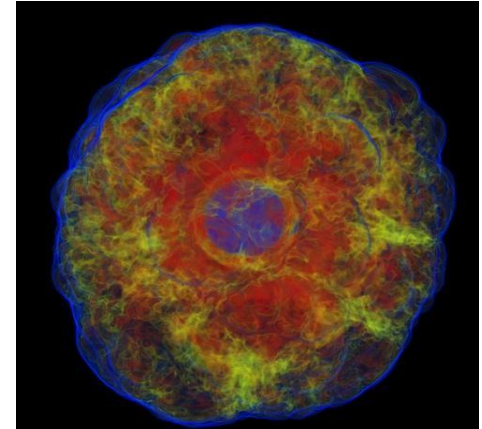
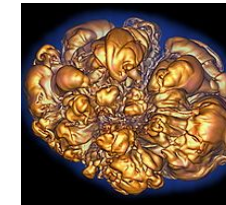
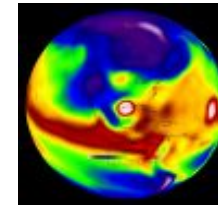
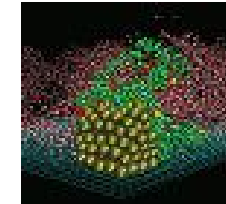
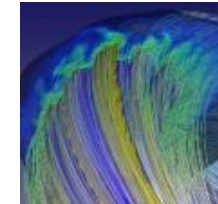
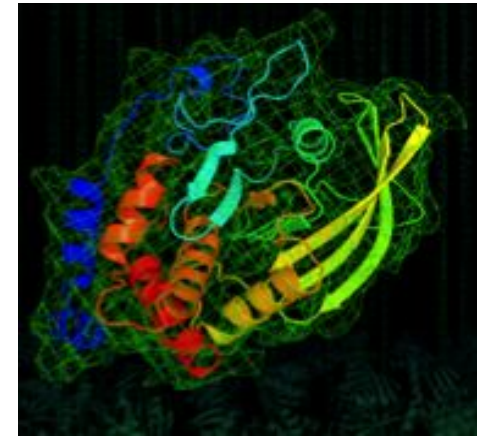
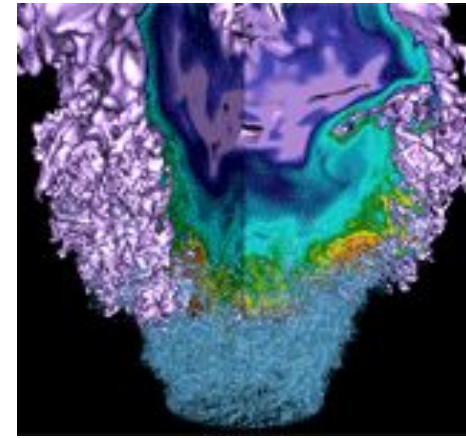


# NERSC, Cori, Knights Landing, And Other Matters



Jack Deslippe  
February, 2016

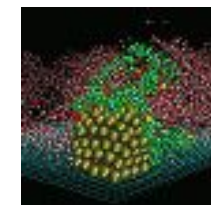
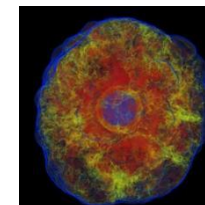
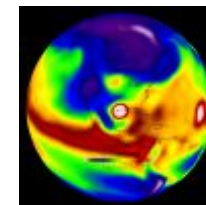
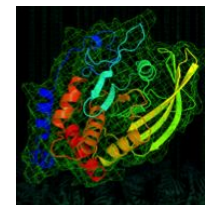
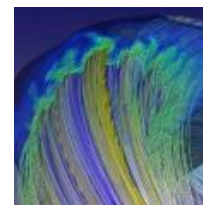
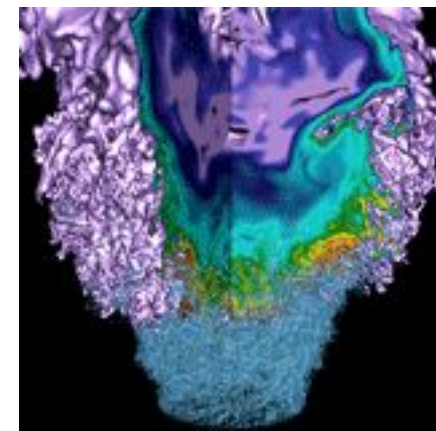
# Agenda

---



1. Overview of NERSC
2. Things I wished I knew back when I took CS267
3. Optimizing Applications For Cori Phase 2
4. Example Case Study

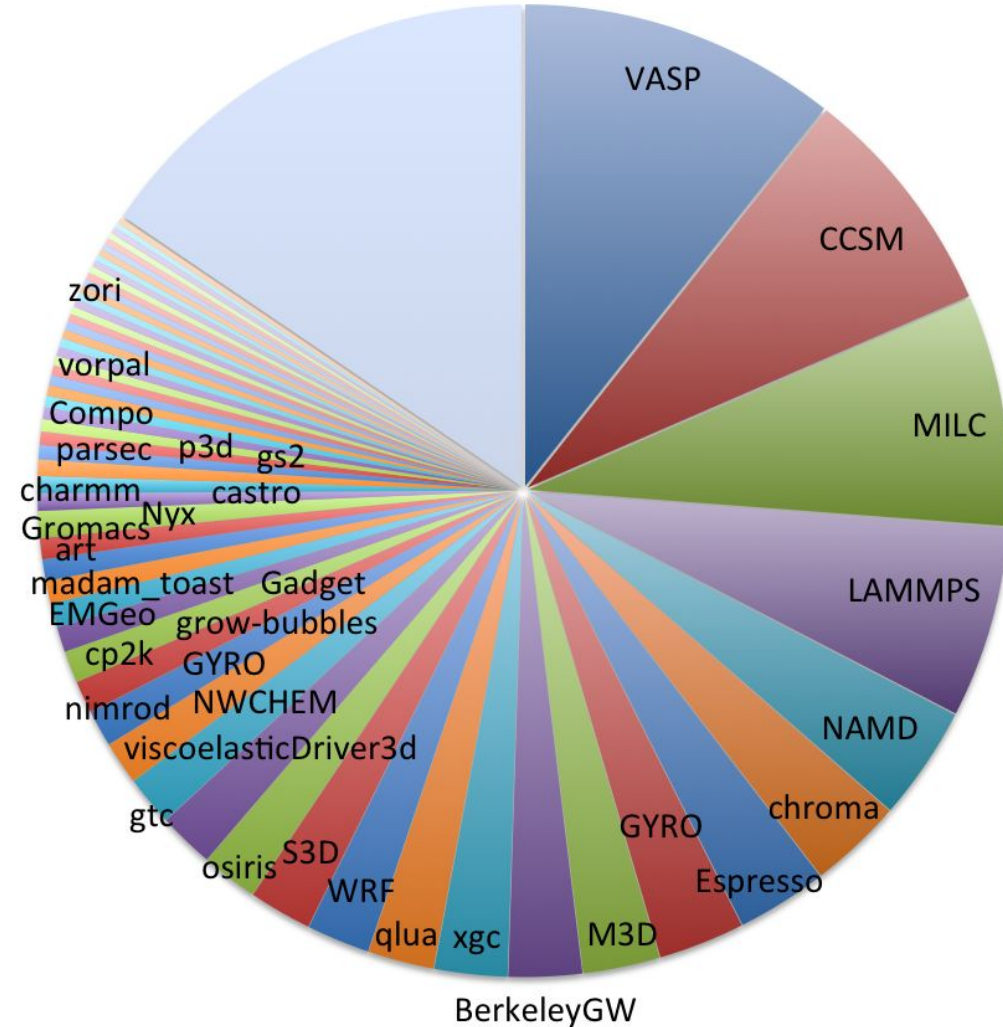
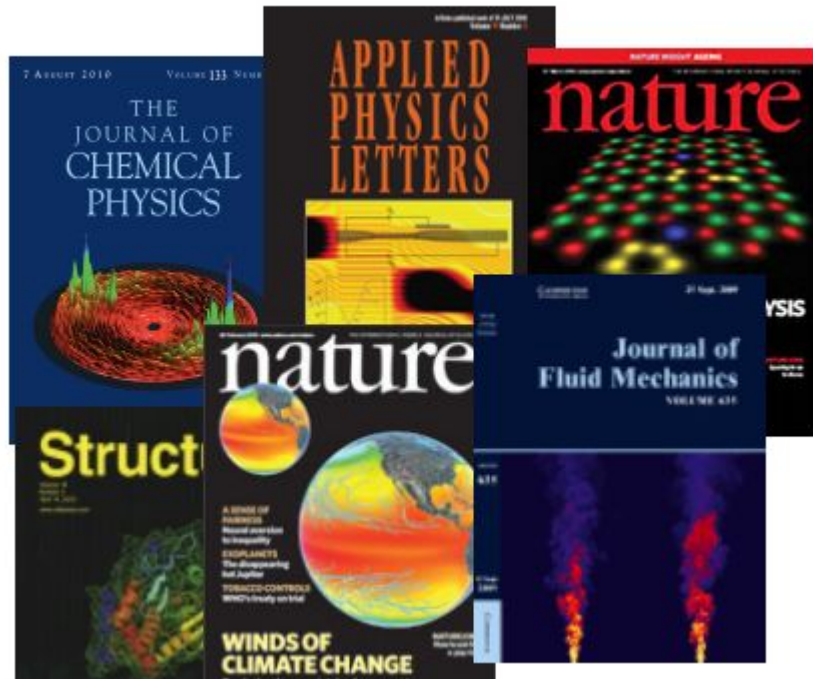
# NERSC



# NERSC is the production HPC center for the DOE Office of Science



- 5000 users, 600 projects
  - From 48 states; 65% from universities
  - Hundreds of users each day
  - 1500 publications per year
- Systems designed for science



# The NERSC-8 System: Cori



- Cori will support the broad Office of Science research community and begin to transition the workload to more energy efficient architectures
- Cray XC system with over 9,300 Intel Knights Landing compute nodes – mid 2016
  - Self-hosted, (not an accelerator) manycore processor with up to 72 cores per node
  - On-package high-bandwidth memory
- **Data Intensive Science Support**
  - 10 Haswell processor cabinets (Phase 1) to support data intensive applications – Summer 2015
  - NVRAM Burst Buffer to accelerate data intensive applications
  - 28 PB of disk, >700 GB/sec I/O bandwidth
- **Robust Application Readiness Plan**
  - Outreach and training for user community
  - Application deep dives with Intel and Cray



# NERSC's Current Big System is Edison



- Edison is the HPCS\* demo system (serial #1)
- First Cray Petascale system with Intel processors (Ivy Bridge), Aries interconnect topology
- Very high memory bandwidth (100 GB/s per node)
- 5,576 nodes, 133K cores, 64 GB/node
- Exceptional application performance



\*DARPA High Productivity Computing System program

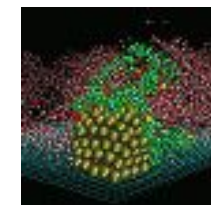
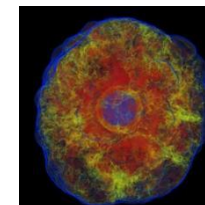
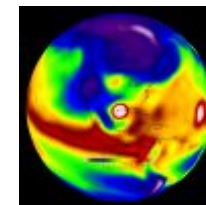
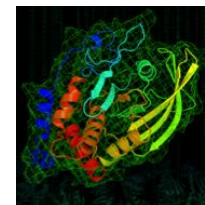
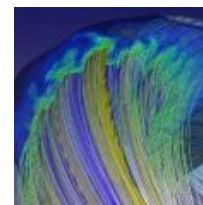
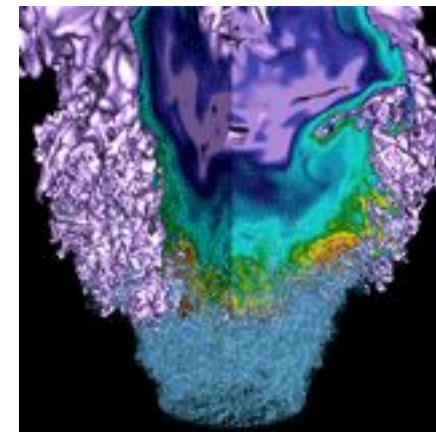
# NERSC moved into Wang Hall late 2015



- **Four story, 140,000 GSF, 300 offices, 20Ksf HPC floor, 12.5->40 MW**
- **Located for collaboration**
  - LBNL, CRD, Esnet, UCB
- **Exceptional energy efficiency**
  - Natural air and water cooling
  - Heat recovery
  - PUE < 1.1



# Things I Wish I Knew When I was In CS267





# Debugging Parallel Programming Bugs



This code hangs because both Task 0 and Task N-1 are blocking on MPI\_Recv

```
if(task_no==0) {  
    ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, totTasks-1, 0, MPI_COMM_WORLD,  
&status);  
    ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, totTasks-1, 0, MPI_COMM_WORLD);  
} else if (task_no==(totTasks-1)) {  
    ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);  
    ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);  
}
```





## FOR USERS

- › Live Status
- › My NERSC
- › Getting Started
- › Computational Systems
- › Data & File Systems
- › Network Connections
  - Connecting to NERSC
  - Using X Windows

### Connecting to NERSC with NX

#### NX FAQ

Download Tested NX Player

NX Configuration File

Startup Tutorial

Transferring Data

Network Performance

› Queues and Scheduling

› Job Logs & Analytics

› Training & Tutorials

› Software

› Accounts & Allocations

› Policies

› Data Analytics & Visualization

› Data Management Policies

Home » For Users » Network Connections » Connecting to NERSC with NX

## NERSC NX SERVICE - X-WINDOWS ACCELERATION AT NERSC

### Introduction

**NX** is a computer program that handles remote X Window System connections and it provides three benefits for NERSC users:

- **SPEED:** NX can greatly improve the performance of X Windows, allowing users with slow, high latency connections (e.g. on cell phone network, traveling in Africa) to use complex X Windows programs (such as rotating a plot in Matlab).
- **SESSION:** NX provides sessions that allow a user to disconnect from the session and reconnect to it at a later time while keeping the state of all running applications inside the session.
- **DESKTOP:** NX gives users a virtual desktop that's running at NERSC. You can customize the desktop according to your work requirement.



### TABLE OF CONTENTS

1. Introduction
2. New NX Service
3. Availability
4. Current Users (Live!)
5. Quick Start Up Tutorial
6. Got A Question?

### Related Information

[Download Tested NX Player](#)

[Download NX Configuration File](#)

[NX FAQ](#)



# Compile & Start DDT



## Compile for debugging

```
edison% make  
cc -c -g hello.c  
cc -o hello -g hello.o
```

## Set up the parallel run environment

```
edison% qsub -I -v -lmpwidth=24  
edison% cd $PBS_O_WORKDIR
```

## Start the DDT debugger

```
edison% module load ddt  
edison% ddt ./hello
```

# DDT Screen Shot



Press Go and then Pause when code appears hung.

Task 0 is at line 44

At hang, tasks are in 3 different places.

