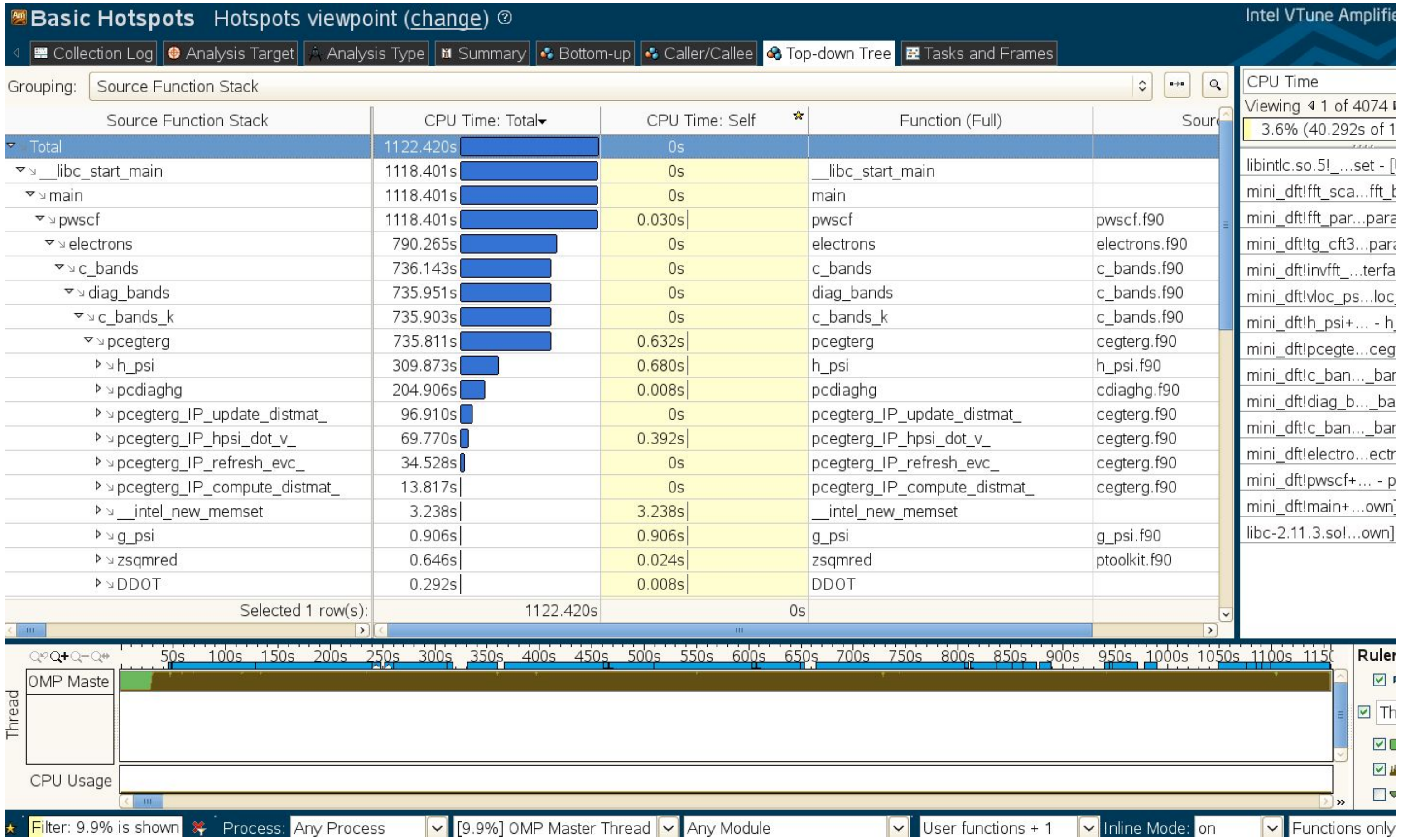
Complex-Division (with -fp model fast=2)



