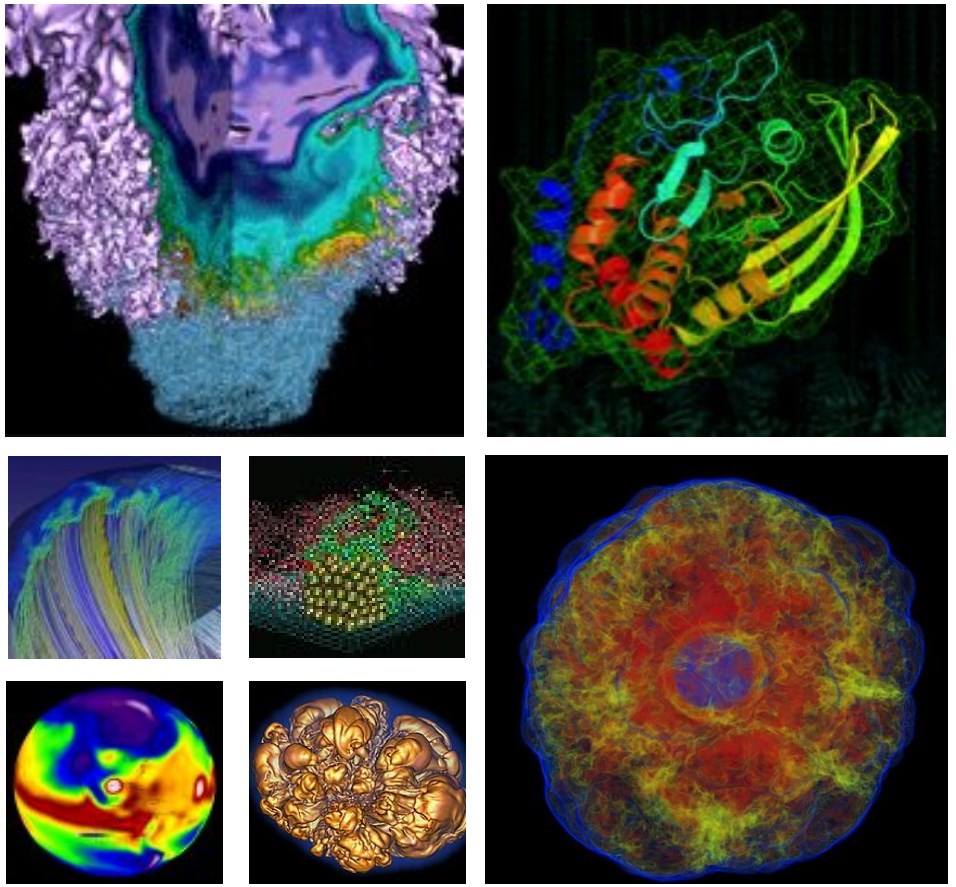
Optimizing Codes For Intel Xeon Phi



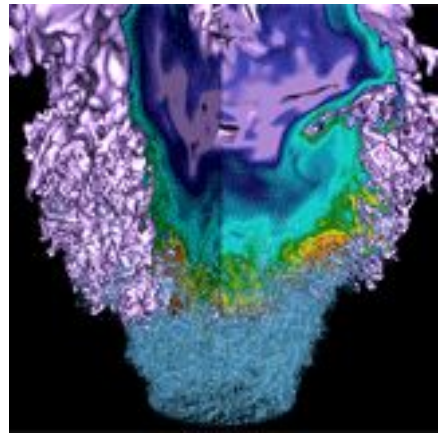
Brian Friesen
NERSC

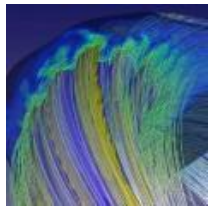
2017 July 26

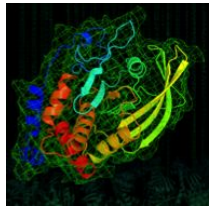
U.S. DEPARTMENT OF ENERGY | Office of Science

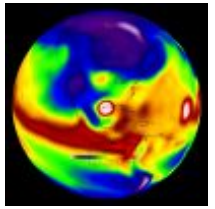
BERKELEY LAB
Lawrence Berkeley National Laboratory

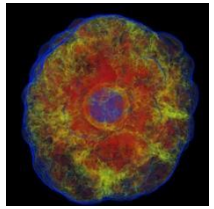
Cori

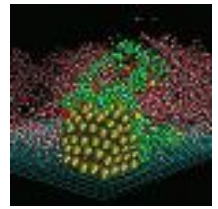












NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

What is different about Cori?

- **Cori is transitioning the NERSC workload to more energy efficient architectures**
- **Cray XC40 system with 9688 Intel Xeon Phi ("Knights Landing") compute nodes**
  - (also 2388 HSW nodes)
  - Self-hosted (not an accelerator), manycore processor with 68 cores per node
  - 16 GB high-bandwidth memory
- **Data Intensive Science Support**
  - 1.5 PB NVRAM burst buffer to accelerate applications
  - 28PB of disk and >700 GB/sec I/O bandwidth

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

What is different about Cori?

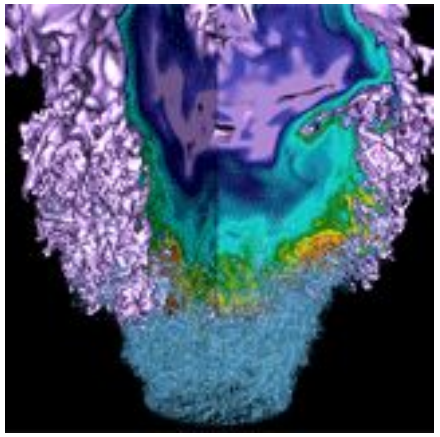
Edison ("Ivy Bridge"):

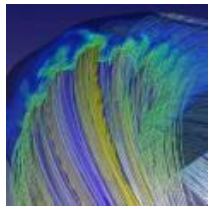
- 12 cores/socket
- 24 hardware threads/socket
- 2.4-3.2 GHz
- Can do 4 Double Precision Operations per Cycle (+ multiply/add)
- 2.5 GB of Memory Per Core
- ~100 GB/s Memory Bandwidth

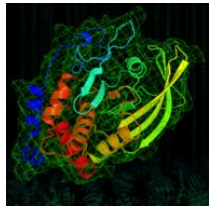
Cori ("Knights Landing"):

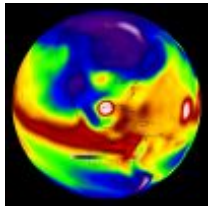
- 68 cores/socket
- 272 hardware threads/socket
- 1.2-1.4 GHz
- Can do 8 Double Precision Operations per Cycle (+ multiply/add)
- < 0.3 GB of Fast Memory Per Core
  < 2 GB of Slow Memory Per Core
- Fast memory has ~ 5x DDR4 bandwidth (~ 460 GB/s)