The screenshot shows the Allinea DDT v3.1-20 interface. The top toolbar contains a 'Go' button (green play icon) and a 'Pause' button (yellow double bars). The 'Current Group' is set to 'All', and the 'Focus on current' options are 'Group', 'Process', and 'Thread'. The code editor displays the following code:

```
36 ret = MPI_Comm_size( MPI_COMM_WORLD, &totTasks );
37
38 printf("task_no is %6d of %6d total tasks\n",
39        task_no,
40        totTasks);
41
42
43 if(task_no==0) {
44     ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, totTasks-1,
45                 MPI_ANY_SOURCE, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
46 } else if (task_no==(totTasks-1)) {
47     ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, totTasks-1,
48                 MPI_ANY_DEST, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
49 }
50
51
52
```

The 'Stacks' window at the bottom shows three entries:

Processes	Function
1	main (hello.c:44)
1	main (hello.c:47)
2	main (hello.c:54)

# Vendors are starting to listen (DDT)



DDT - Cross-Process Comparison View

Expression: ranno

Processes in current group (All, 24 procs)

Limit comparison to 1 s.f.

Only show if: ranno .gt. .7 [See Examples](#)

Align stack frames

Use as MPI Rank Create Groups Export **Statistics**

Values	Process(es)
0.0499837324	19
0.067494303	6
0.134988606	13
0.152499184	0
0.202482924	20
0.219993487	7
0.287487805	14
0.304998368	1
0.354982108	21
0.372492671	8
0.439986974	15
0.457497567	2
0.507481277	22
0.52499187	9
0.592486143	16
0.609996736	3
0.659980476	23
0.677491069	10
0.744985342	17

Statistics

Count: 24  
Not shown: 0  
Errors: 0  
Aggregate: 0  
Numerical: 24  
Sum: 11.7498  
Minimum: 0.0499837  
Maximum: 0.982489  
Range: 0.932506  
Mean: 0.489573  
Variance: 0.0788268  
nan: 0  
-nan: 0  
inf: 0  
-inf: 0  
<0: 0  
=0: 0

Locals

Variable Name	Value
count	1
heads	99998695
i	200000001
ierror	0
iter	1
j	7
m	34
mpi_argv_null	([1] = '')
mpi_argvs_null	([1] = ([1] = '000'))
mpi_bottom	0
mpi_errcodes_ignore	([1] = 0)
mpi_in_place	0
mpi_statuses_ignore	([1] = 0, ...)
mpi_status_ignore	([1] = 0, ...)
mpi_type	1275070513
mpi_unweighted	0
mype	99
ntimes	200000000
numpes	1200
pct	1.40129846e-45
ranno	0.24998655
root	0
seed	([1] = 100, ...)
totheads	0

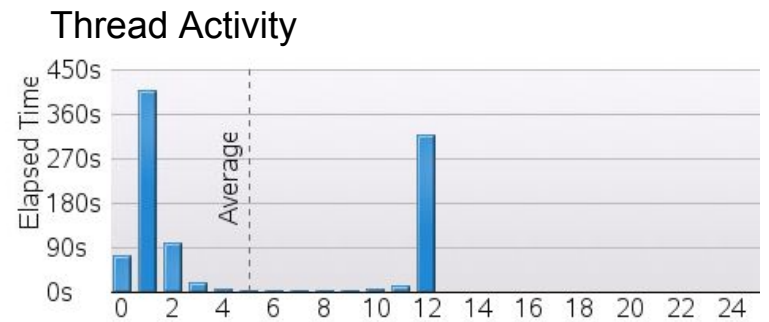
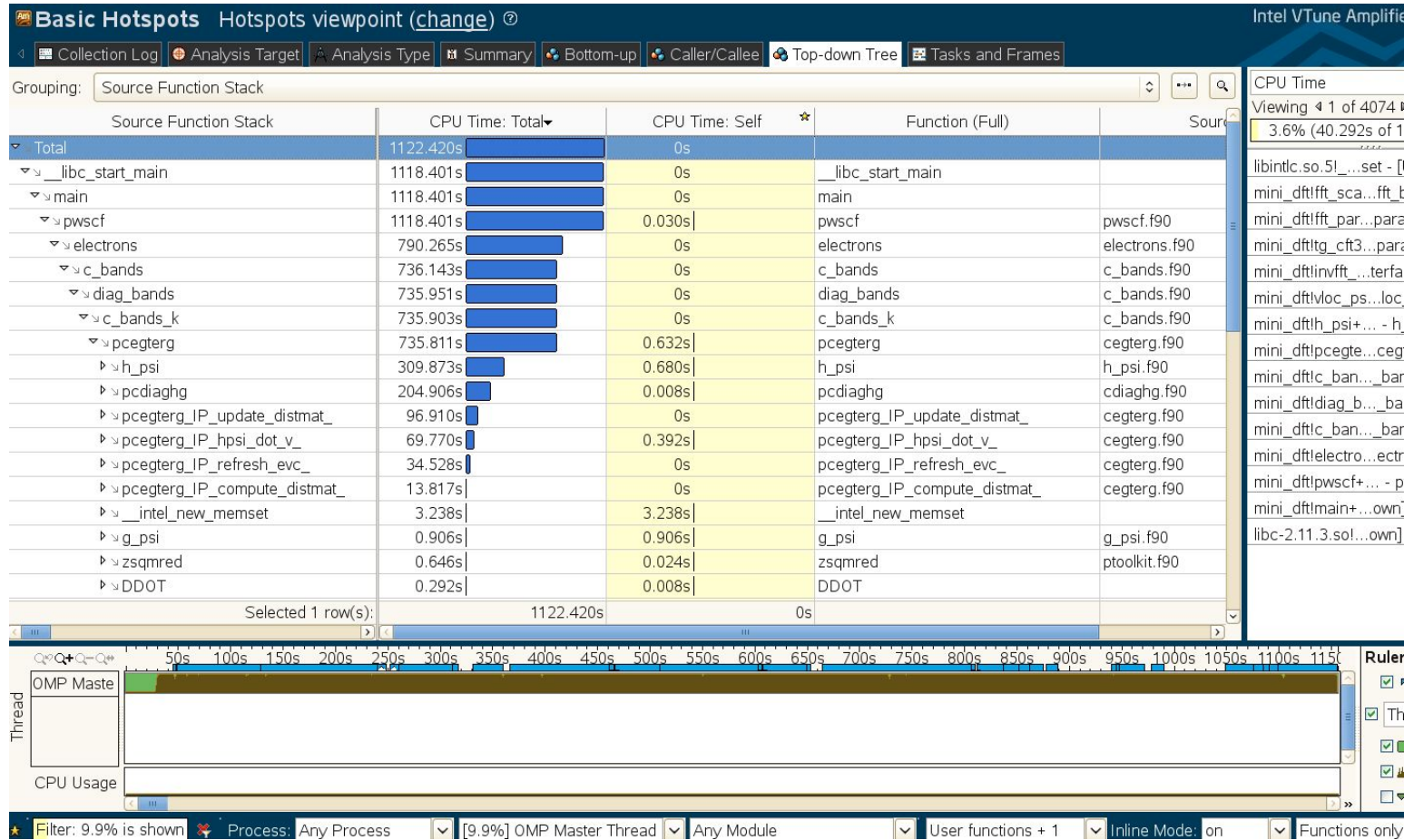
Type: none selected

# Debuggers on NERSC machines

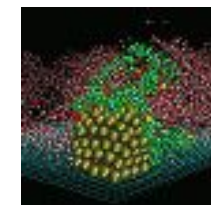
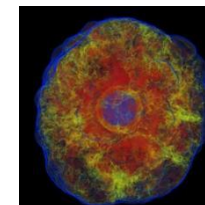
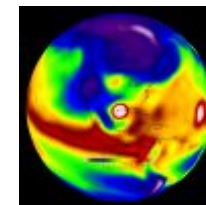
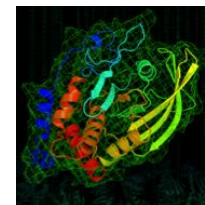
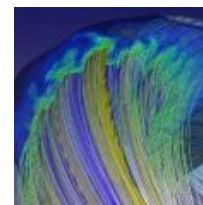
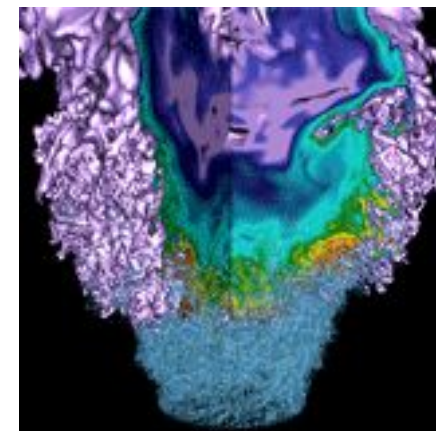


- **Parallel debuggers with a graphical user interface**
  - DDT (Distributed Debugging Tool)
  - TotalView
- **Specialized debuggers on Hopper and Edison**
  - STAT (Stack Trace Analysis Tool)
    - Collect stack backtraces from all (MPI) tasks
  - ATP (Abnormal Termination Processing)
    - Collect stack backtraces from all (MPI) tasks when an application fails
  - CCDB (Cray Comparative Debugger)
    - Comparative debugging

# Profile Your Application (VTune / CrayPat)



# Cori Phase 2





# What is different about Cori?

- **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 72 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 Cori?



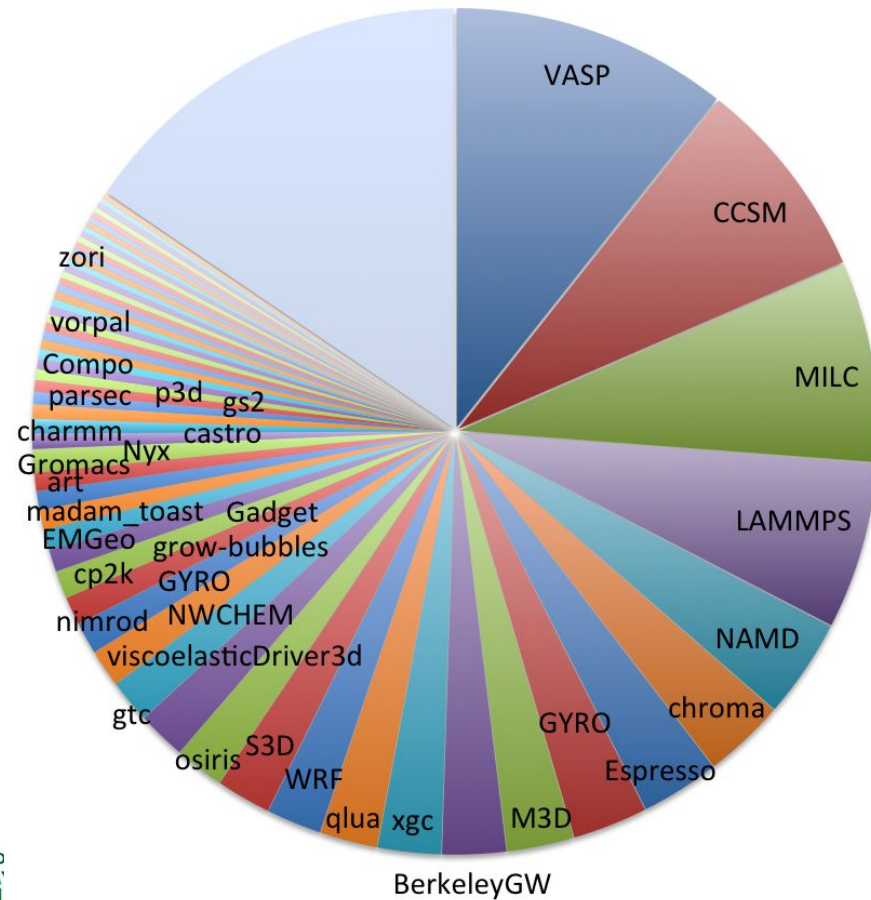
## Edison (Ivy-Bridge):

- 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

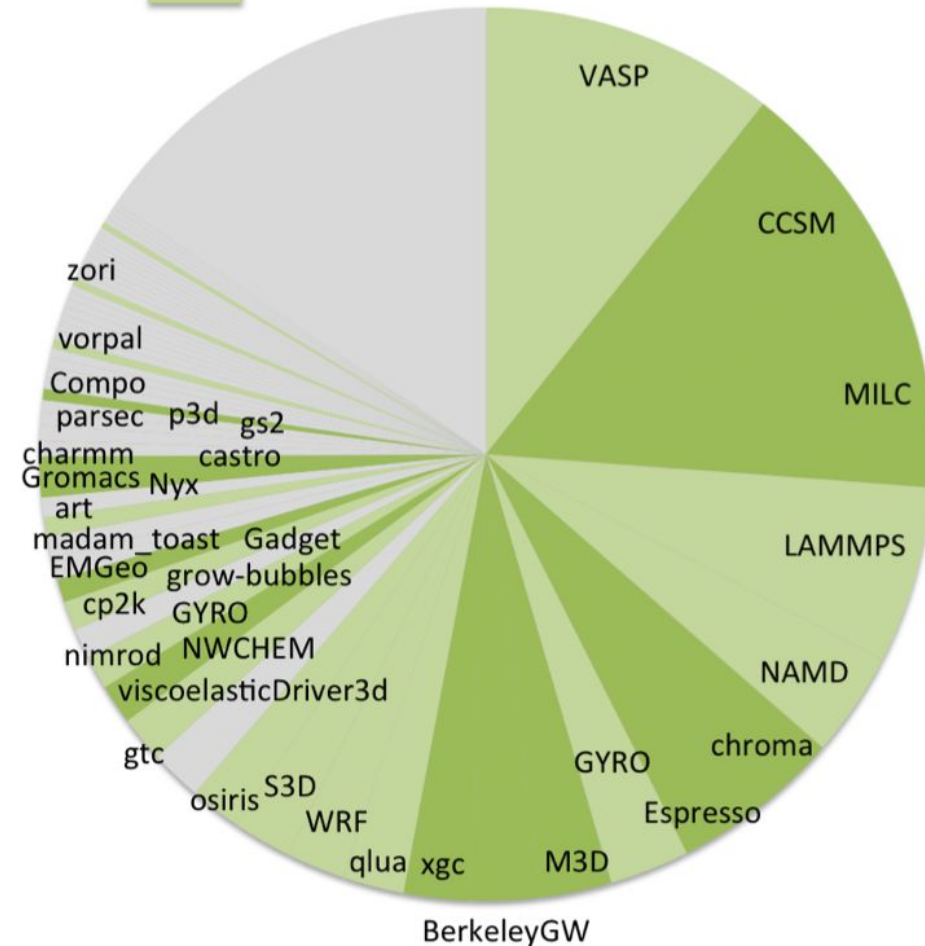
## Cori (Knights-Landing):

- Up to 72 Physical Cores Per CPU
- Up to 288 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