S. Li.	Source	I
472	`!                    do ig = 1, igmax`	
473		
474	`                      wdiff = wxt - wtilde_array(ig,my_igp)`	4
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`	6
481	`                      delw = wtilde_array(ig,my_igp) * cden`	17
482	`                      delwr = delw*CONJG(delw)`	3
483	`                      wdiffr = wdiff*CONJG(wdiff)`	4
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`	16
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)`	2
492	`                       endif`	
493		
494	`!                      scha_mult = merge(1.0,0.0,wdiffr.gt.limittwo .and. delwr.lt.li`	
495	`!                      scht = scht + scha(ig)*scha_mult`	
496		
497	`                    enddo ! loop over g`	
498		

Address	Sou.. Line	Assembly	Effect Idle Poor
0x4085d9	471	add $0x4, %rax	0.013s
0x4085dd	474	vmovupdy (%r14,%rdi,1), %ymm10	0.094s
0x4085e3	480	vmovupsy -0x70(%rbp), %ymm2	0.390s
0x4085e8	474	vmovupdy 0x20(%r14,%rdi,1), %ymm1	0.429s
0x4085ef	474	vsubpd %ymm10, %ymm0, %ymm5	3.454s
0x4085f4	474	vsubpd %ymm1, %ymm0, %ymm3	0.083s
0x4085f8	480	vmulpd %ymm5, %ymm5, %ymm7	0.505s
0x4085fc	480	vunpckhpd %ymm5, %ymm5, %ymm9	0.342s
0x408600	480	vmulpd %ymm2, %ymm9, %ymm15	0.019s
0x408604	480	vmovddup %ymm5, %ymm8	0.080s
0x408608	480	vfmaddsub213pd %ymm15, %ymm13, %ymm8	0.213s
0x40860d	480	vshufpd $0x5, %ymm7, %ymm7, %ymm6	0.380s
0x408612	480	vaddpd %ymm6, %ymm7, %ymm7	0.001s
0x408616	480	vshufpd $0x5, %ymm8, %ymm8, %ymm9	0.075s
0x40861c	480	vdivpd %ymm7, %ymm9, %ymm6	0.232s
0x408620	480	vmulpd %ymm3, %ymm3, %ymm8	2.619s
0x408624	480	vunpckhpd %ymm3, %ymm3, %ymm9	0.267s
0x408628	480	vmulpd %ymm2, %ymm9, %ymm2	0.114s
0x40862c	480	vmovddup %ymm3, %ymm15	0.062s
0x408630	480	vfmaddsub213pd %ymm2, %ymm13, %ymm15	0.425s
0x408635	480	vshufpd $0x5, %ymm8, %ymm8, %ymm9	0.525s
0x40863b	480	vaddpd %ymm9, %ymm8, %ymm8	0.107s
0x408640	480	vshufpd $0x5, %ymm15, %ymm15, %ymm7	0.031s
0x408646	480	vdivpd %ymm8, %ymm7, %ymm9	0.481s
0x40864b	481	vunpckhpd %ymm10, %ymm10, %ymm10	15.927s
0x408650	481	vshufpd $0x5, %ymm6, %ymm6, %ymm2	0.107s
0x408655	481	vmulpd %ymm2, %ymm10, %ymm10	0.003s

Profile Your Application (VTune / CrayPat)