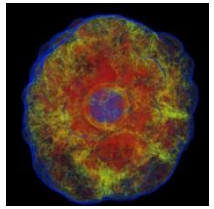
Basic Optimization Concepts

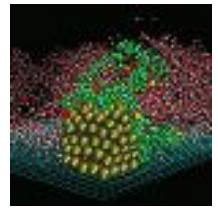






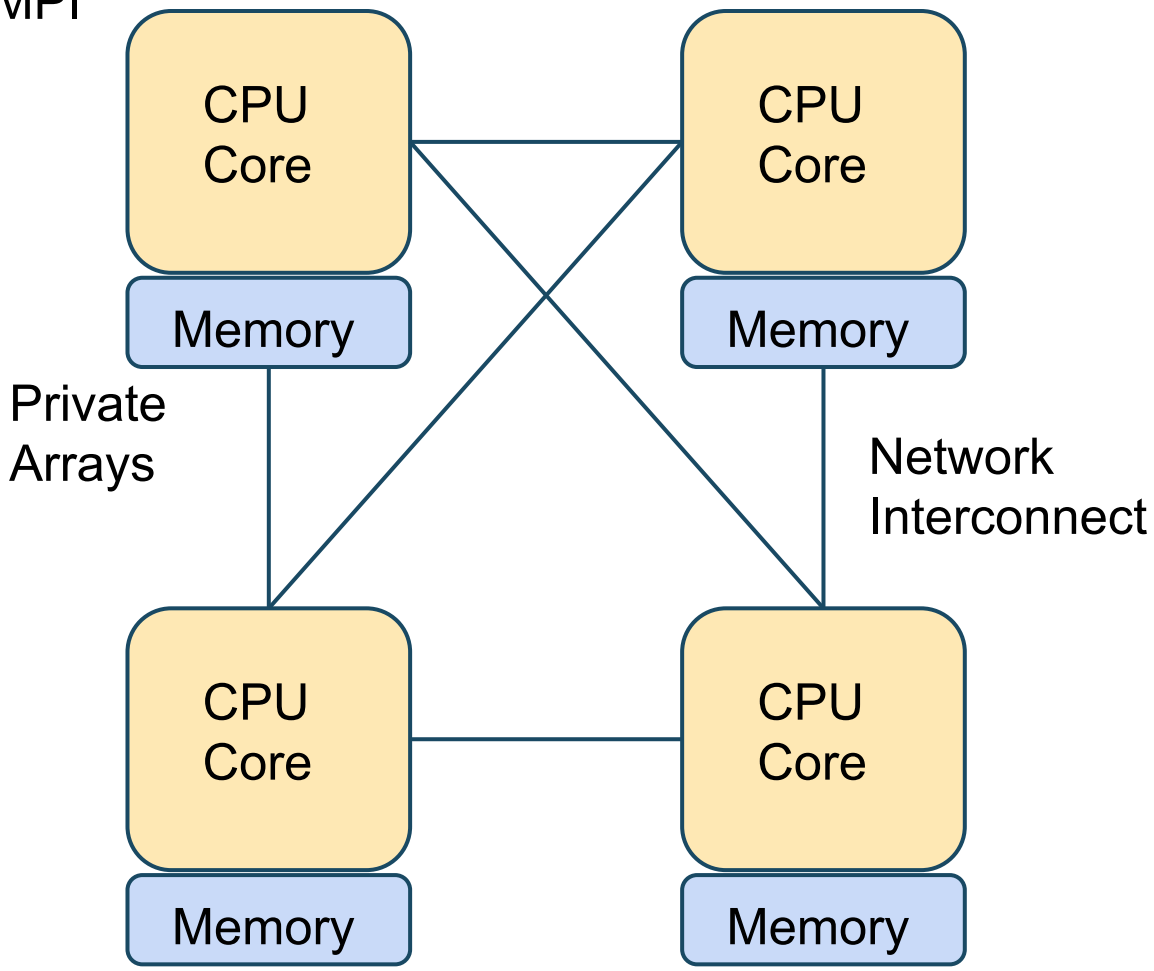




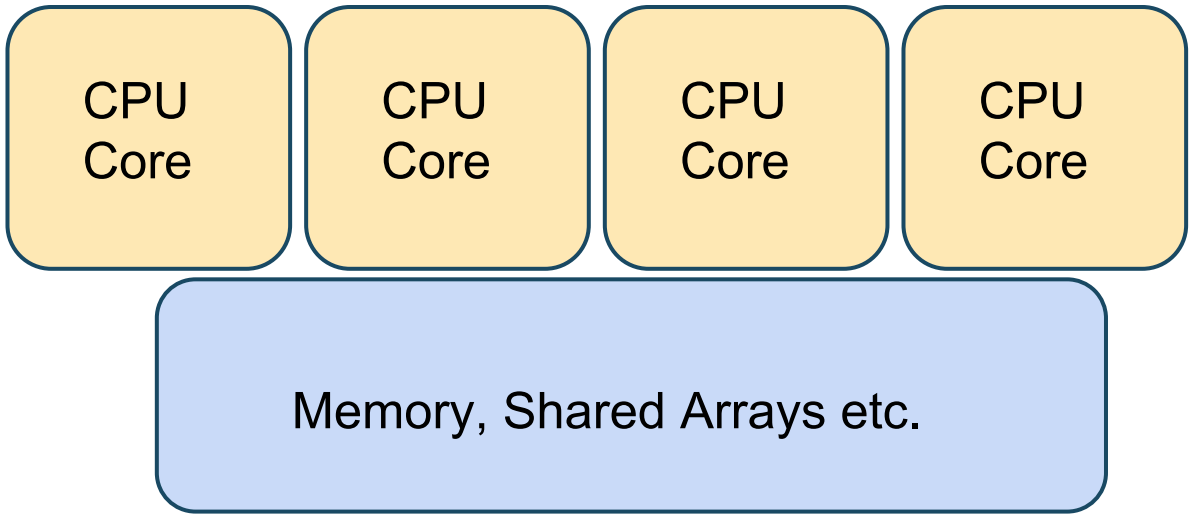


MPI Vs. OpenMP For Multi-Core Programming

MPI



OpenMP



Typically less memory overhead/duplication. Communication often implicit, through cache coherency and runtime