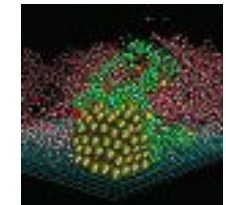
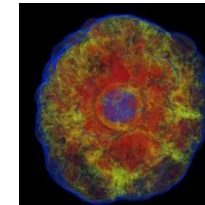
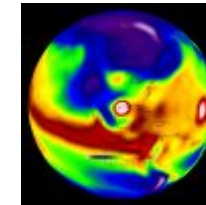
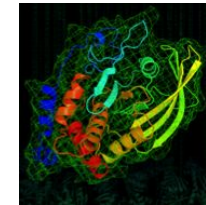
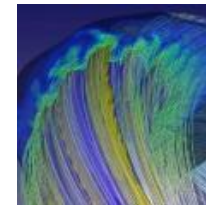
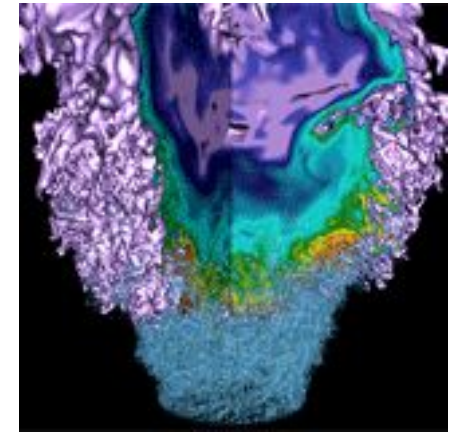
## Breakdown of Application Hours on Hopper and Edison 2013



NESAP Tier-1, 2 Code  
 NESAP Proxy Code or Tier-3 Code



# Basic Optimization Concepts



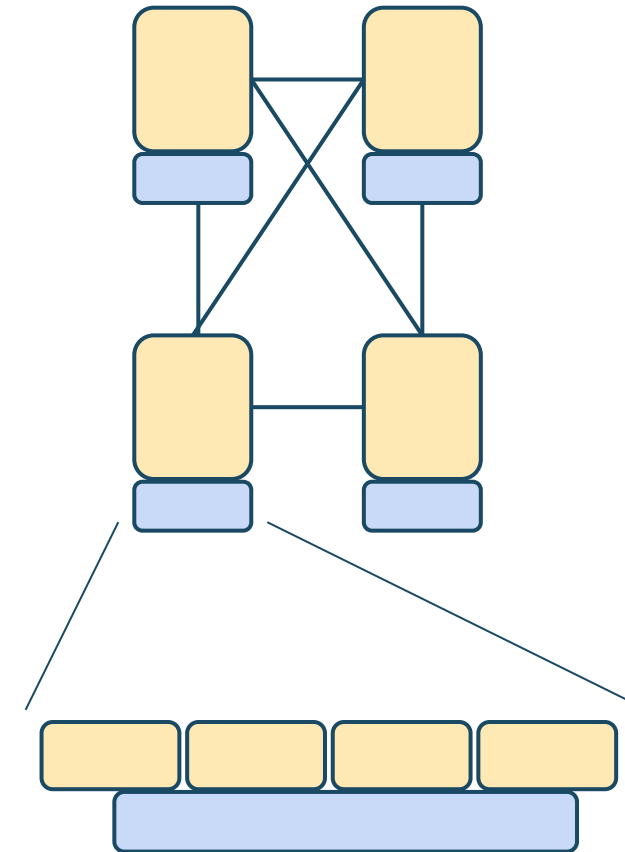
Need to explicitly consider both inter and on-node parallelism in application.

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.

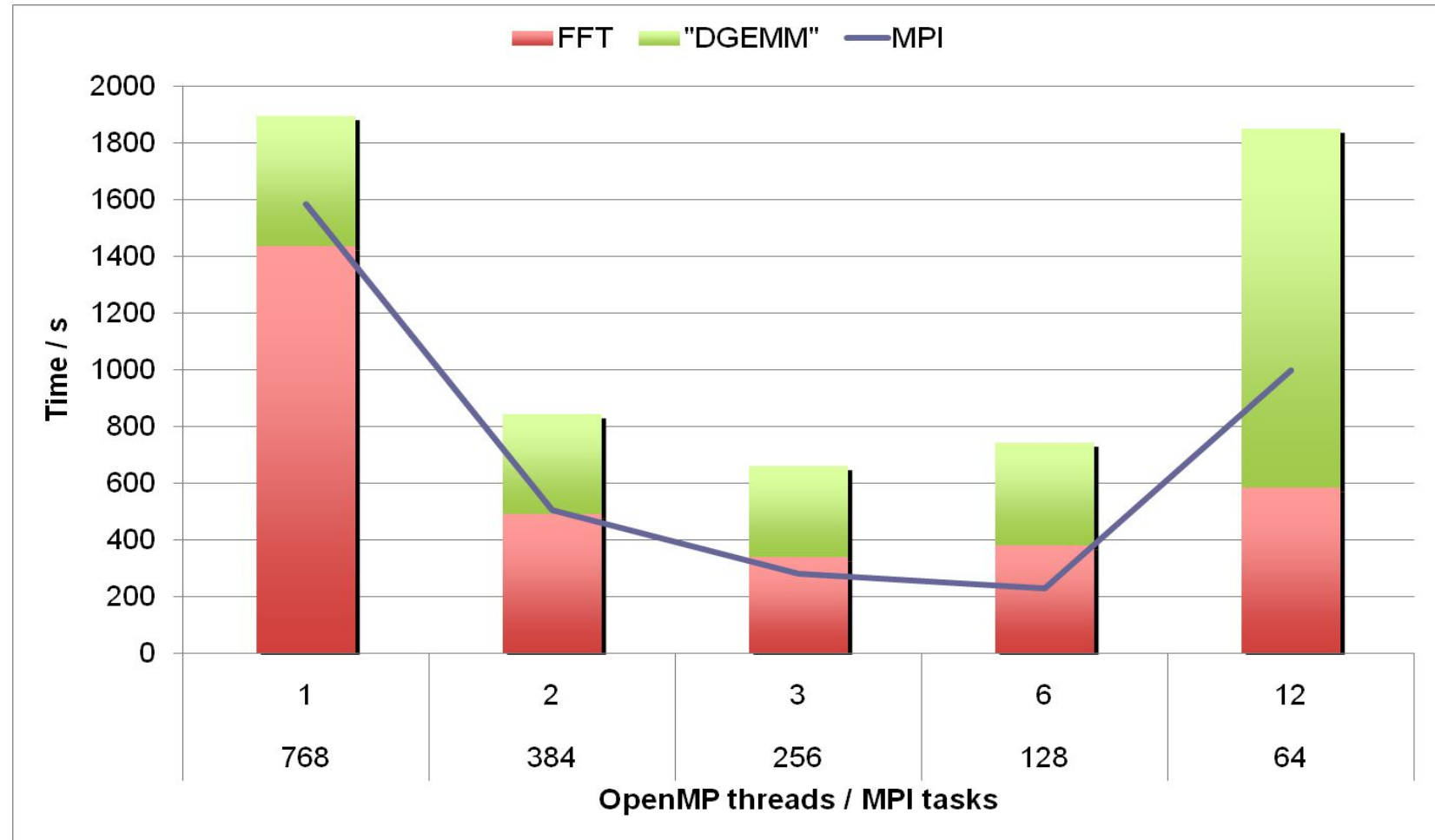



Figure Courtesy of Andrew Canning

There is a another important form of on-node parallelism

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


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

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

Vectoriz  
above l

of the



# 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

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_Nzml
  do index_jp = 1, mesh_Nrml
    index_2dp = index_jp+mesh_Nrml*(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_Nzml
  do index_jp = 1, mesh_Nrml
    index_2dp = index_jp+mesh_Nrml*(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_dfdz_v
    + Ms(index_ip,2,index_ip,index_2D)*
```

Example From Cray COE Work on XGC1

# Things that prevent vectorization in your code

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_Nzml  
  do index_jp = 1, mesh_Nrml  
    index_2dp = index_jp+mesh_Nrml*(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_Nzml  
  do index_jp = 1, mesh_Nrml  
    index_2dp = index_jp+mesh_Nrml*(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_dfdz_v  
    + Ms(index_ip,2,index_ip,index_2D)*
```