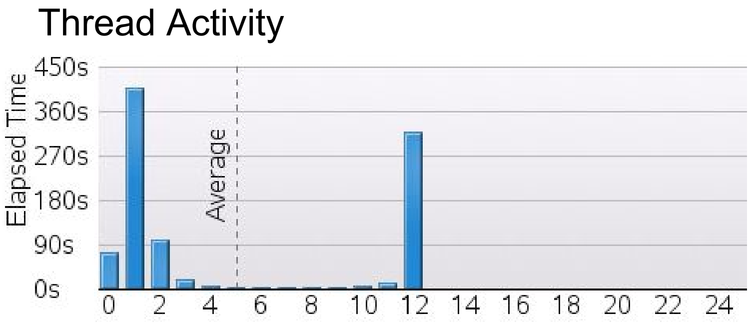


Grouping: Source Function Stack

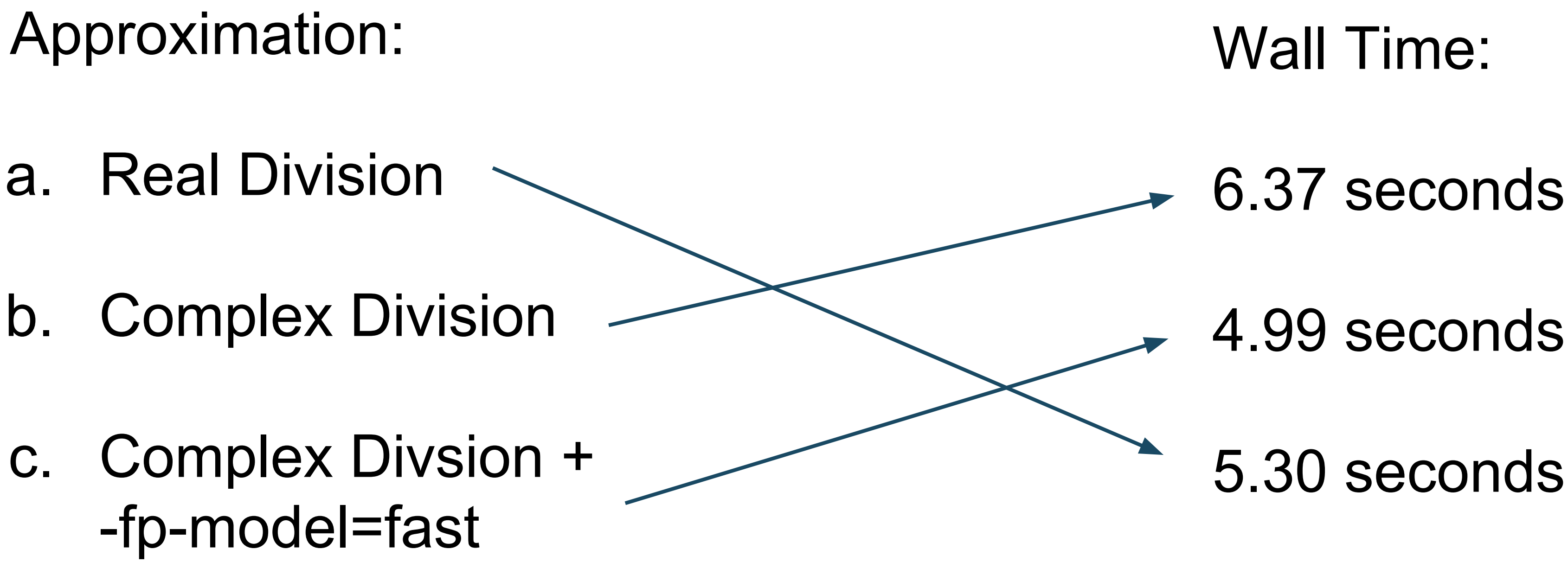
Source Function Stack	CPU Time: Total	CPU Time: Self	Function (Full)	Sour
Total	1122.420s	0s		
__libc_start_main	1118.401s	0s	__libc_start_main	
main	1118.401s	0s	main	
pwscf	1118.401s	0.030s	pwscf	pwscf.f90
electrons	790.265s	0s	electrons	electrons.f90
c_bands	736.143s	0s	c_bands	c_bands.f90
diag_bands	735.951s	0s	diag_bands	c_bands.f90
c_bands_k	735.903s	0s	c_bands_k	c_bands.f90
pcegterg	735.811s	0.632s	pcegterg	cegterg.f90
h_psi	309.873s	0.680s	h_psi	h_psi.f90
pcdiaghg	204.906s	0.008s	pcdiaghg	cdiaghg.f90
pcegterg_IP_update_distmat_	96.910s	0s	pcegterg_IP_update_distmat_	cegterg.f90
pcegterg_IP_hpsi_dot_v_	69.770s	0.392s	pcegterg_IP_hpsi_dot_v_	cegterg.f90
pcegterg_IP_refresh_evc_	34.528s	0s	pcegterg_IP_refresh_evc_	cegterg.f90
pcegterg_IP_compute_distmat_	13.817s	0s	pcegterg_IP_compute_distmat_	cegterg.f90
__intel_new_memset	3.238s	3.238s	__intel_new_memset	
g_psi	0.906s	0.906s	g_psi	g_psi.f90
zsqmred	0.646s	0.024s	zsqmred	ptoolkit.f90
DDOT	0.292s	0.008s	DDOT	
Selected 1 row(s):	1122.420s	0s		