OpenMP Syntax Example

```
      INTEGER I, N
      REAL A(100), B(100), TEMP, SUM

!$OMP PARALLEL DO PRIVATE(TEMP) REDUCTION(+:SUM)
      DO I = 1, N
        TEMP = I * 5
        SUM = SUM + TEMP * (A(I) * B(I))
      ENDDO

      ...
```

https://computing.llnl.gov/tutorials/openMP/exercise.html

Vectorization

There is a another important form of on-node parallelism

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

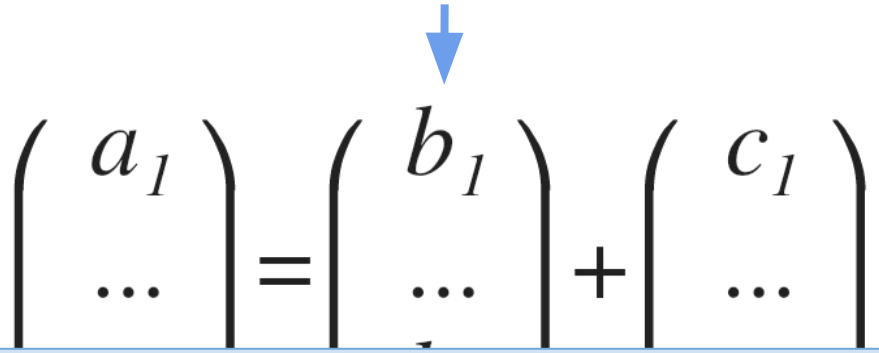
$$\begin{pmatrix} a_1 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \dots \\ b_n \end{pmatrix} + \begin{pmatrix} c_1 \\ \dots \\ c_n \end{pmatrix}$$

Vectorization: CPU does identical operations on different data; e.g., multiple iterations of the above loop can be done concurrently.

Vectorization

There is a another important form of on-node parallelism

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



Vectori of the above l

Intel Xeon Sandy Bridge/Ivy Bridge:	4 Double Precision Ops Concurrently
Intel Xeon Phi:	8 Double Precision Ops Concurrently
NVIDIA Pascal GPUs:	3000+ CUDA cores

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
```

The compiler will happily tell you how it feels about your code

Happy:

```
LOOP BEGIN at hack-a-kernel.f90(212,15)
<Peeled loop for vectorization>
   remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at hack-a-kernel.f90(212,15)
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at hack-a-kernel.f90(212,15)
<Remainder loop for vectorization>
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

The compiler will happily tell you how it feels about your code

Sad:

```
Non-optimizable loops:


LOOP BEGIN at hack-a-kernel.f90(252,7)
   remark #15543: loop was not vectorized: loop with function call not considered an optimization candidate.
LOOP END
```

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,

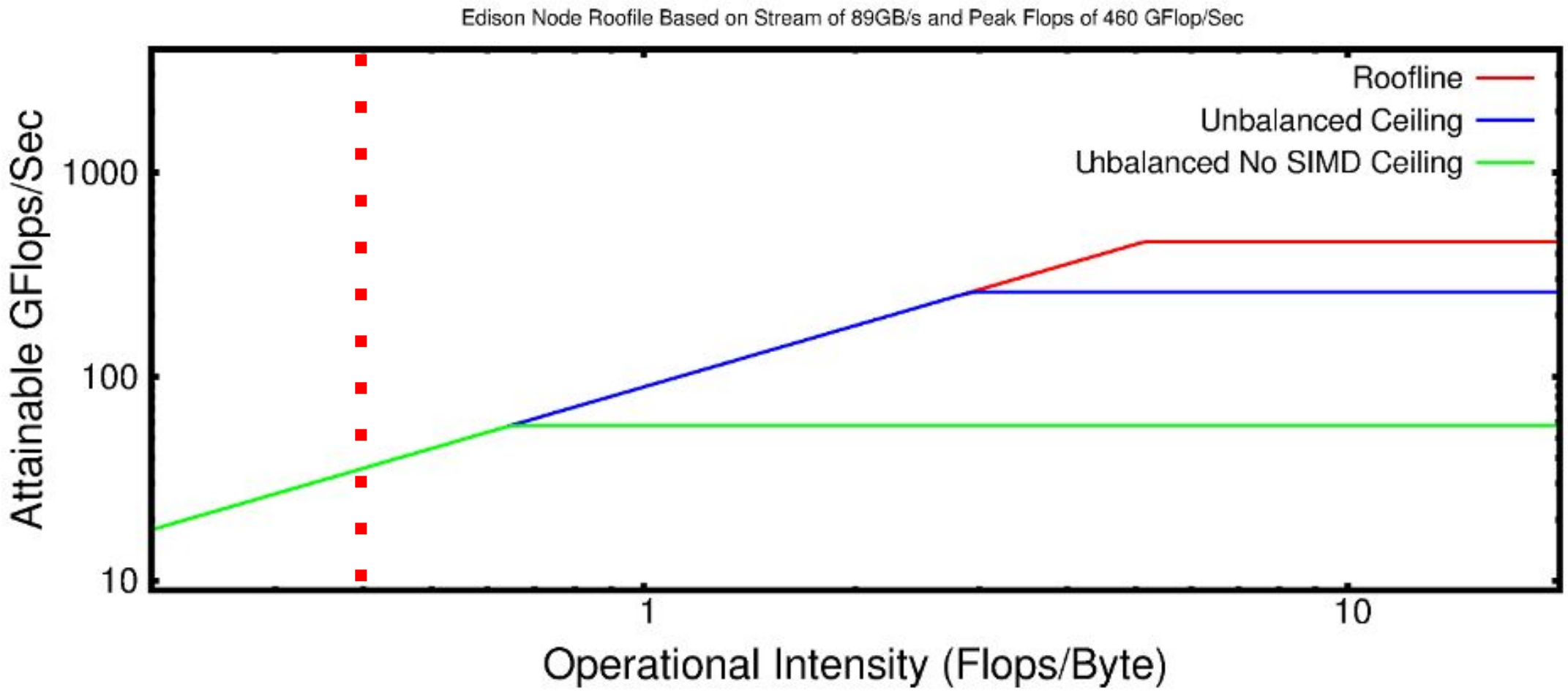
During execution, the **number of loads From DRAM** is

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



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:

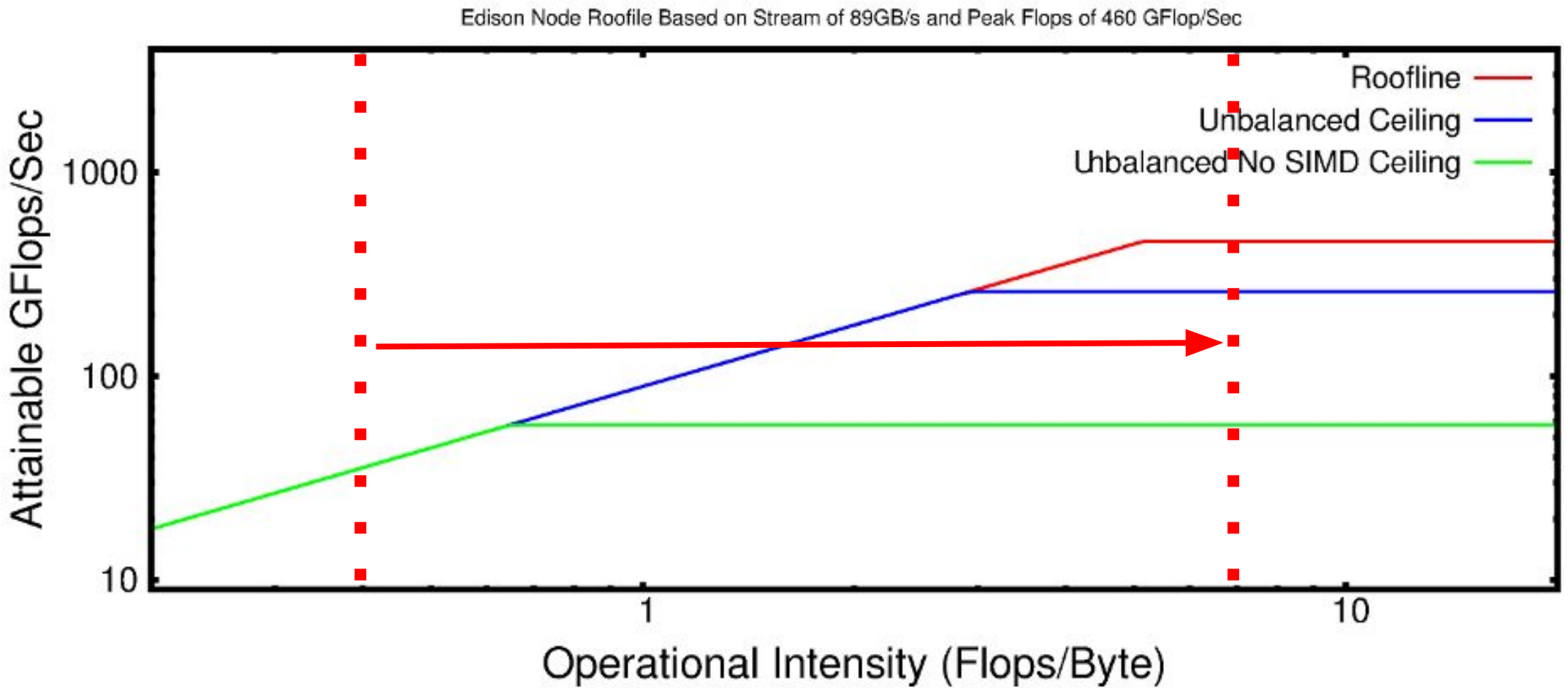
n*m + n

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

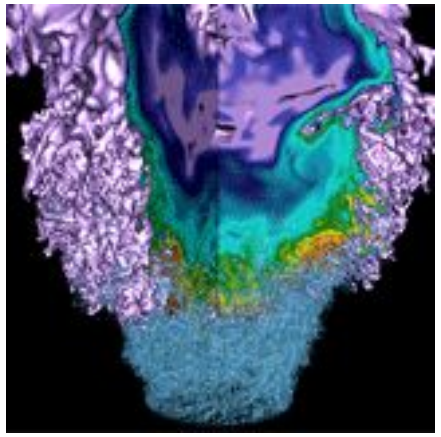
Loads From DRAM:

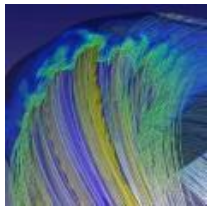
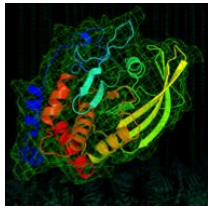
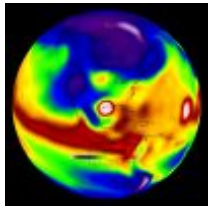
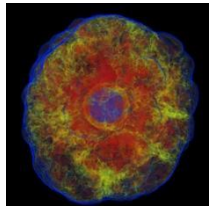
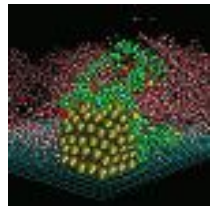
m/block * (n+block)
= **n*m/block** + m

Improving Memory Locality Moves you to the Right on the Roofline



Optimization Strategy



NERSC 40 YEARS at the FOREFRONT

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

How how to let profiling guide your optimization (with VTune)

- Start with the "`general-exploration`" collection
  - nice high-level summary of code performance
  - identifies most time-consuming loops in the code
  - tells you if you're compute-bound, memory bandwidth-bound, etc.
- If you are memory bandwidth-bound:
  - run the "`memory-access`" collection
    - lots more detail about memory access patterns
    - which variables are responsible for all the bandwidth
- If you are compute-bound:
  - run the "`hotspots`" or "`advanced-hotspots`" collection
    - will tell you how busy your OpenMP threads are
    - Will isolate the longest-running sections of code

Measuring Your Memory Bandwidth Usage (VTune)