Example From Cray COE Work on XGC1

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

```
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

**$n*m + n$**

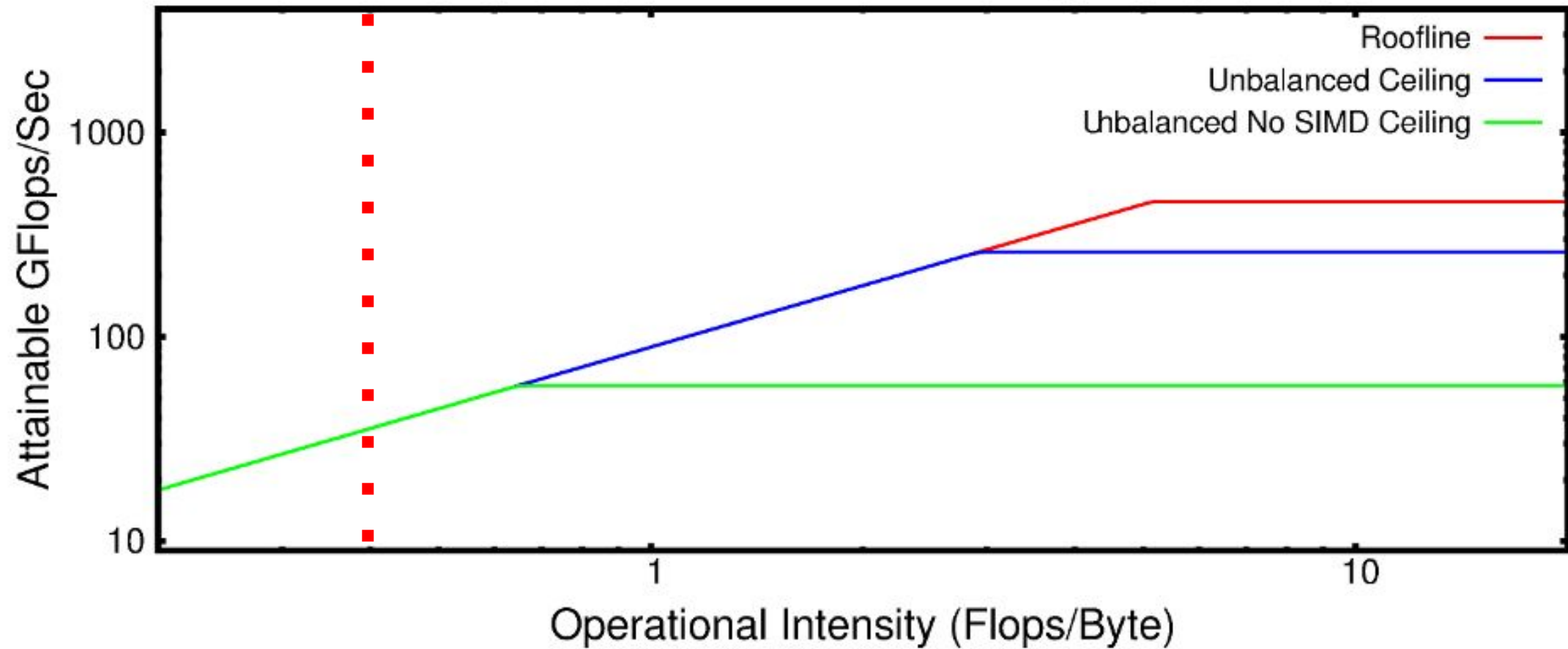
Requires 8 bytes loaded from DRAM per FMA (if supported). Assuming 100 GB/s bandwidth on Edison, we can **at most achieve 25 GFlops/second** (2 Flops per FMA)

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



# Roofline Model For Edison

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec



# 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
```



```
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:

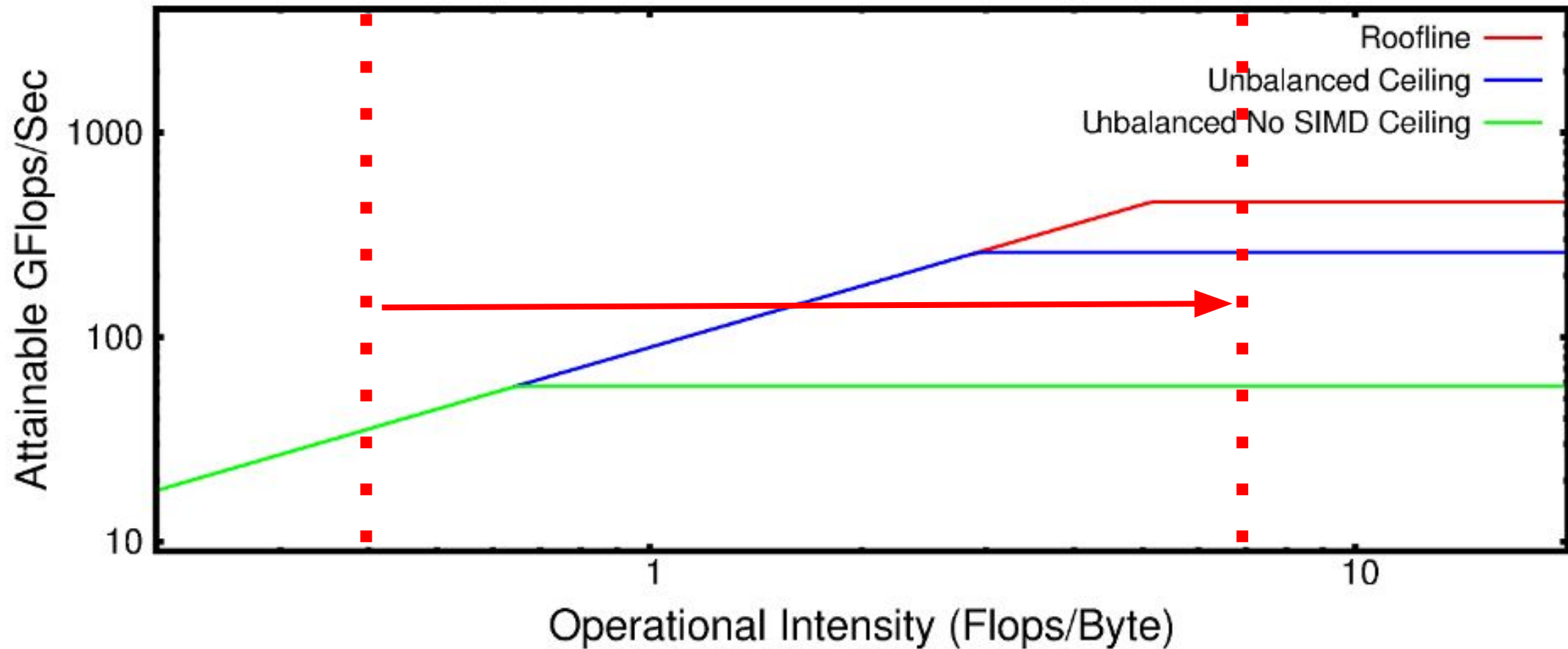
$$n*m + n$$

Loads From DRAM:

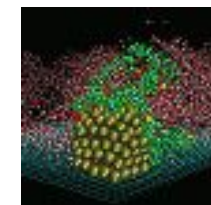
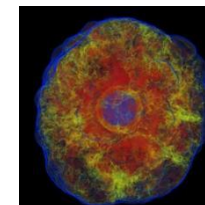
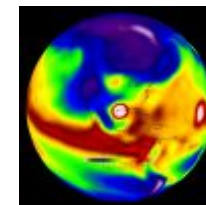
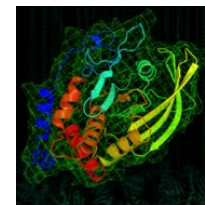
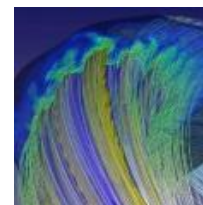
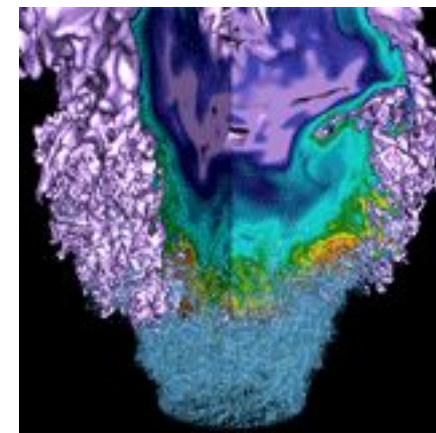
$$m/block * (n+block) \\ = n*m/block + m$$

# Improving Memory Locality Moves you to the Right on the Roofline

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec



# Optimization Strategy



# The Ant Farm!



OpenMP scales only to 4 Threads

large cache miss rate

Code shows no improvements when turning on vectorization

50% Walltime is IO

Communication dominates beyond 100 nodes

Compute intensive doesn't vectorize

Memory bandwidth bound kernel

IO bottlenecks

MPI/OpenMP Scaling Issue

Can you use a library?

Increase Memory Locality

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

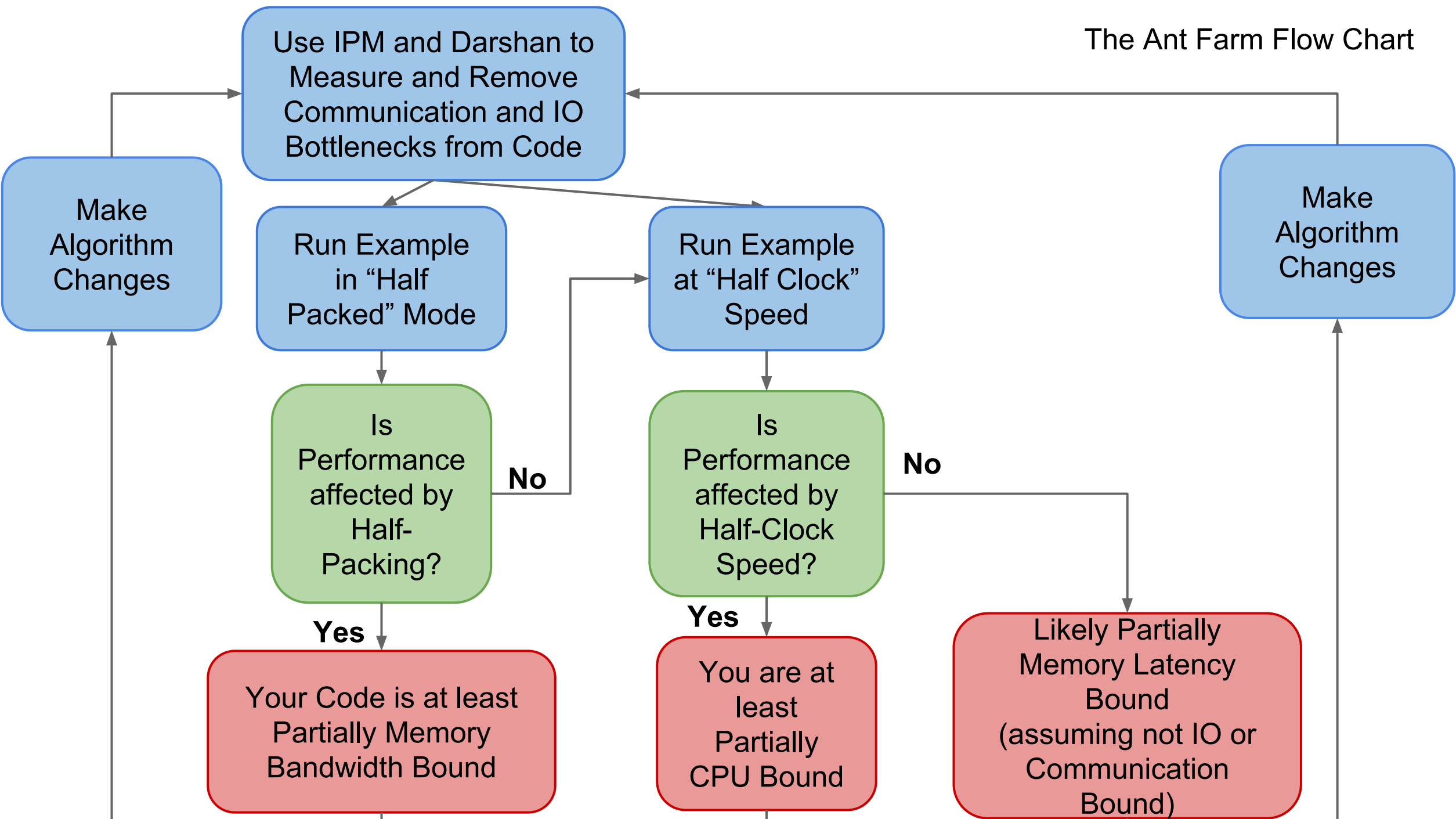
Use Edison to Test/Add OpenMP Improve Scalability. Help from NERSC/Cray COE Available.

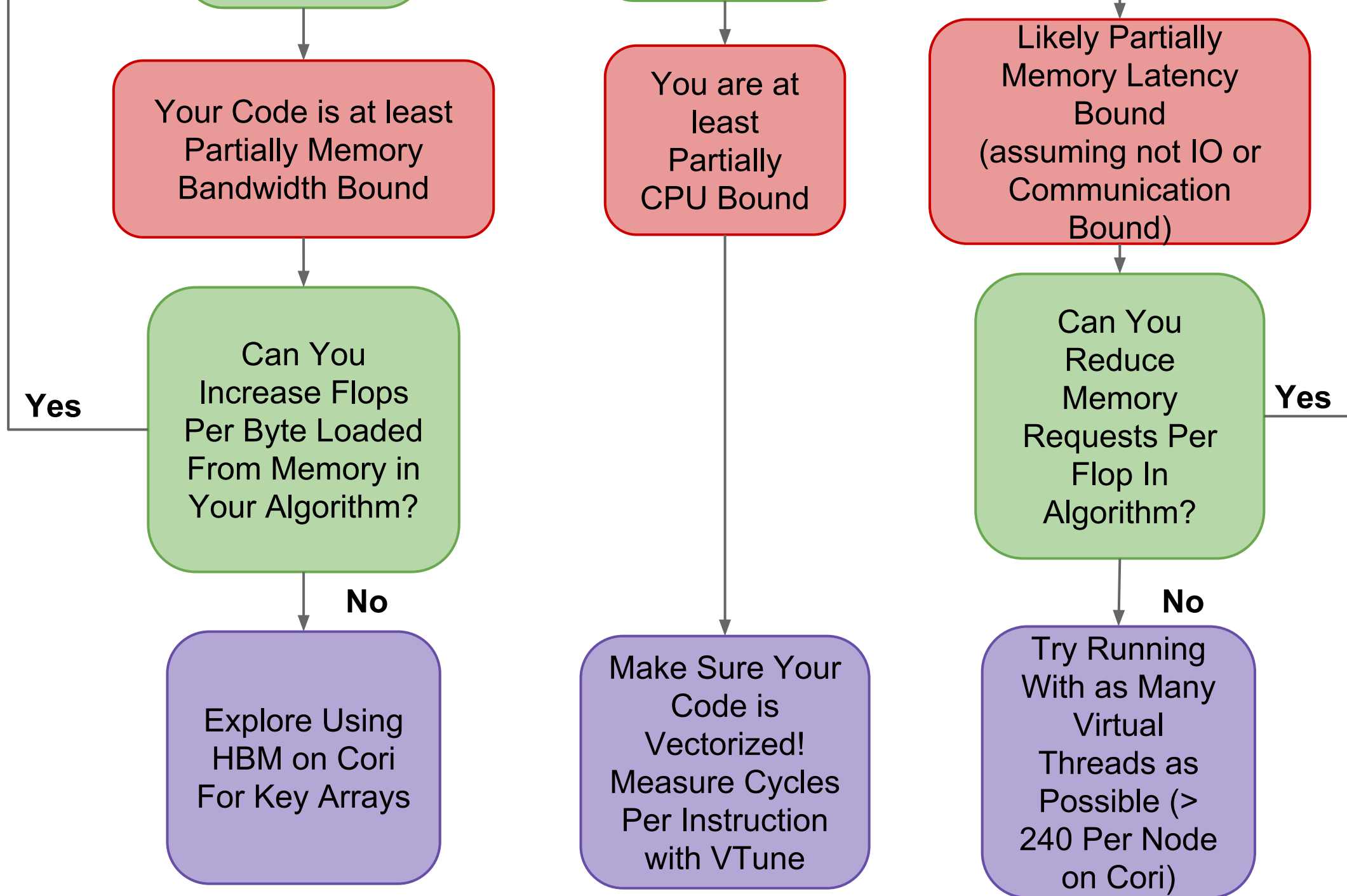
Create micro-kernels or examples to examine thread level performance, vectorization, cache use, locality.

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

Utilize performant / portable libraries

# The Ant Farm Flow Chart





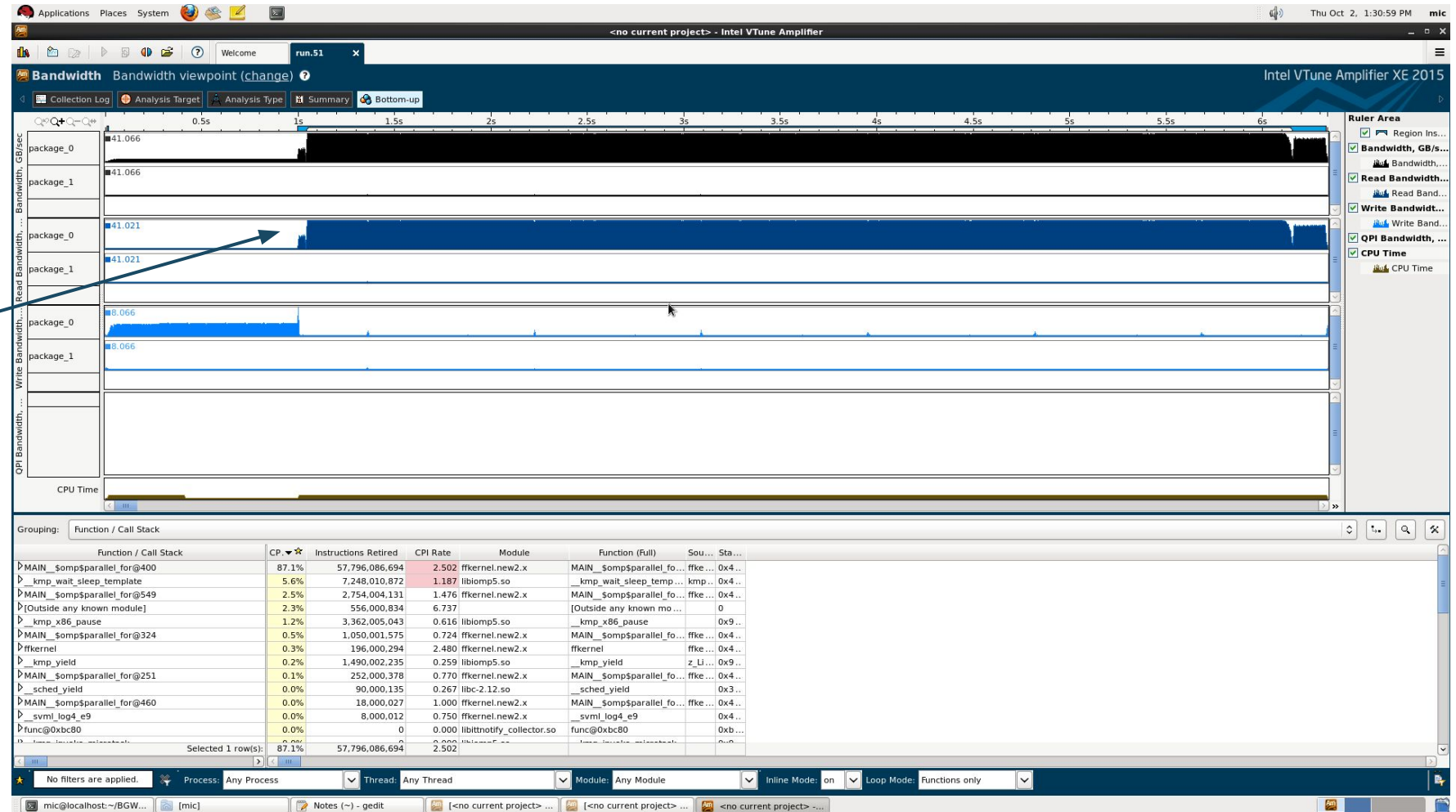
# Measuring Your Memory Bandwidth Usage (VTune)

Measure memory bandwidth usage in VTune. (Next Talk)

Compare to Stream GB/s.

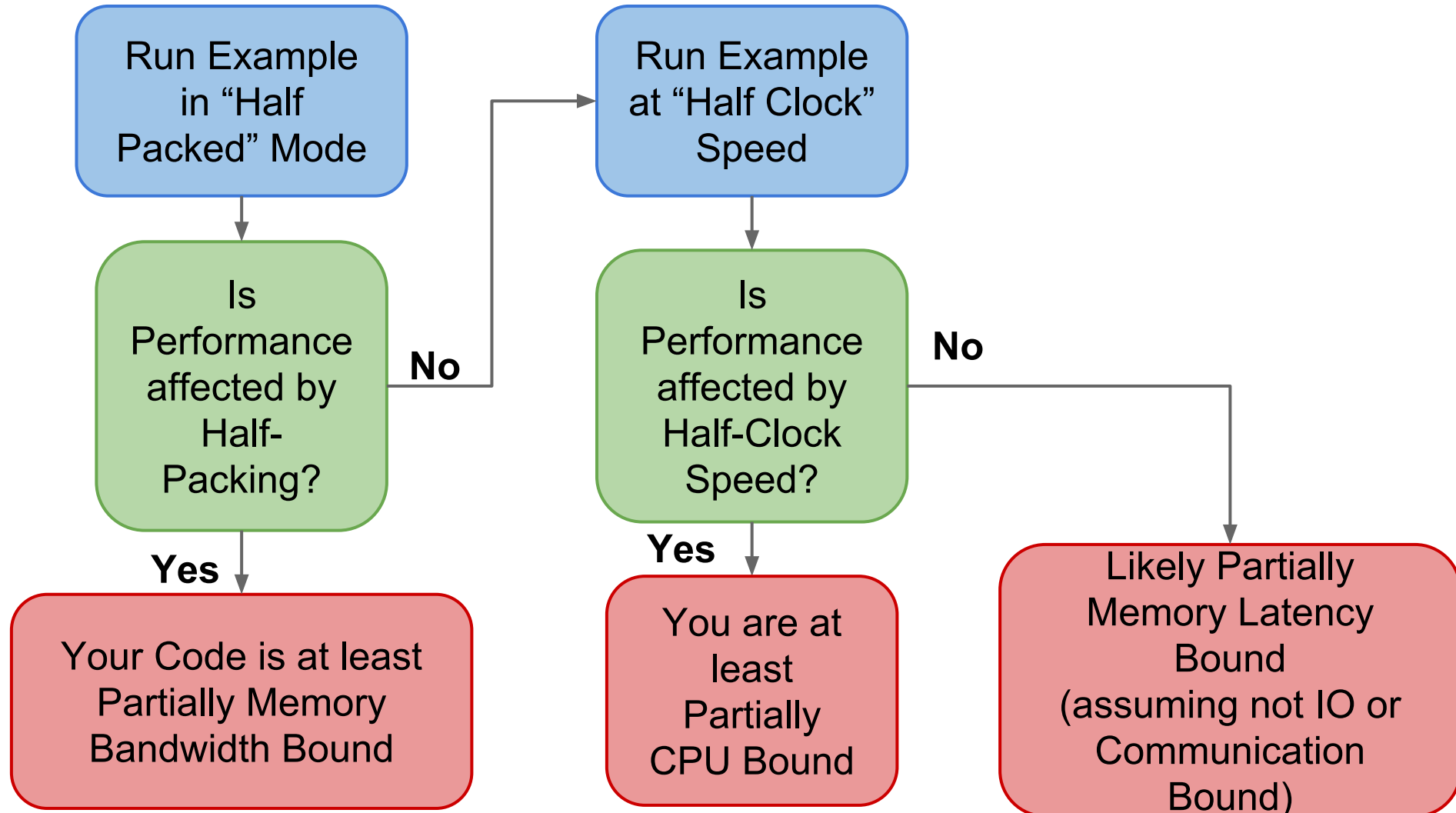
If 90% of stream, you are memory bandwidth bound.

If less, more tests need to be done.





# Are you memory or compute bound? Or both?



# 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?

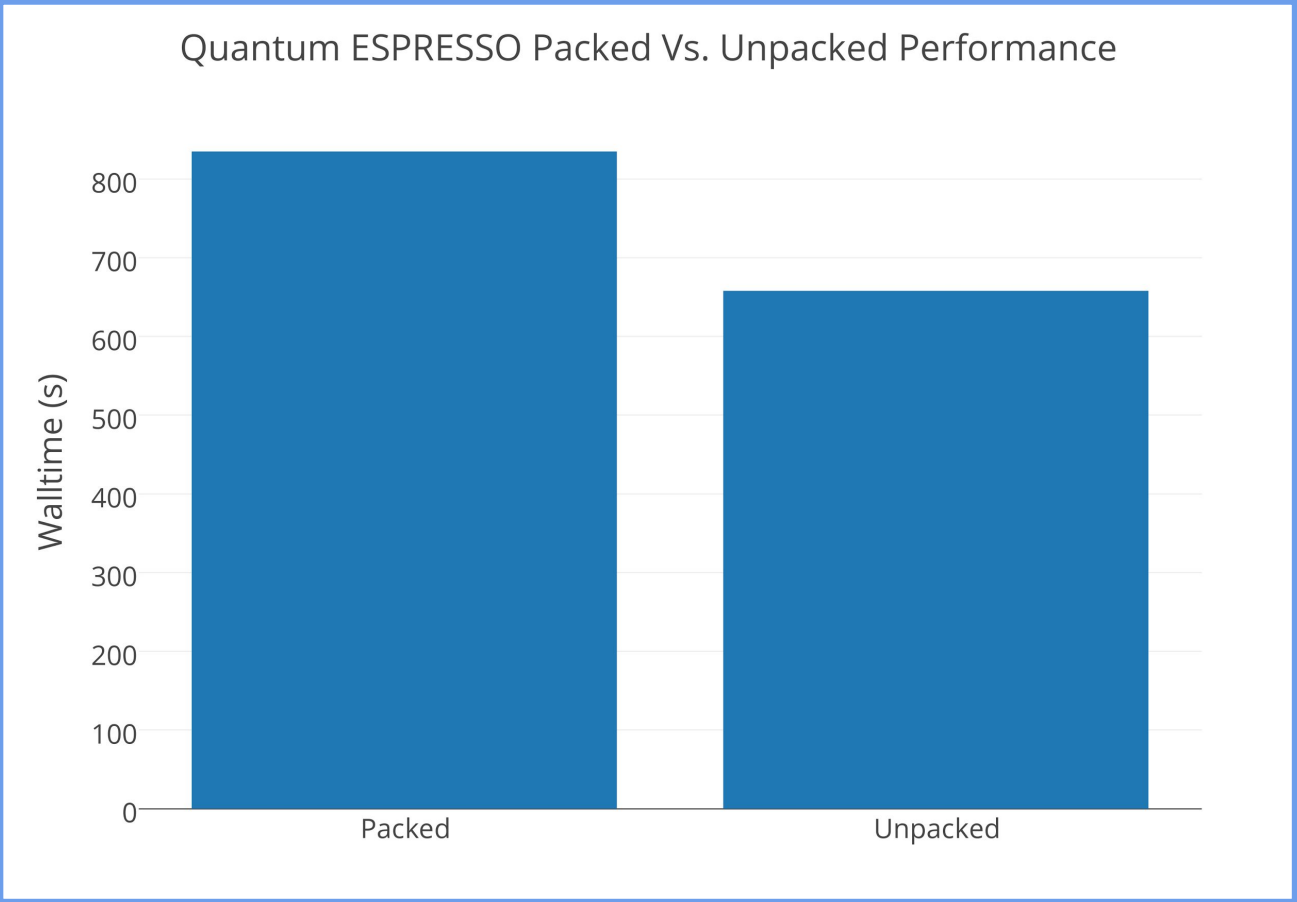
Run Example  
in “Half  
Packed” Mode

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 performance



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

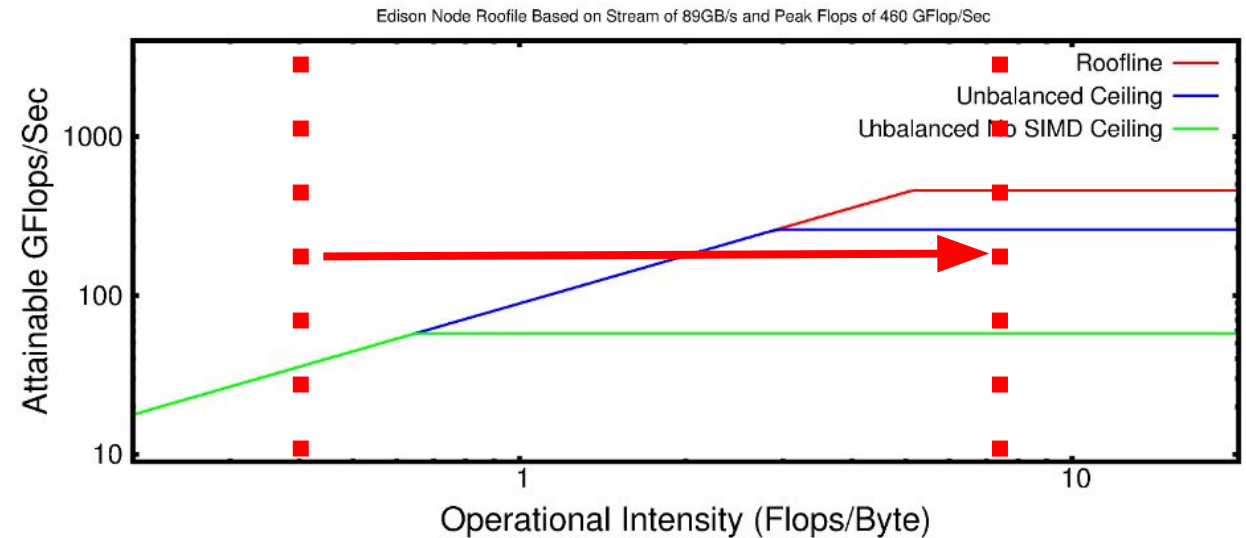
```
aprun --p-state=1900000 ...
```

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



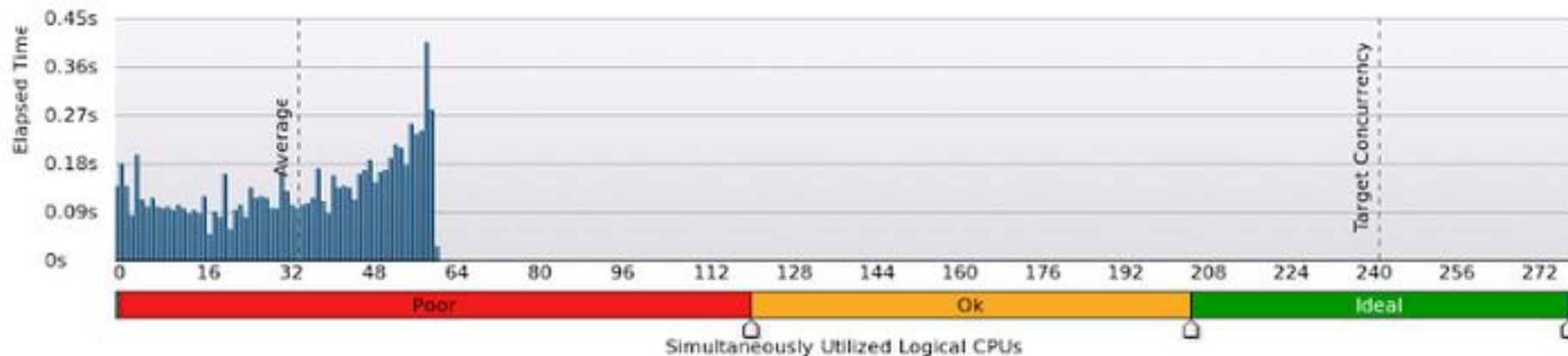
2. Identify the key arrays leading to high memory bandwidth usage and make sure they are/will-be allocated in HBM on Cori.

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`

# Complex-Division (without -fp model fast=2)



Intel VTune Amplifier XE 2015

Hotspots viewpoint (change) ?

Collection Log Analysis Target Analysis Type Summary Bottom-up Caller/Callee Top-down Tree Tasks and Frames gppkernel...

Source Assembly Assembly grouping: Address

S. Li.	Source	Address	Sou.. Line	Assembly	Effective Time by Utilization
					Idle Poor Ok Ideal Over
466	scht = scht + scha(ig)	0x408745	481	vunpckhpd %ymm3, %ymm3, %ymm3	0.001s
467	endif	0x408749	480	vmovapd %xmm5, %xmm15	