CPU Time
Viewing ◂ 1 of 4074 ▸
3.6% (40.292s of 1
libintlc.so.5!__set - [
mini_dft!fft_sca…fft_b
mini_dft!fft_par…para
mini_dft!tg_cft3…para
mini_dft!invfft_…terfa
mini_dft!vloc_ps…loc_
mini_dft!h_psi+… - h_
mini_dft!pcegte…ceg
mini_dft!c_ban…_bar
mini_dft!diag_b…_ba
mini_dft!c_ban…_bar
mini_dft!electro…ectr
mini_dft!pwscf+… - p
mini_dft!main+…own]
libc-2.11.3.so!…own]

Filter: 9.9% is shown | Process: Any Process | [9.9%] OMP Master Thread | Any Module | User functions + 1 | Inline Mode: on | Functions only



Approximation:		Wall Time:
a. Real Division		6.37 seconds
b. Complex Division	?	4.99 seconds
c. Complex Division + -fp-model fast=2		5.30 seconds



Approximation:

a. Real Division

b. Complex Division

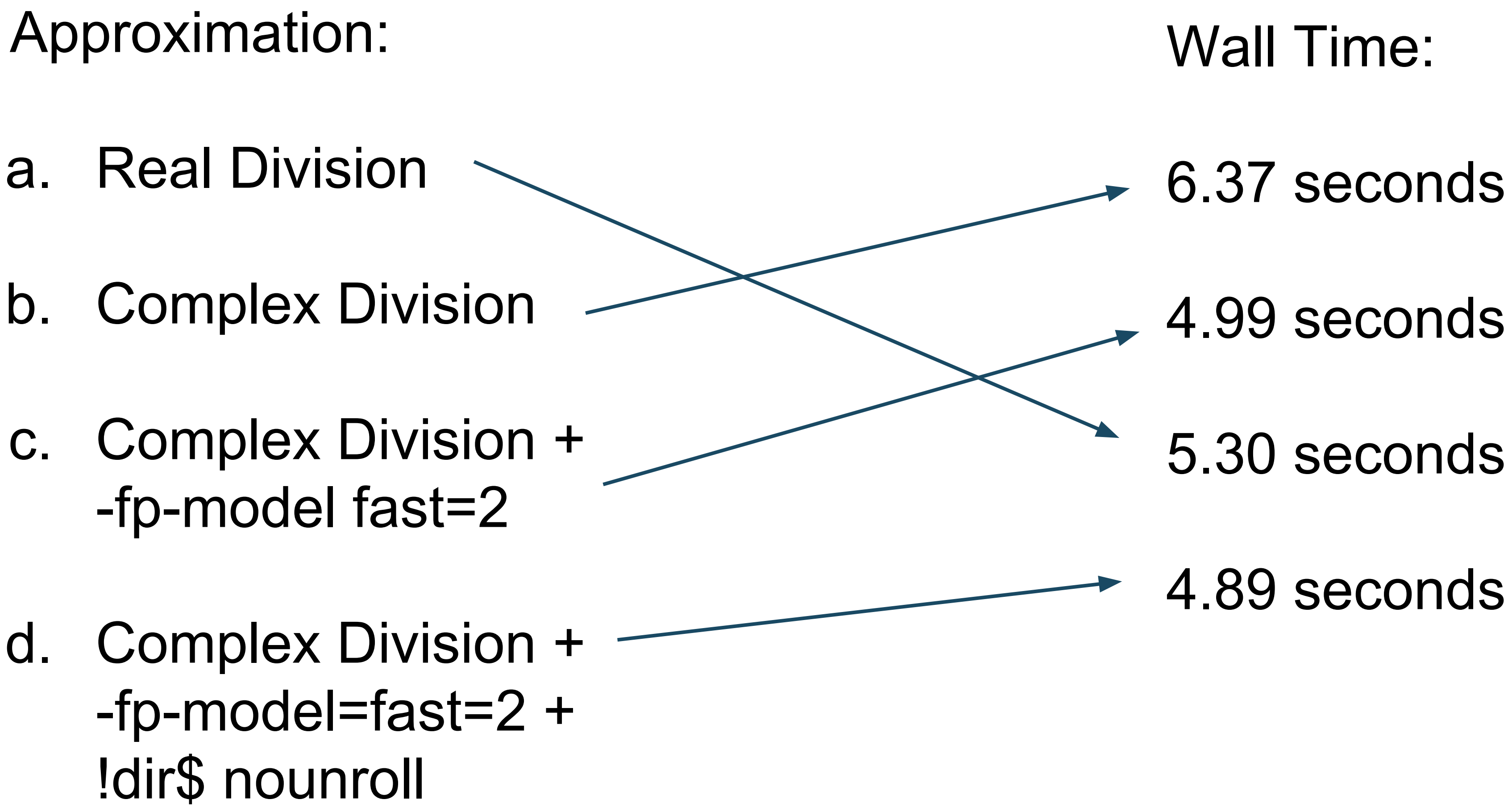
c. Complex Divsion + -fp-model=fast

Wall Time:

6.37 seconds

4.99 seconds

5.30 seconds



Approximation:

a. Real Division

b. Complex Division

c. Complex Division +
-fp-model fast=2

d. Complex Division +
-fp-model=fast=2 +
!dir$ nounroll

Wall Time:

6.37 seconds

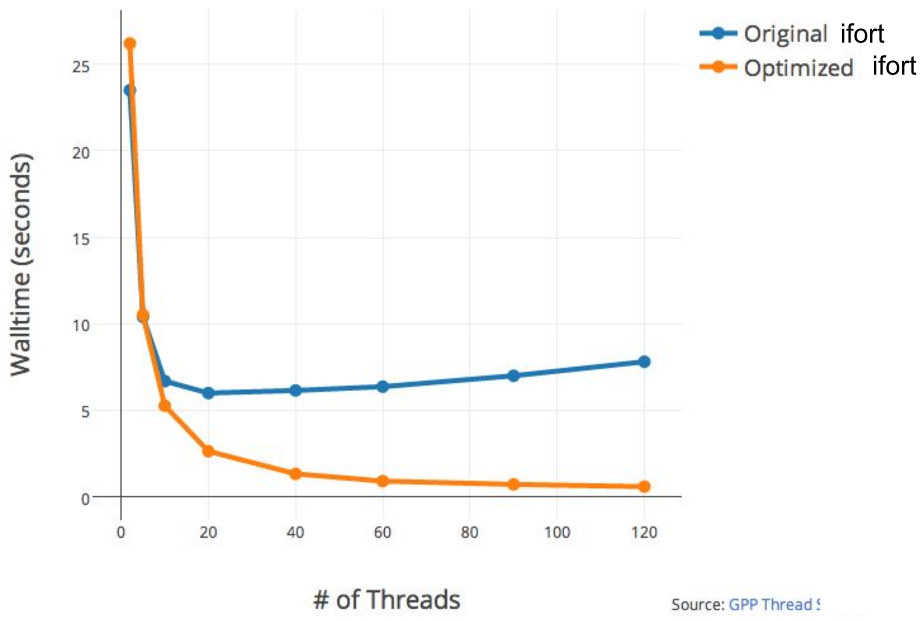
4.99 seconds

5.30 seconds

4.89 seconds

Early NESAP (Advances with Cray and Intel) Advances

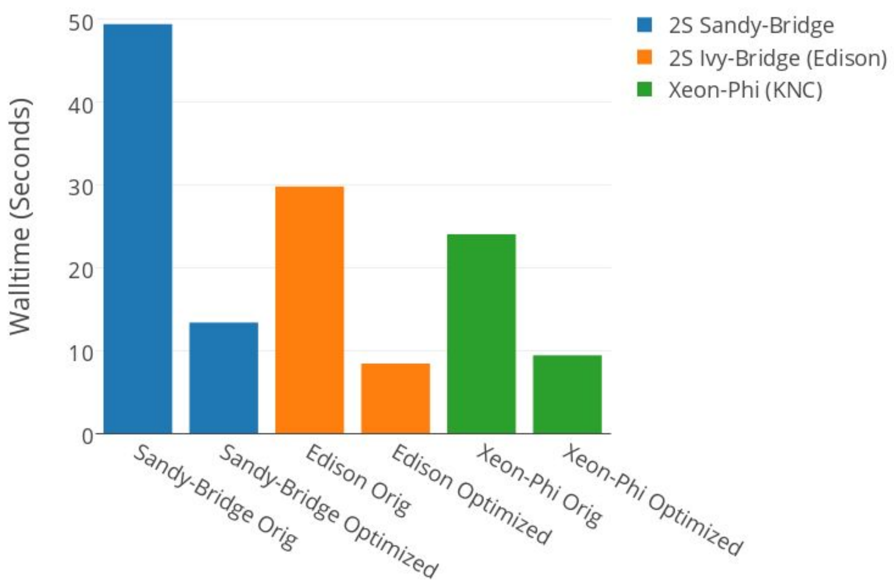