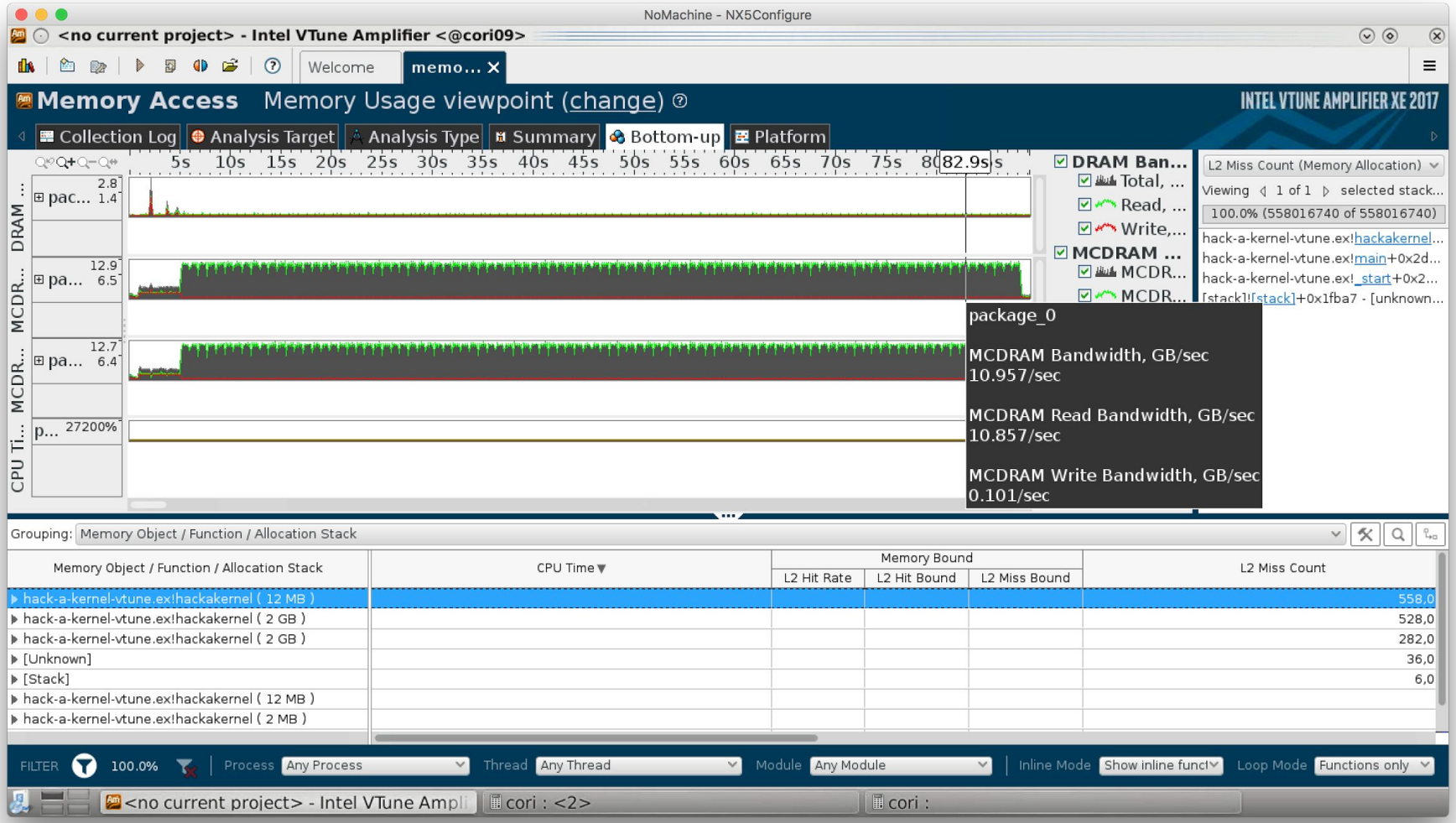
Measure memory bandwidth usage in VTune.

Compare to Stream GB/s.

Peak DRAM bandwidth is ~100 GB/s

Peak MCDRAM bandwidth is ~400 GB/s

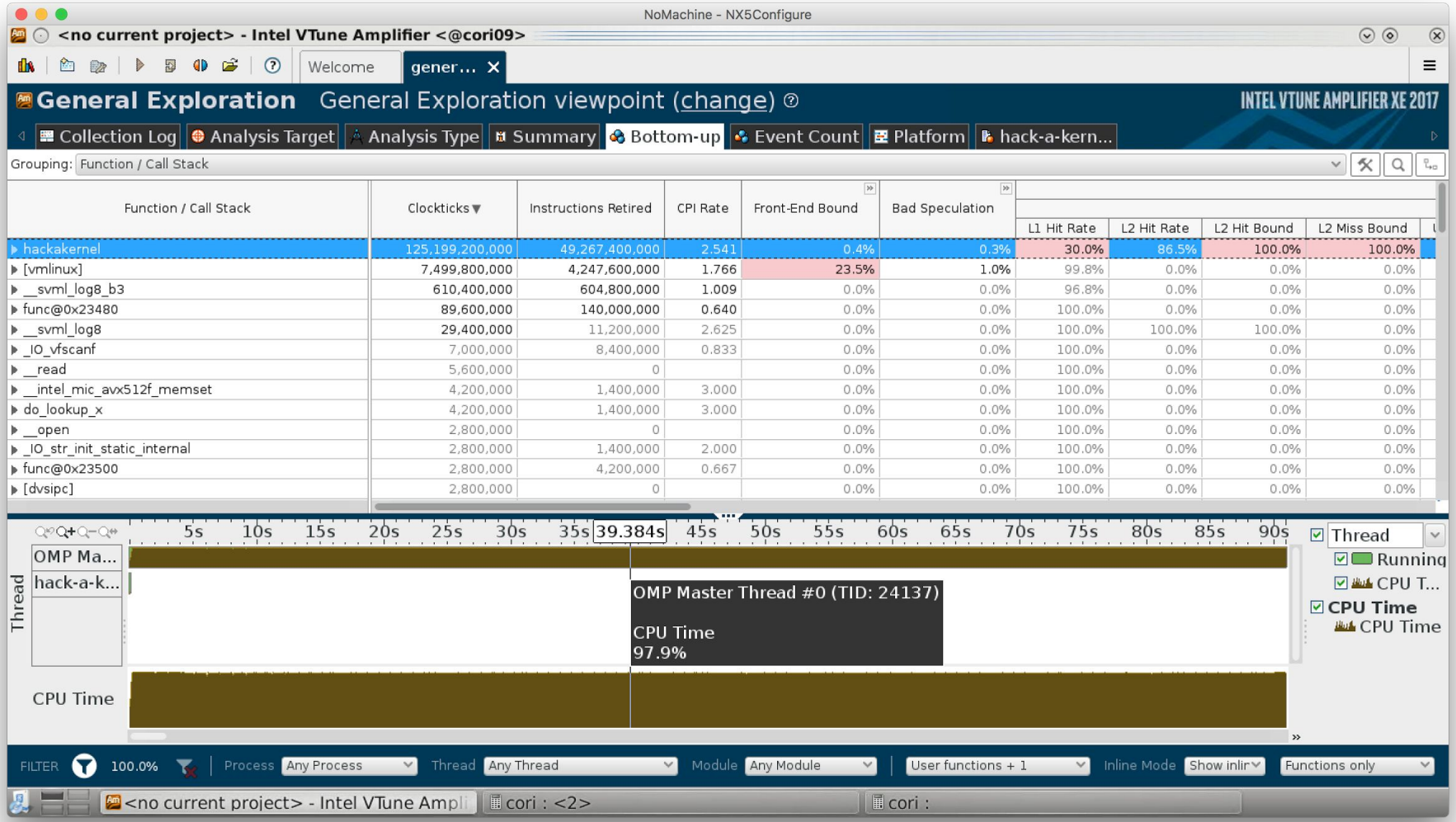
If 90% of stream, you are memory bandwidth bound.