468		0x40874d	480	vmovsdq %xmm15, -0x28(%rbp)	0.202s
469	else	0x408752	480	fldq -0x28(%rbp), %st0	0.456s
470	! !dir\$ no unroll	0x408755	480	vunpckhpd %xmm5, %xmm5, %xmm11	0.001s
471	do ig = igbeg, min(igend,igmax)	0x408759	480	fld %st0, %st0	
472	! do ig = 1, igmax	0x40875b	480	vmovsdq %xmm11, -0x28(%rbp)	0.184s
473		0x408760	480	fmul %st1, %st0	0.444s
474	wdiff = wxt - wtilde_array(ig,my_igp)	0x408762	480	vextractf128 \$0x1, %ymm5, %xmm9	0.006s
475		0x408768	480	fldq -0x28(%rbp), %st0	
476	cden = wdiff	0x40876b	480	fld %st0, %st0	0.183s
477	!rden = cden * CONJG(cden)	0x40876d	480	fmul %st1, %st0	0.418s
478	!rden = 1D0 / rden	0x40876f	480	vmovsdq %xmm12, -0x28(%rbp)	0.006s
479	!delw = wtilde_array(ig,my_igp) * CONJG(cden) * rden	0x408774	480	faddp %st0, %st2	0.001s
480	cden = 1 /cden	0x408776	480	fxch %st1, %st0	0.196s
481	delw = wtilde_array(ig,my_igp) * cden	0x408778	480	fdivr %st3, %st0	0.462s
482	delwr = delw*CONJG(delw)	0x40877a	480	fldq -0x28(%rbp), %st0	0.113s
483	wdiffr = wdiff*CONJG(wdiff)	0x40877d	480	vmovsdq %xmm7, -0x28(%rbp)	0.192s
484		0x408782	480	fld %st0, %st0	0.418s
485	! JRD: Complex division is hard to vectorize. So, we help the compiler.	0x408784	480	fmul %st4, %st0	0.001s
486	scha(ig) = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,n1)	0x408786	480	fxch %st1, %st0	0.025s
487	! scha_temp = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,n1)	0x408788	480	fmul %st3, %st0	0.602s
488		0x40878a	480	fldq -0x28(%rbp), %st0	0.002s
489	! JRD: This if is OK for vectorization	0x40878d	480	fld %st0, %st0	0.026s
490	if (wdiffr.gt.limittwo .and. delwr.lt.limitone) then	0x40878f	480	fmulp %st0, %st5	0.185s
491	scht = scht + scha(ig)	0x408791	480	vunpckhpd %xmm9, %xmm9, %xmm4	0.404s
492	endif	0x408796	480	fxch %st4, %st0	0s

Selected 1 row(s): Highlighted 217 row(s):



# So, you are neither compute nor memory bandwidth 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

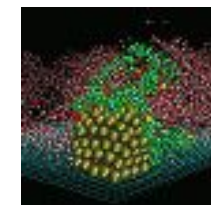
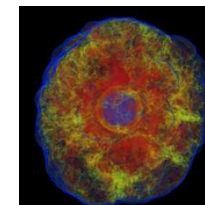
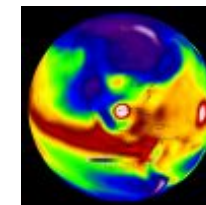
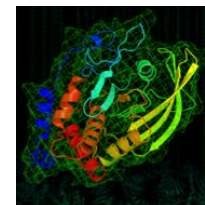
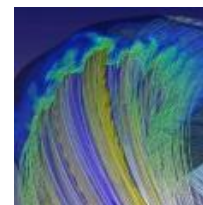
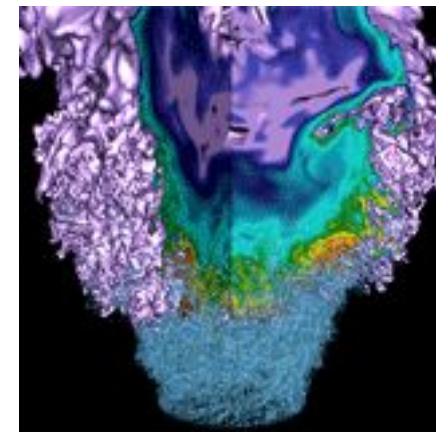
```
aprun -n 24 ....
```

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.

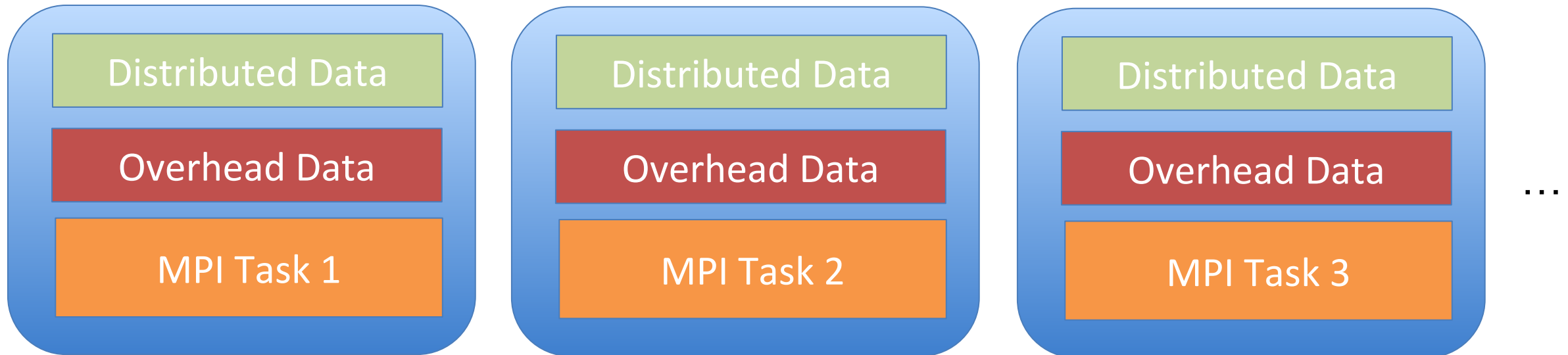
On Cori, each core will support up to 4 threads. Use them all.



# NESAP Case Study



- ★ Big systems require more memory. Cost scales as  $N_{\text{atoms}}^2$  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.**



# Computational Bottlenecks

---



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

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.

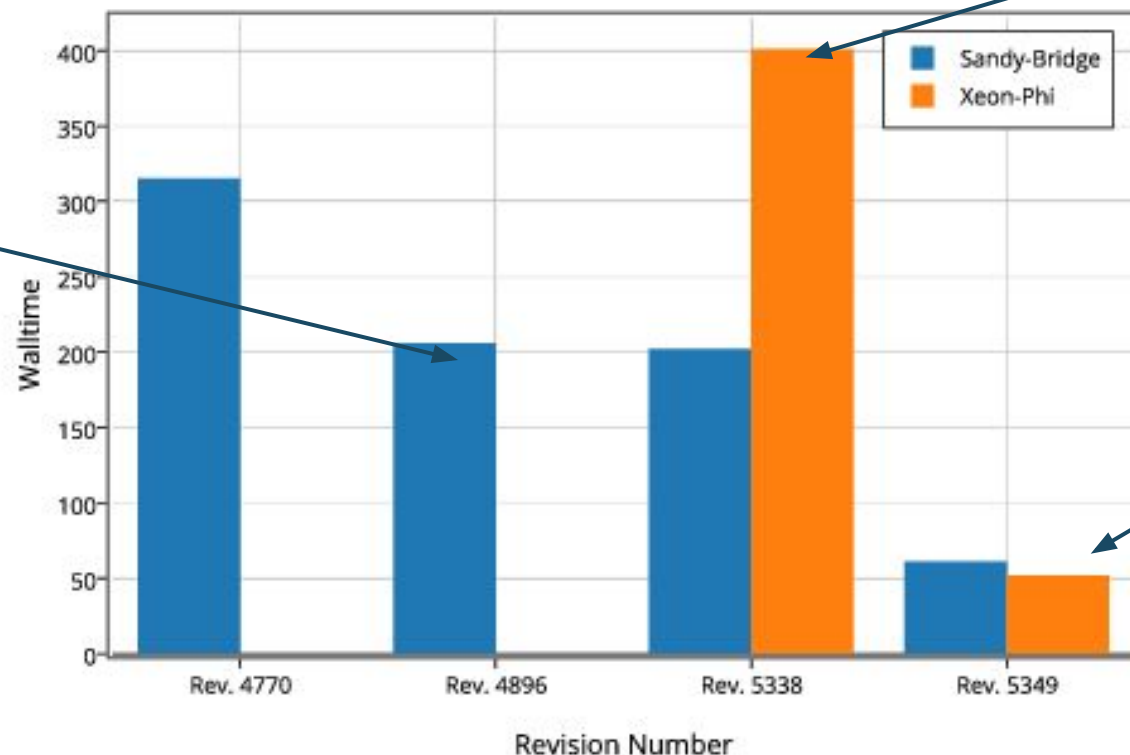
$$\langle n\mathbf{k} | \Sigma_{\text{CH}}(E) | n'\mathbf{k} \rangle = \frac{1}{2} \sum_{n''} \sum_{\mathbf{q}\mathbf{G}\mathbf{G}'} M_{n''n}^*(\mathbf{k}, -\mathbf{q}, -\mathbf{G}) M_{n''n'}(\mathbf{k}, -\mathbf{q}, -\mathbf{G}') \\ \times \frac{\Omega_{\mathbf{G}\mathbf{G}'}^2(\mathbf{q}) (1 - i \tan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

# Early Optimization Work

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

## Example: Optimization steps for Xeon Phi Coprocessor

Sigma Summation Optimization Process



Refactor to Have 3 Loop Structure:

Outer: MPI  
Middle: OpenMP  
Inner: Vectorization

Add OpenMP

Ensure Vectorization

# Final Loop Structure