Thread Scaling in BerkeleyGW GPP Kernel on Xeon-Phi



Source: GPP Thread S

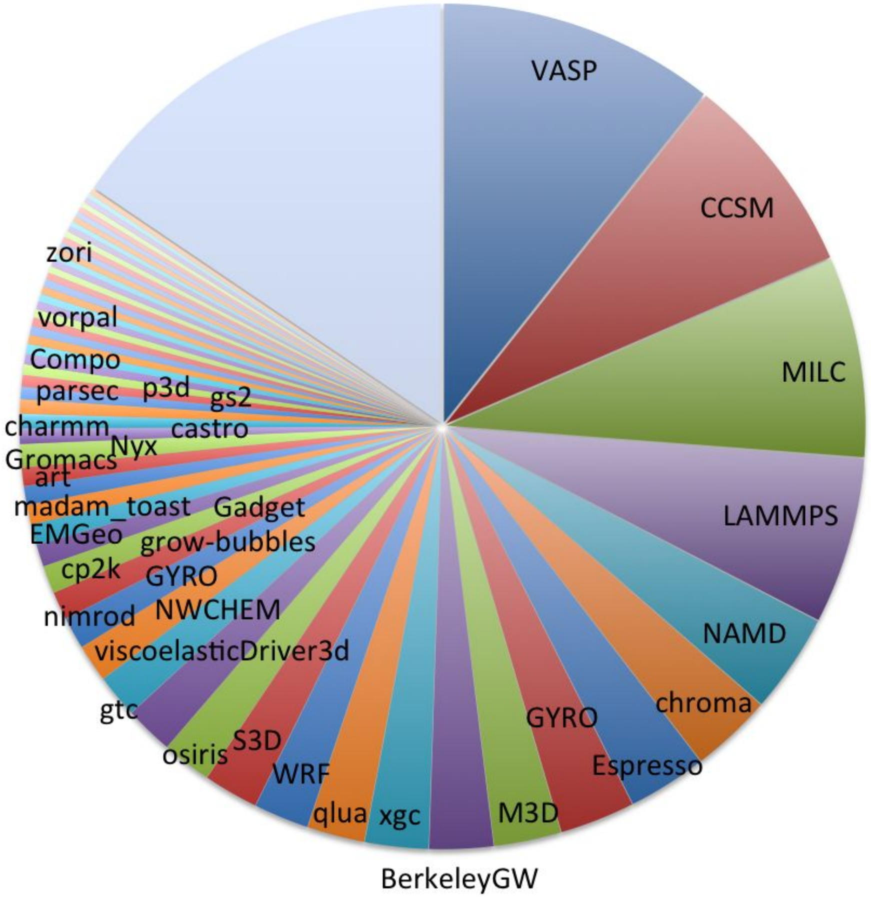
BerkeleyGW FF Kernel Runtimes on Xeon and Xeon-Phi

(Nathan)