Measuring Code Hotspots (VTune)

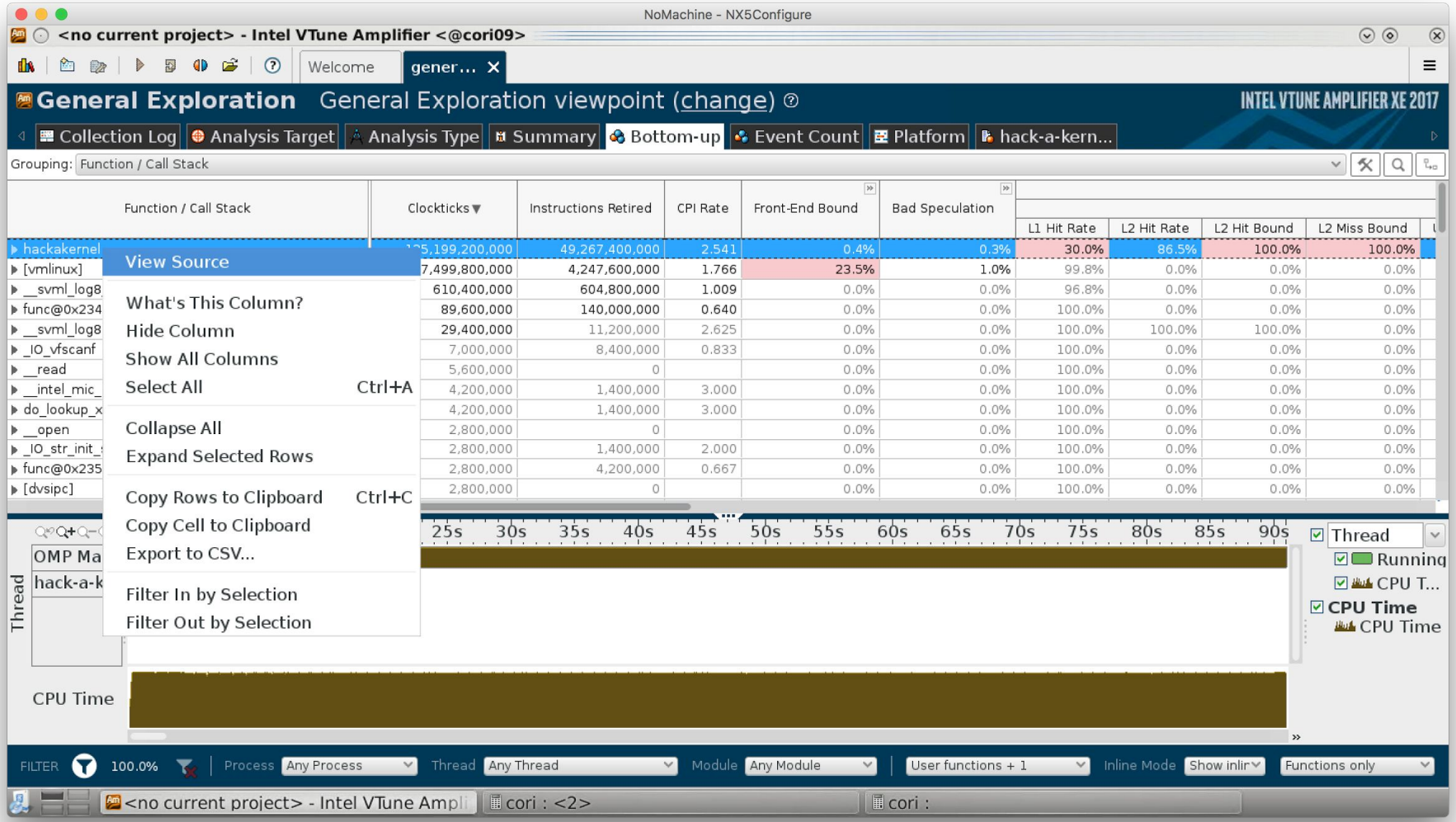
"general-exploration" and "hotspots" collections tell you which lines of code take the most time

Click "bottom-up" tab to see the most time-consuming parts of the code



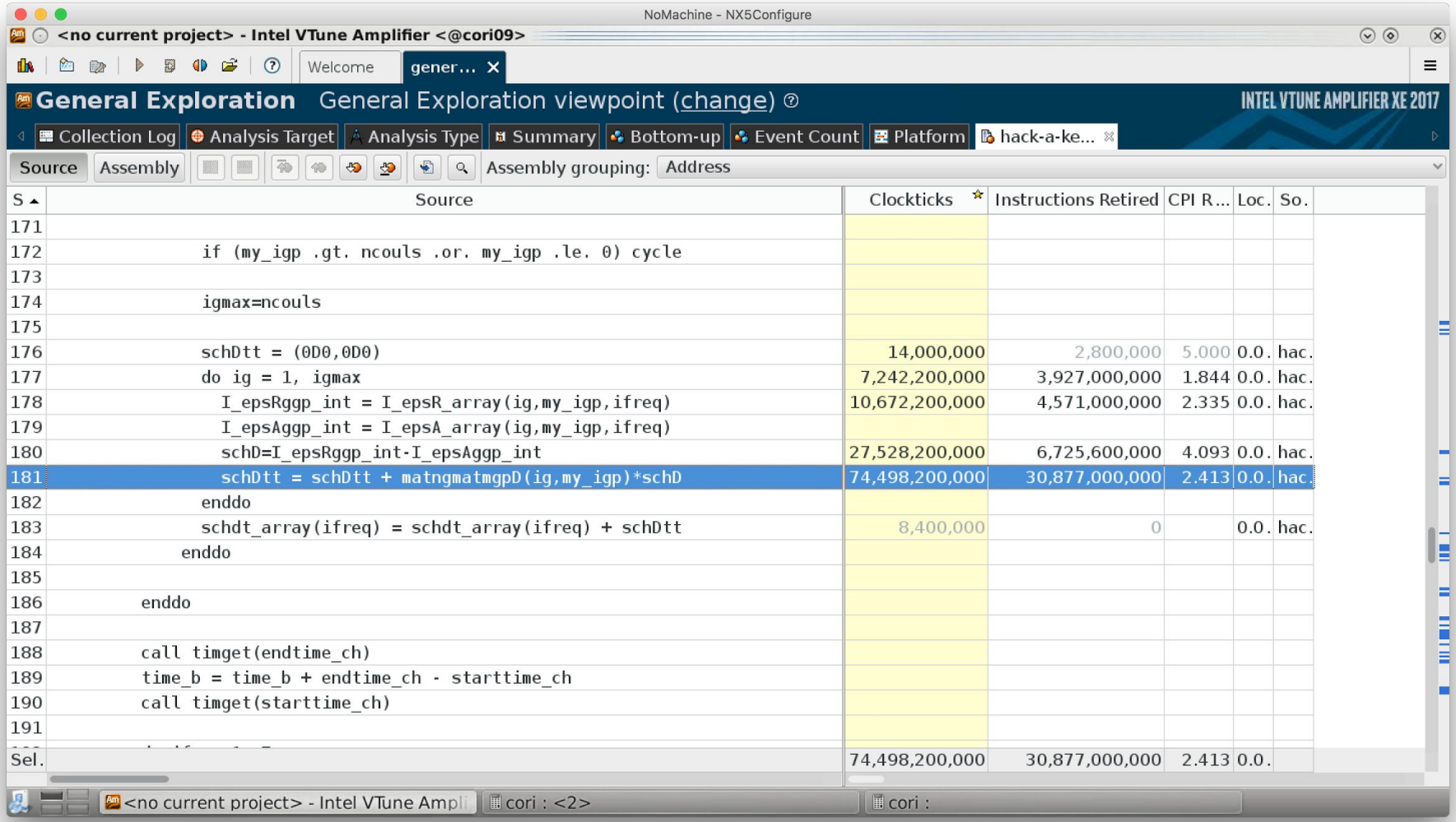
Measuring Code Hotspots (VTune)