```
!$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.

We've had two dungeon sessions with Intel. What kinds of questions can they help with?

1. Why is KNC slower than Haswell for this problem?

- Known to be bandwidth bound on KNC, but not on Haswell.
- How is it bound on Haswell?

2. How much gain can we get by allocating just a few arrays in HBM?

## Why KNC worse than Haswell for GPP Kernel?

---

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

```
do my_igp = 1, ngpown (OpenMP)
```

```
  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 l_eps_array(ig,my_igp) 819 MB, 512KB per row
```

```
      do work (including complex divide) depends on ig, iw ...
```



## Why KNC worse than Haswell for GPP Kernel?

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

```
do my_igp = 1, ngpown (OpenMP)
```

```
  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 l_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 KNC is 256 KB per Hardware Thread

L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core

## Why KNC worse than Haswell for GPP Kernel?

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

```
do my_igp = 1, ngpown (OpenMP)
  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 l_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 KNC is 256 KB per Hardware Thread  
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)

igblk = 2048

do my\_igp = 1, ngpown (OpenMP)

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 l\_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 KNC is 256 KB per Hardware Thread

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.**

lgblk=2048 - to enable reuse of L2 cache on KNC

- Morning:        2S Haswell 27.9s        KNC 39.9s
- Afternoon:      2S Haswell 27.5s        KNC 29.7s

The loss of L3 on MIC makes locality more important.

# How much performance can we get from 3 arrays in Fast Memory?

- **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

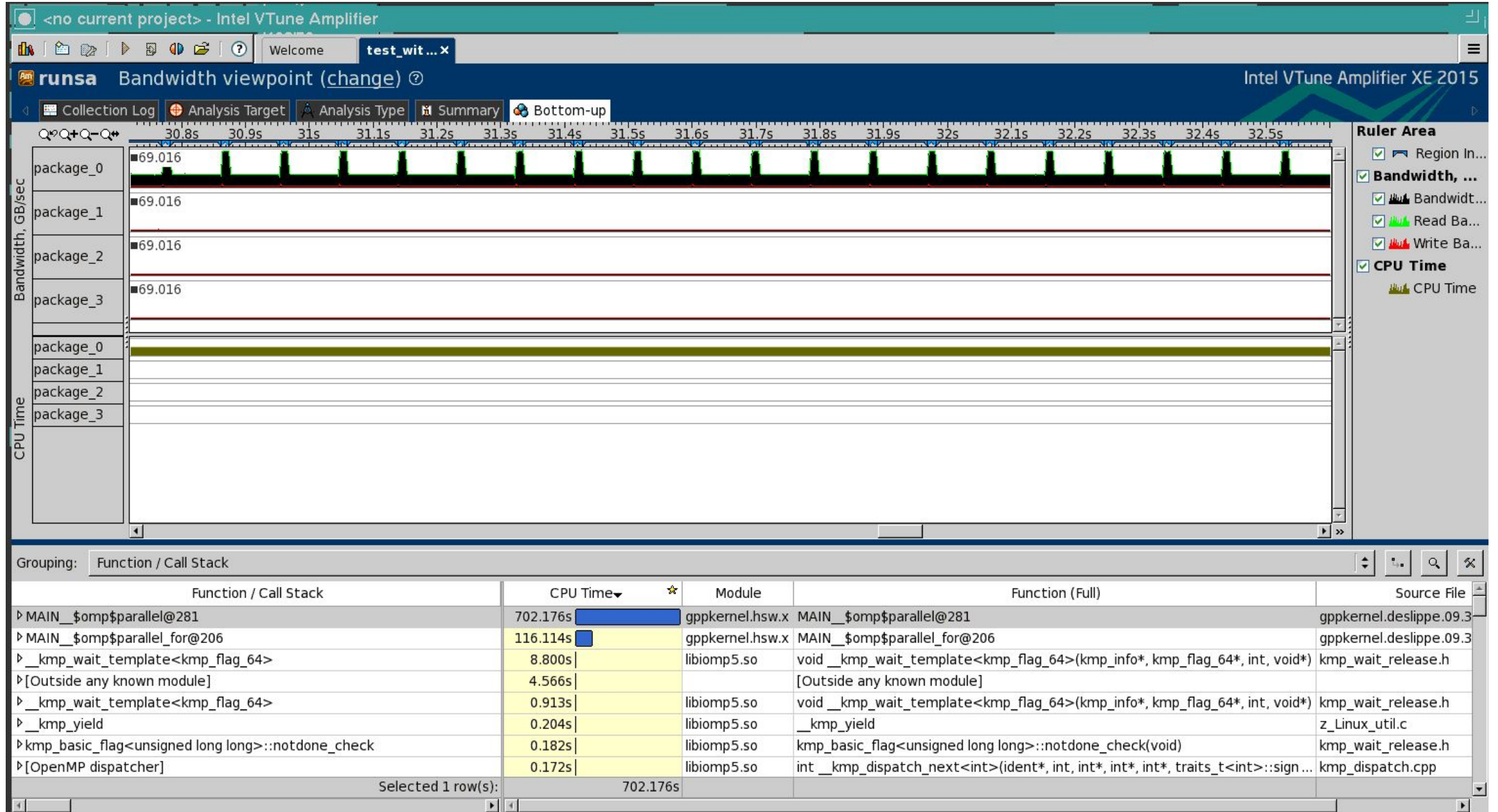
On Haswell:

Link with '-ljemalloc -lmemkind -lpthread -lnuma'

% **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

Bandwidth collection on Haswell. Now \*Mostly\* not bandwidth bound.



# General Exploration of two OpenMP regions



VNC config

<no current project> - Intel VTune Amplifier

Welcome test\_ge x

General Exploration General Exploration viewpoint (change) ? Intel VTune Amplifier XE 2015

Collection Log Analysis Target Analysis Type Summary Bottom-up Top-down Tree Tasks and Frames gppkernel.d...

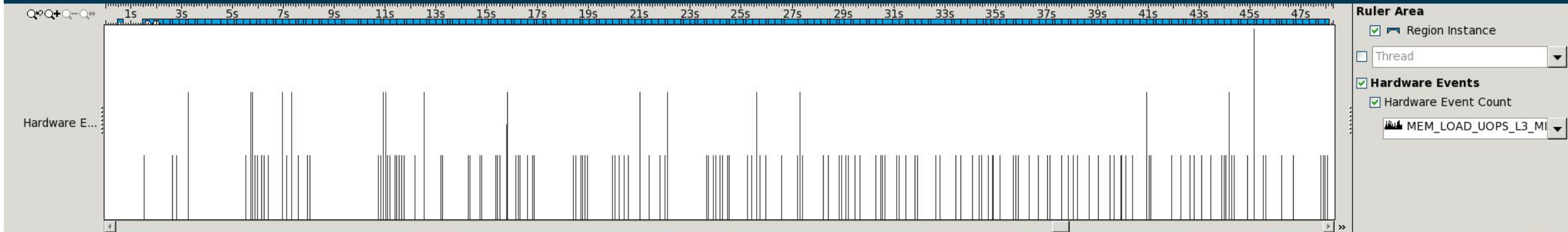
Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	MUX Reliability	Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)			Pot... Gain	Pot.. Gain (% of Col... Tim..)	Elaps... Time	Nu. of Ope. thr..	Ins.. Cou.	Mod.	Fun... (Full)	Sou... File	Start Address
					Retiring	Bad Specul...	Back-End Bound		Front-... Bound									
							Memory Bound	Core Bound										
▷ MAIN_\$omp\$parallel:18@unknown:281:529	1,757,836,636,751	2,159,187,238,776	0.814		0.328	0.001	0.141	0.731	0.003	0.926s	1.9%	39.849s	18	512			0	
▷ MAIN_\$omp\$parallel:18@unknown:67:74	3,946,005,919	4,100,006,150	0.962		0.245	0.004	0.105	0.357	0.004	0.233s	0.5%	0.234s	18	1			0	
▷ MAIN_\$omp\$parallel:18@unknown:206:260	297,620,446,430	194,362,291,543	1.531		0.162	0.001	0.621	0.305	0.010	0.160s	0.3%	6.751s	18	512			0	
▷ [Serial - outside any region]	13,782,020,673	11,316,016,974	1.218		0.258	0.000	0.392	0.302	0.021		0.0%	1.465s					0	

The dynamic loop is now core bound, not memory bound. Removing the divide shows it to be the culprit!