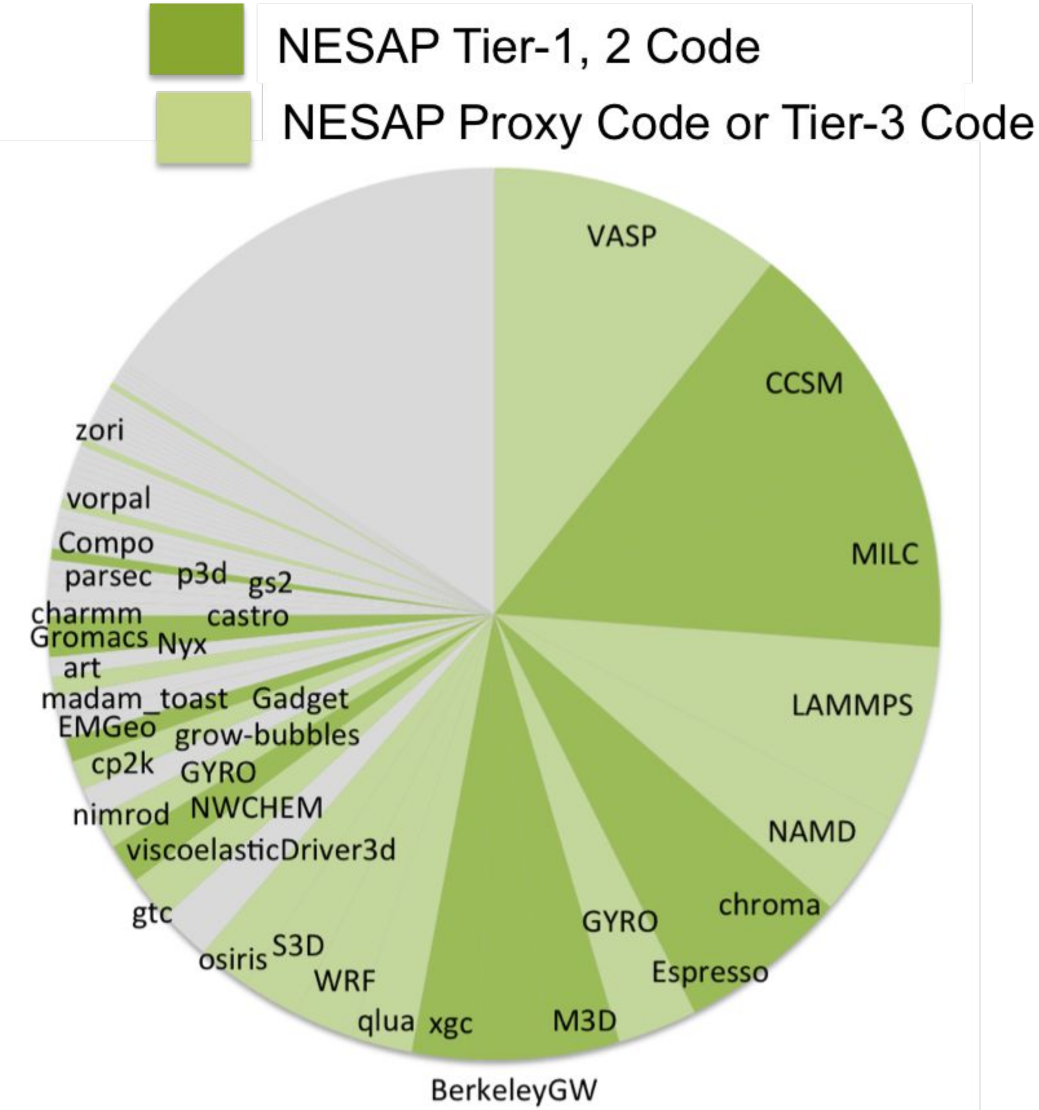


	Overall Improvement	Notes
BGW GPP Kernel	0-10%	Pretty optimized to begin with. Thread scalability improved by fixing ifort allocation performance.
BGW FF Kernel	2x-4x	Unoptimized to begin with. Cache reuse improvements
BGW Chi Kernel	10-30%	Moved threaded region outward in code
BGW BSE Kernel	10-50%	Created custom vector matmuls

NESAP



Breakdown of Application Hours on Hopper and Edison 2013



Early Lessons Learned

Cray and Intel very helpful in profiling/optimizing the code. See following slides for using Intel resources effectively

Generating small tangible kernels is important for succes

Targeting Many-Core greatly helps performance back on Xeon.

Complex division is slow on (particularly on KNC)