Right-click on a row in the “bottom-up” view to navigate directly to the source code



Measuring Code Hotspots (VTune)

Here you can see the raw source code that takes the most execution time



S	Source	Clockticks	Instructions Retired	CPI R...	Loc.	So.
171						
172	`if (my_igp .gt. ncouls .or. my_igp .le. 0) cycle`					
173						
174	`igmax=ncouls`					
175						
176	`schDtt = (0D0,0D0)`	14,000,000	2,800,000	5.000	0.0.	hac.
177	`do ig = 1, igmax`	7,242,200,000	3,927,000,000	1.844	0.0.	hac.
178	`I_epsRggp_int = I_epsR_array(ig,my_igp,ifreq)`	10,672,200,000	4,571,000,000	2.335	0.0.	hac.
179	`I_epsAggp_int = I_epsA_array(ig,my_igp,ifreq)`					
180	`schD=I_epsRggp_int-I_epsAggp_int`	27,528,200,000	6,725,600,000	4.093	0.0.	hac.
181	`schDtt = schDtt + matngmatmgpD(ig,my_igp)*schD`	74,498,200,000	30,877,000,000	2.413	0.0.	hac.
182	`enddo`					
183	`schdt_array(ifreq) = schdt_array(ifreq) + schDtt`	8,400,000	0		0.0.	hac.
184	`enddo`					
185						
186	`enddo`					
187						
188	`call timget(endtime_ch)`					
189	`time_b = time_b + endtime_ch - starttime_ch`					
190	`call timget(starttime_ch)`					
191						
Sel.		74,498,200,000	30,877,000,000	2.413	0.0.	

There are scripts!

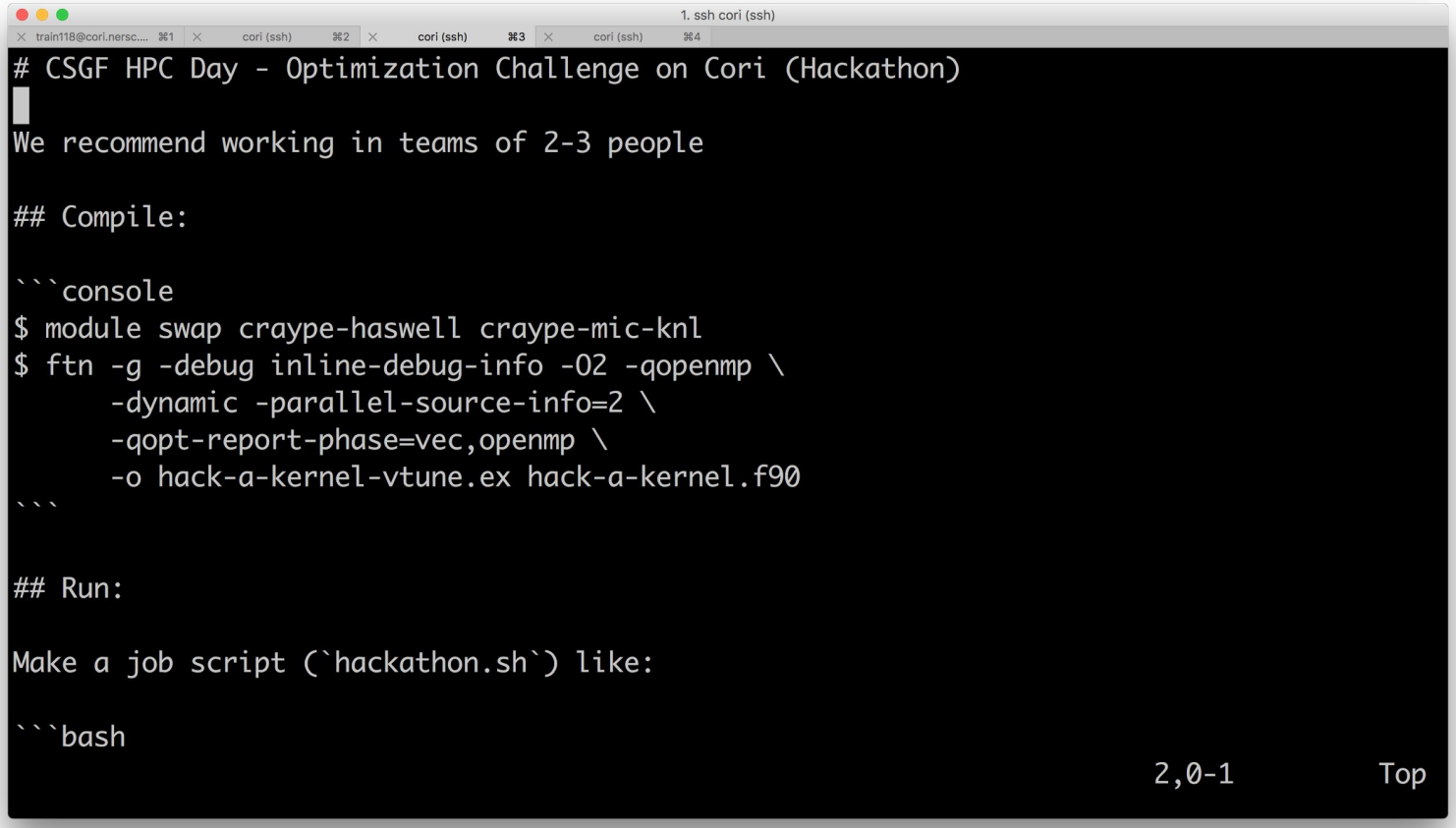
- There are now be some scripts in `train/csgf-hack-day/hack-a-kernel` which do different VTune collections
- If you don't see these new scripts (e.g., "`general-exploration_part_1_collection.sh`"), then you can update your git repository and they will show up:
  - `git pull origin/master`

There are scripts!