Selected 1 row(s):

1,757,836,636,751	2,159,187,238,776	0.814	0.996	0.328	0.001	0.141	0.731	0.003	0.0%	39.849s	512						
-------------------	-------------------	-------	-------	-------	-------	-------	-------	-------	------	---------	-----	--	--	--	--	--	--

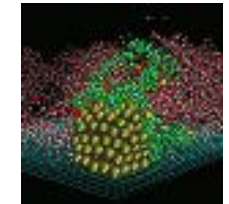
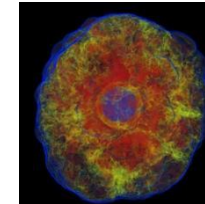
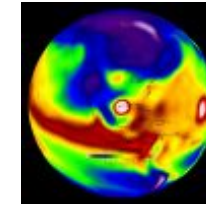
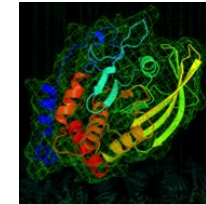
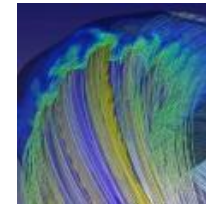
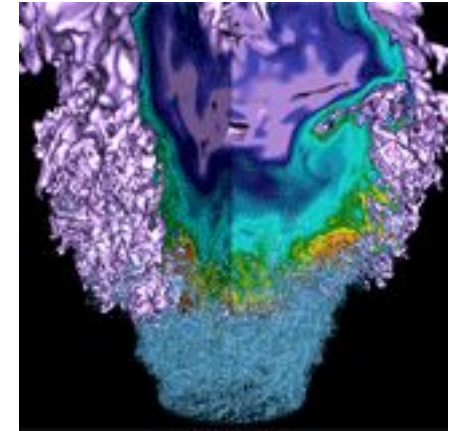


Ruler Area

- Region Instance
- Thread
- Hardware Events
  - Hardware Event Count
    - MEM\_LOAD\_UOPS\_L3\_MI

No filters are applied. Process: Any Process Thread: Any Thread Module: Any Module Call Stack Mode: User functions + 1 Inline Mode: on Loop Mode: Functions only

# Conclusions

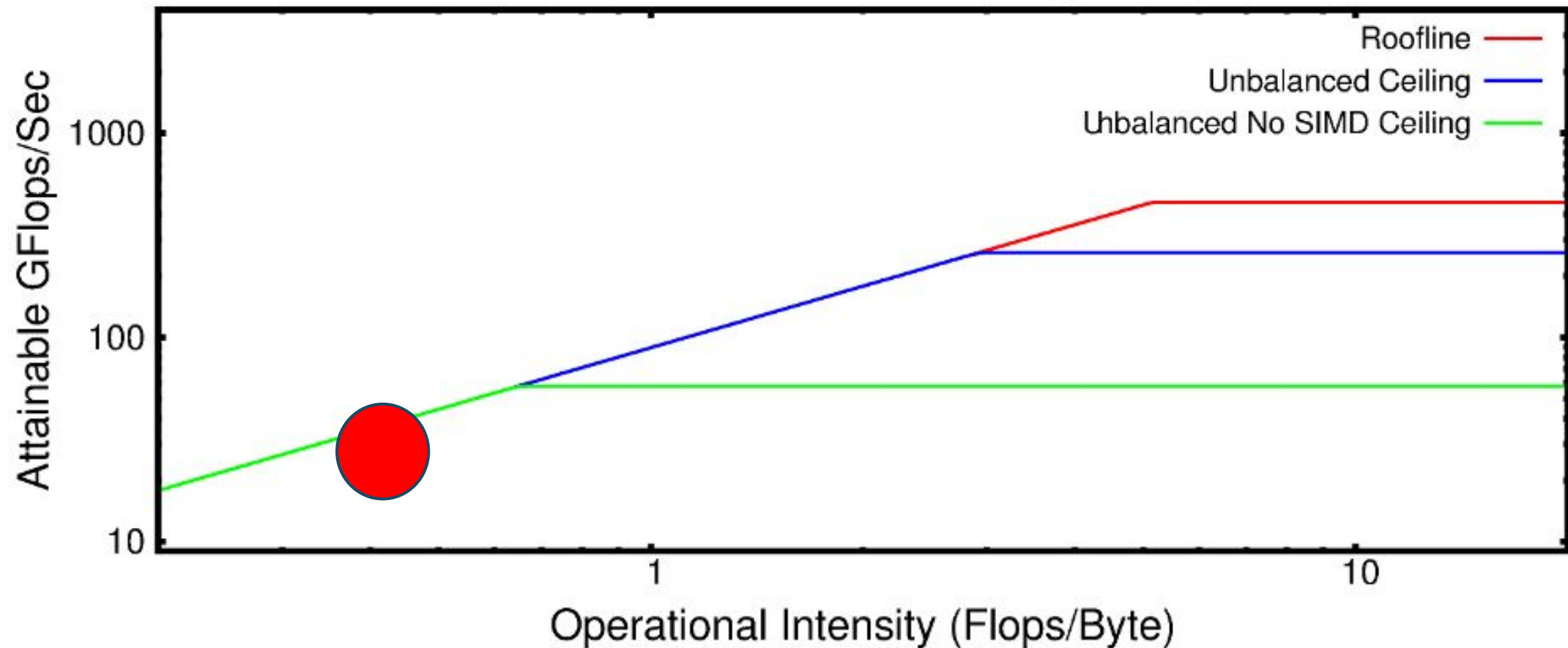




1. Optimizing code for Cori is not always straightforward. It is a continual discovery process that involves many sequential and coupled changes.

## 2013 - Poor locality, loop ordering issues

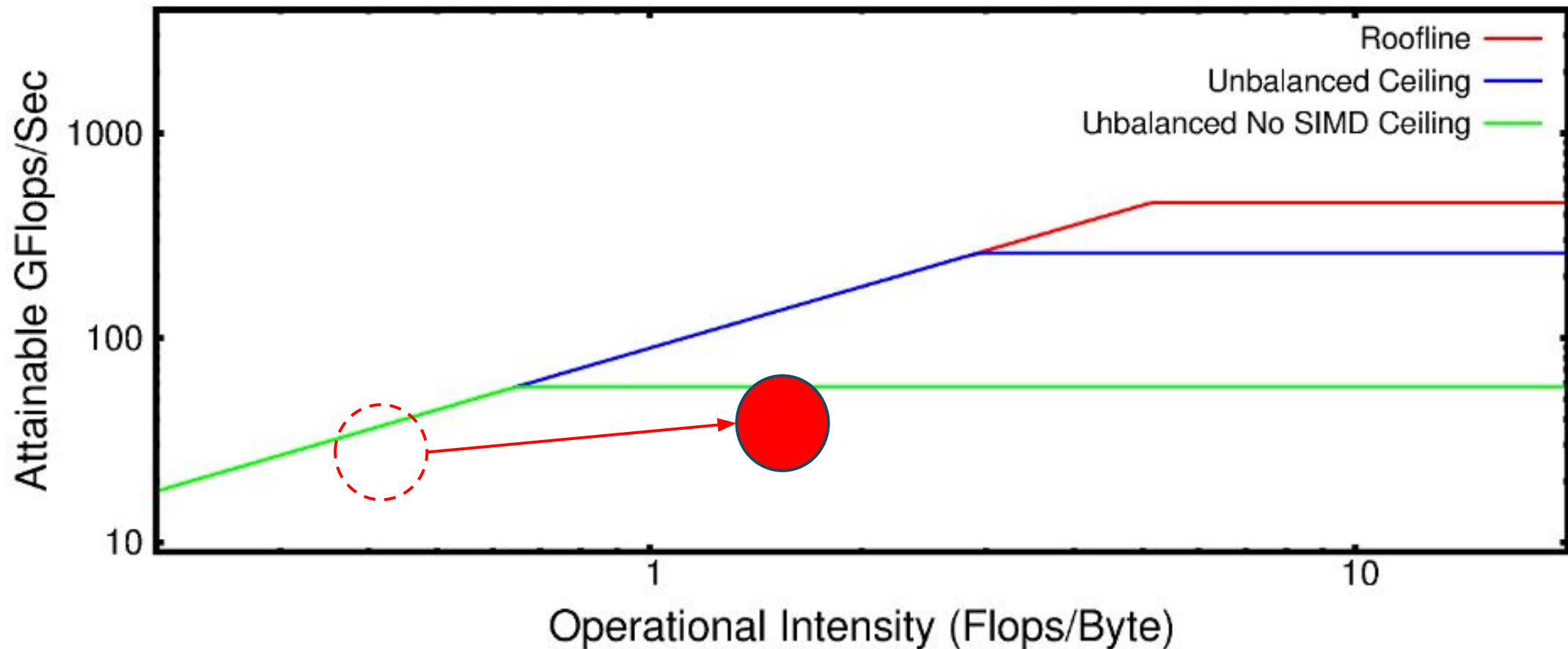
Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec



Disclaimer - this is a rough schematic

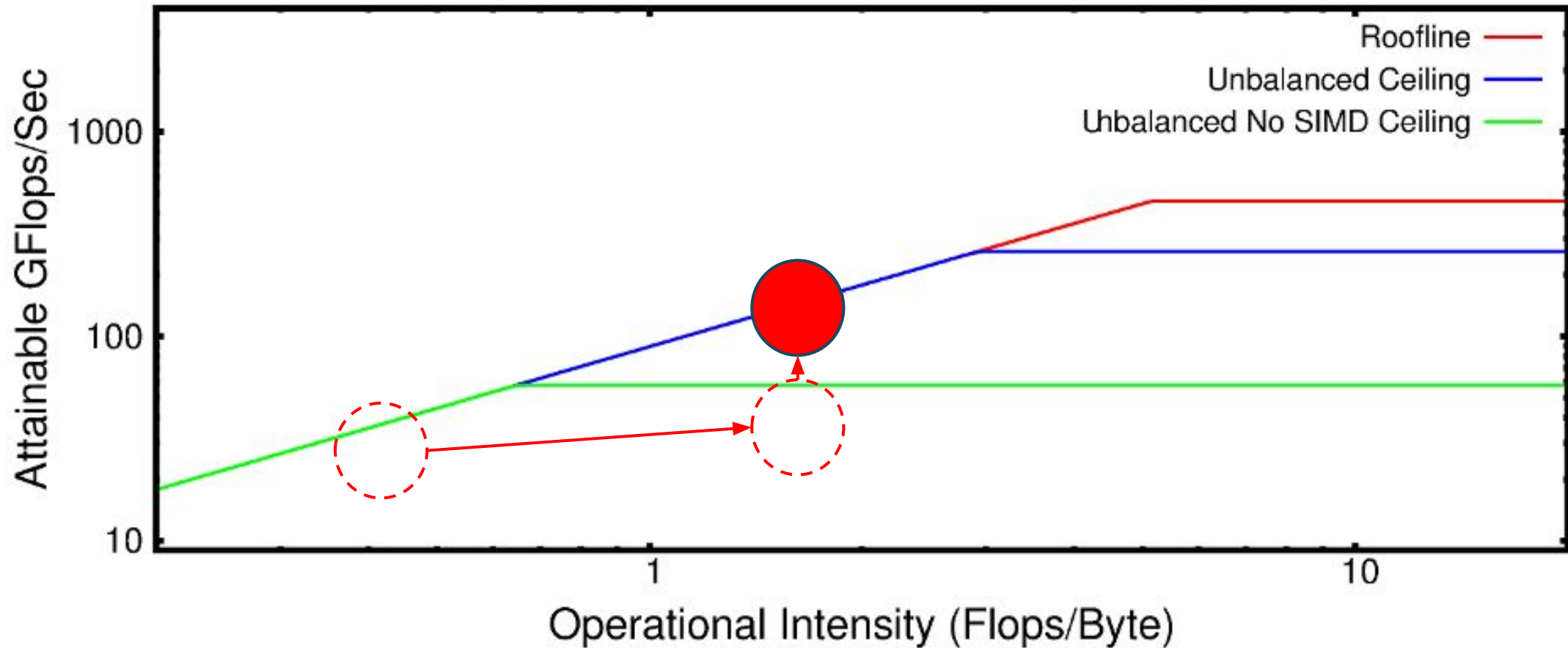
## 2014 - Refactored loops, improved locality

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec



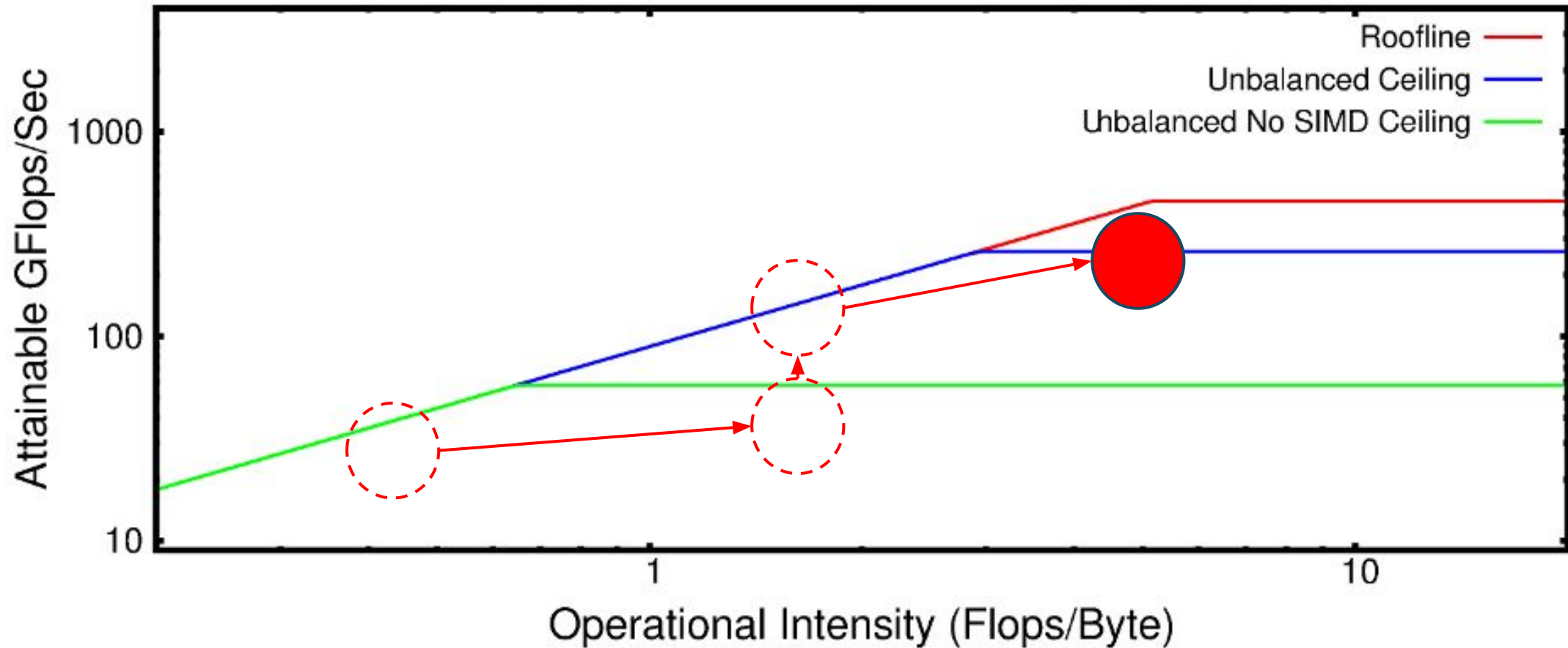
## 2014 - Vectorized Code

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec



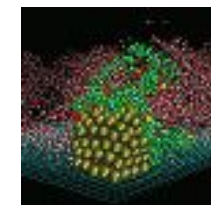
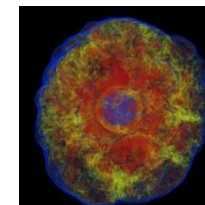
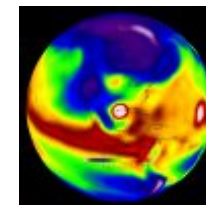
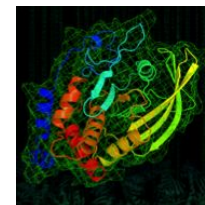
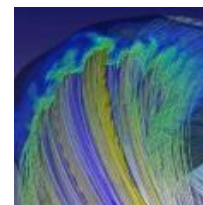
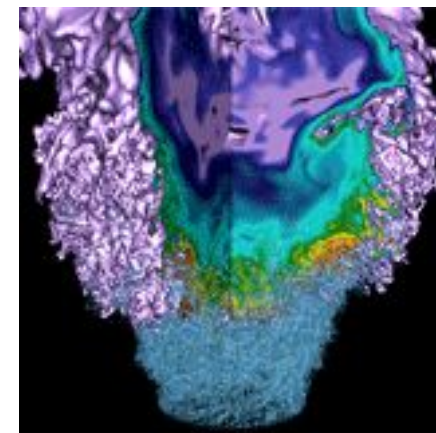
## 2015 - Cache Blocking

Edison Node Roofline Based on Stream of 89GB/s and Peak Flops of 460 GFlop/Sec



1. Optimizing code for Cori is not always straightforward. It is a continual discovery process that involves many sequential and coupled changes.
2. Use profiling tools like VTune and CrayPat on Edison 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)



# 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)  $r = c * \text{conjg}(c)$   
 $r = 1 / r$   
 $c = \text{conjg}(c) * r$

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



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



Intel VTune Amplifier XE 2015

Advanced Hotspots Hotspots viewpoint (change) ?

Collection Log Analysis Target Analysis Type Summary Bottom-up Caller/Callee Top-down Tree Tasks and Frames gppkernel... x

Source Assembly Assembly grouping: Address

S. Li.	Source	Address	Sou.. Line	Assembly	Effective Time by Utili
					Idle Poor Ok Ideal
469	else	0x4087b0	477	vmovddup %ymm3, %ymm2	0.455s
470	! !dir\$ no unroll	0x4087b4	477	vmovddup %ymm1, %ymm3	0.214s
471	do ig = igbeg, min(igend,igmax)	0x4087b8	477	vfmaddsub231pd %ymm2, %ymm7, %ymm14	0.349s
472	! do ig = 1, igmax	0x4087bd	477	vfmaddsub231pd %ymm3, %ymm5, %ymm8	0.045s
473		0x4087c2	477	vperm2f128 \$0x20, %ymm8, %ymm14, %ymm7	0.494s
474	wdiff = wxt - wtilde_array(ig,my_igp)	0x4087c8	477	vperm2f128 \$0x31, %ymm8, %ymm14, %ymm14	0.565s
475		0x4087ce	477	vunpcklpd %ymm14, %ymm7, %ymm1	0.423s
476	cden = wdiff	0x4087d3	478	vdivpd %ymm1, %ymm13, %ymm7	0.141s
477	rden = cden * CONJG(cden)	0x4087d7	480	vunpckhpd %xmm7, %xmm7, %xmm0	14.800s
478	rden = 1D0 / rden	0x4087db	480	vinsertrf128 \$0x1, %xmm0, %ymm7, %ymm8	0.727s
479	!rden = 1D0 + rden	0x4087e1	480	vmovddup %ymm8, %ymm14	1.869s
480	delw = wtilde_array(ig,my_igp) * CONJG(cden) * rden	0x4087e6	480	vextractf128 \$0x1, %ymm7, %xmm7	1.249s
481	delwr = delw*CONJG(delw)	0x4087ec	480	vmulpy (%r14,%rdi,1), %ymm14, %ymm8	0.005s
482	wdiffrr = wdiff*CONJG(wdiff)	0x4087f2	480	vunpckhpd %xmm7, %xmm7, %xmm0	3.126s
483		0x4087f6	480	vinsertrf128 \$0x1, %xmm0, %ymm7, %ymm7	0.015s
484	! JRD: Complex division is hard to vectorize. So, we help the compiler.	0x4087fc	480	vmovddup %ymm7, %ymm14	
485	scha(ig) = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,n1)	0x408800	480	vshufpd \$0x5, %ymm8, %ymm8, %ymm7	0.032s
486	! scha_temp = mygpvar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,n1)	0x408806	480	vmulpd %ymm7, %ymm6, %ymm6	0.619s
487		0x40880a	485	vmovupdy (%r14,%r15,1), %ymm7	3.080s
488	! JRD: This if is OK for vectorization	0x408810	480	vmulpy 0x20(%r14,%rdi,1), %ymm14, %ymm0	0.019s
489	if (wdiffrr.gt.limittwo .and. delwr.lt.limitone) then	0x408817	480	vfmaddsub213pd %ymm6, %ymm8, %ymm2	0.399s
490	scht = scht + scha(ig)	0x40881c	485	vunpckhpd %ymm7, %ymm7, %ymm14	3.034s
491	endif	0x408820	485	vmovupdy -0x50(%rbp), %ymm7	0.017s
492		0x408825	480	vshufpd \$0x5, %ymm0, %ymm0, %ymm8	0.025s
493	! scha_mult = merge(1.0,0.0,wdiffrr.gt.limittwo .and. delwr.lt.limitone)	0x40882a	480	vmulpd %ymm8, %ymm15, %ymm15	0.014s
494	! scht = scht + scha(ig)*scha_mult	0x40882f	485	vmovupdy 0x20(%r14,%r15,1), %ymm8	0.637s
495		0x408836	485	vmulpd %ymm7, %ymm14, %ymm6	0.333s

Selected 1 row(s):

Highlighted 3 row(s):



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

Intel VTune Amplifier XE 2015

Advanced Hotspots Hotspots viewpoint (change) ?

Collection Log Analysis Target Analysis Type Summary Bottom-up Caller/Callee Top-down Tree Tasks and Frames gppkernel...

Source Assembly Address

S. Li.	Source	Address	Sou.. Line	Assembly	Effect
472	! do ig = 1, igmax	0x4085d9	471	add \$0x4, %rax	0.013s
473		0x4085dd	474	vmovupdy (%r14,%rdi,1), %ymm10	0.094s
474	wdiff = wxt - wtilde_array(ig,my_igp)	0x4085e3	480	vmovupsy -0x70(%rbp), %ymm2	0.390s
475		0x4085e8	474	vmovupdy 0x20(%r14,%rdi,1), %ymm1	0.429s
476	cden = wdiff	0x4085ef	474	vsubpd %ymm10, %ymm0, %ymm5	3.454s
477	!rden = cden * CONJG(cden)	0x4085f4	474	vsubpd %ymm1, %ymm0, %ymm3	0.083s
478	!rden = 1D0 / rden	0x4085f8	480	vmulpd %ymm5, %ymm5, %ymm7	0.505s
479	!delw = wtilde_array(ig,my_igp) * CONJG(cden) * rden	0x4085fc	480	vunpckhpd %ymm5, %ymm5, %ymm9	0.342s
480	cden = 1 / cden	0x408600	480	vmulpd %ymm2, %ymm9, %ymm15	0.019s
481	delw = wtilde_array(ig,my_igp) * cden	0x408604	480	vmovddup %ymm5, %ymm8	0.080s
482	delwr = delw*CONJG(delw)	0x408608	480	vfmaddsub213pd %ymm15, %ymm13, %ymm8	0.213s
483	wdiffr = wdiff*CONJG(wdiff)	0x40860d	480	vshufpd \$0x5, %ymm7, %ymm7, %ymm6	0.380s
484		0x408612	480	vaddpd %ymm6, %ymm7, %ymm7	0.001s
485	! JRD: Complex division is hard to vectorize. So, we help the compiler.	0x408616	480	vshufpd \$0x5, %ymm8, %ymm8, %ymm9	0.075s
486	scha(ig) = mygpar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,n1)	0x40861c	480	vdivpd %ymm7, %ymm9, %ymm6	0.232s
487	! scha_temp = mygpar1 * aqsntemp(ig,n1) * delw * I_eps_array(ig,n1)	0x408620	480	vmulpd %ymm3, %ymm3, %ymm8	2.619s
488		0x408624	480	vunpckhpd %ymm3, %ymm3, %ymm9	0.267s
489	! JRD: This if is OK for vectorization	0x408628	480	vmulpd %ymm2, %ymm9, %ymm2	0.114s
490	if (wdiffr.gt.limittwo .and. delwr.lt.limitone) then	0x40862c	480	vmovddup %ymm3, %ymm15	0.062s
491	scht = scht + scha(ig)	0x408630	480	vfmaddsub213pd %ymm2, %ymm13, %ymm15	0.425s
492	endif	0x408635	480	vshufpd \$0x5, %ymm8, %ymm8, %ymm9	0.525s
493		0x40863b	480	vaddpd %ymm9, %ymm8, %ymm8	0.107s
494	! scha_mult = merge(1.0,0.0,wdiffr.gt.limittwo .and. delwr.lt.limitone)	0x408640	480	vshufpd \$0x5, %ymm15, %ymm15, %ymm7	0.031s
495	! scht = scht + scha(ig)*scha_mult	0x408646	480	vdivpd %ymm8, %ymm7, %ymm9	0.481s
496		0x40864b	481	vunpckhpd %ymm10, %ymm10, %ymm10	15.927s
497	enddo ! loop over g	0x408650	481	vshufpd \$0x5, %ymm6, %ymm6, %ymm2	0.107s
498		0x408655	481	vmulpd %ymm2, %ymm10, %ymm10	0.003s

Selected 1 row(s): Highlighted 40 row(s):



Approximation:

a. Real Division

b. Complex Division

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

?

Wall Time:

6.37 seconds

4.99 seconds

5.30 seconds

Approximation:

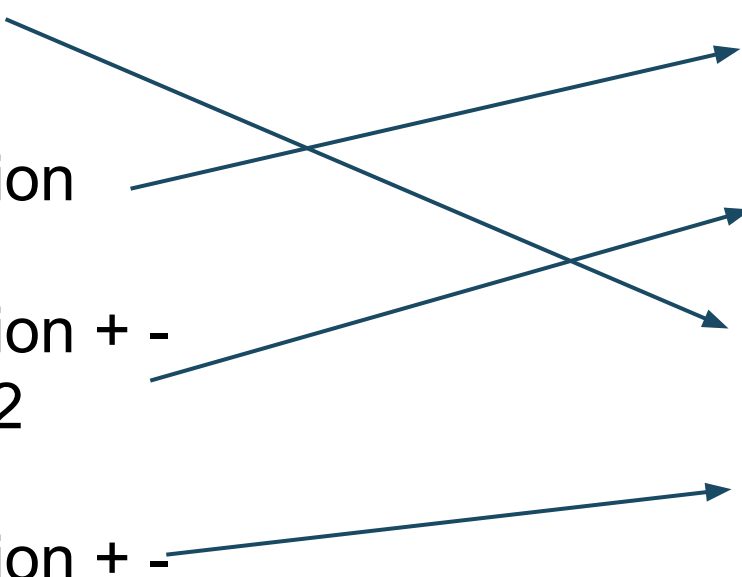
Wall Time:

a. Real Division	→	6.37 seconds
b. Complex Division	→	4.99 seconds
c. Complex Division + -fp-model=fast	→	5.30 seconds

Approximation:

Wall Time:

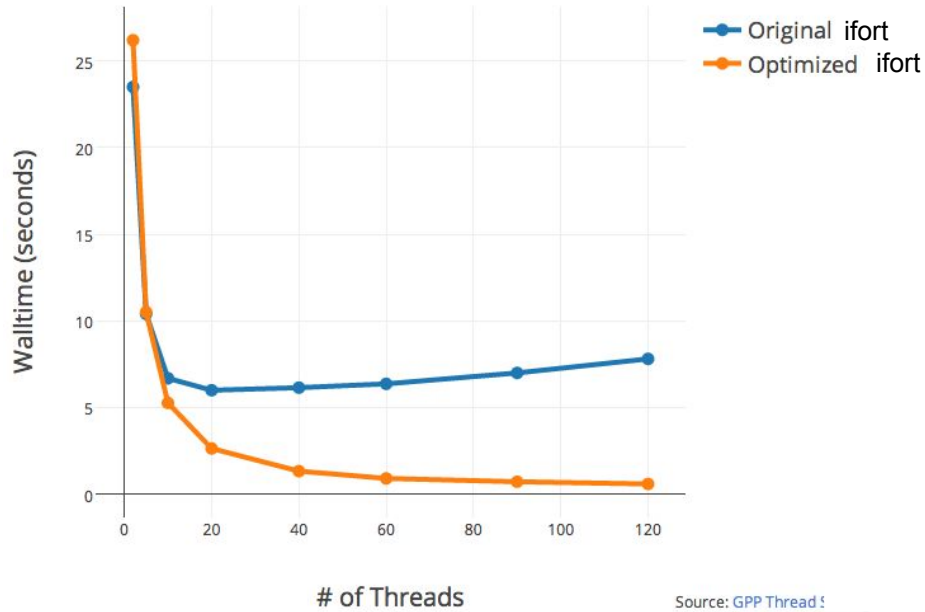
a. Real Division	6.37 seconds
b. Complex Division	4.99 seconds
c. Complex Division + - fp-model fast=2	5.30 seconds
d. Complex Division + - fp-model=fast=2 + ! dir\$ nounroll	4.89 seconds



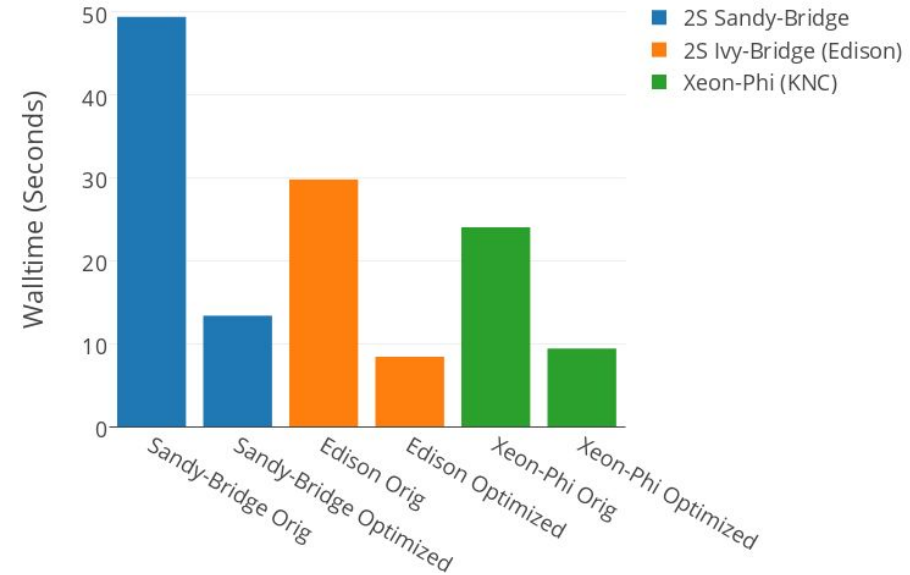
# Early NESAP (Advances with Cray and Intel) Advances



Thread Scaling in BerkeleyGW GPP Kernel on Xeon-Phi



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

# 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 success

Targeting Many-Core greatly helps performance back on Xeon.

Complex division is slow on (particularly on KNC)

BGW 1.0 vs 1.1 Sigma Performance