- Edit the "`part_1`" script to point to the correct location of your executable (very bottom of the script)
- Submit the "`part_1`" scripts with `sbatch`
- Edit the "`part_2`" scripts to point to the dir where you saved your VTune collection in `part_1`
- Run the "`part_2`" scripts with bash:
  - `bash hotspots_part_2_finalize.sh`
- Launch the VTune GUI from NX with `amplxe-gui <collection_dir>`

How to compile the kernel for VTune profiling

- In the `hack-a-kernel` dir, there is a `README-rules.md` file which shows how to compile:



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

```
srun -n 68 --ntasks-per-node=32 ...
```

VS

```
srun -n 68 ...
```

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

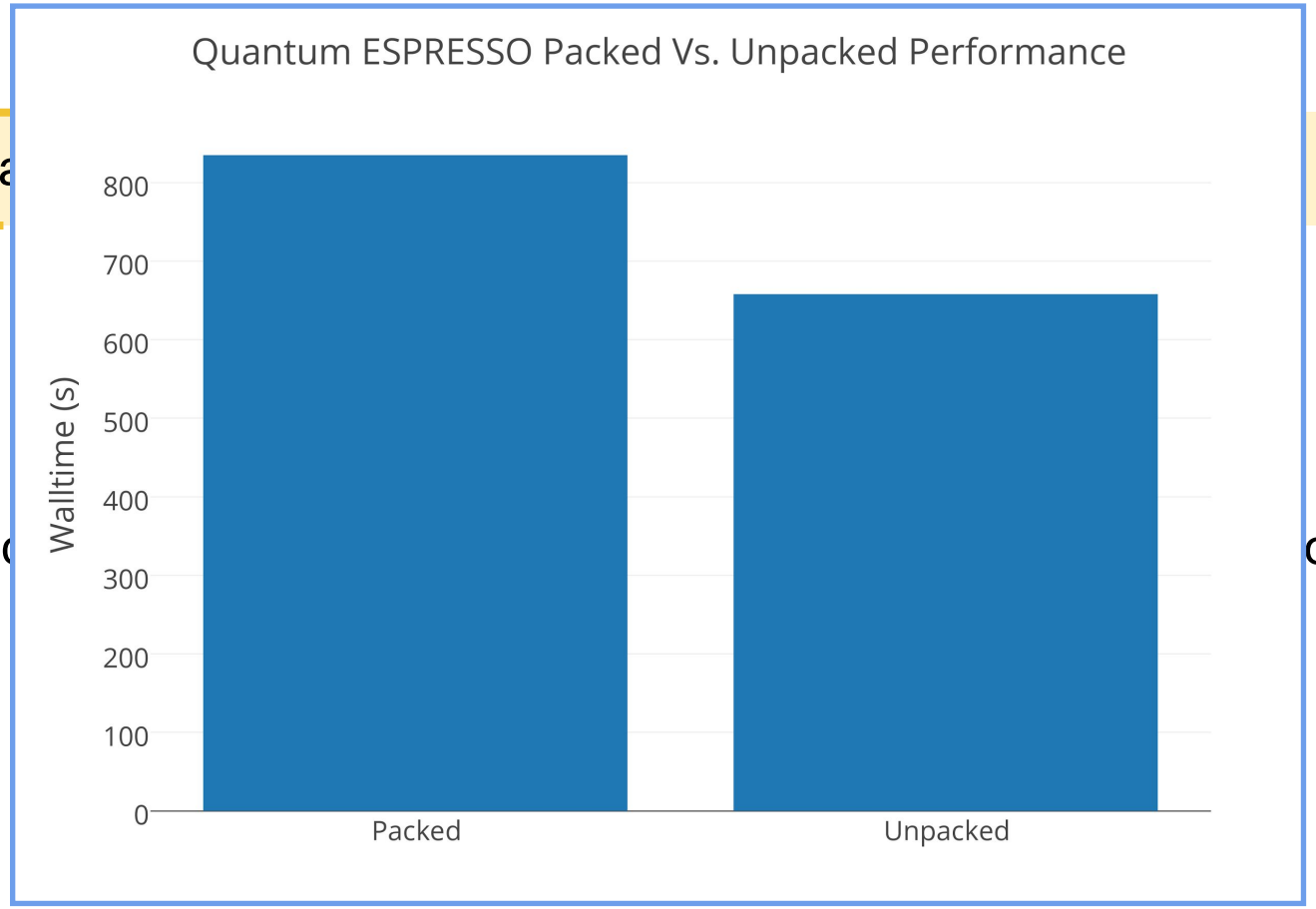
Are you memory or compute bound? Or both?

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

```
srun -n 68 --nta
...
```

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.

```
srun --cpu-freq=1200000 ...
```

VS

```
srun --cpu-freq=1000000 ...
```

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

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 Cori improves performance, you *might* be latency bound:

`srun -n 136 -c 2 ….` VS `srun -n 68 -c 4 ….`

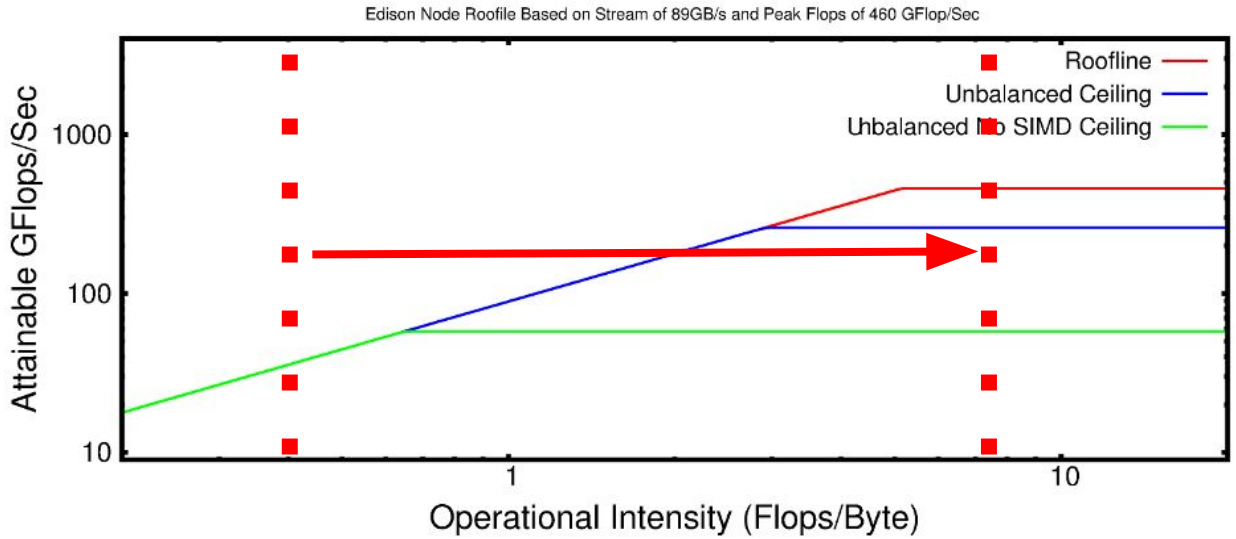
If you can, try to reduce the number of memory requests per flop by accessing contiguous and predictable segments of memory and reusing variables in cache as much as possible.

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

So, you are Memory Bandwidth Bound?

What to do?

1. Try to improve memory locality, cache reuse



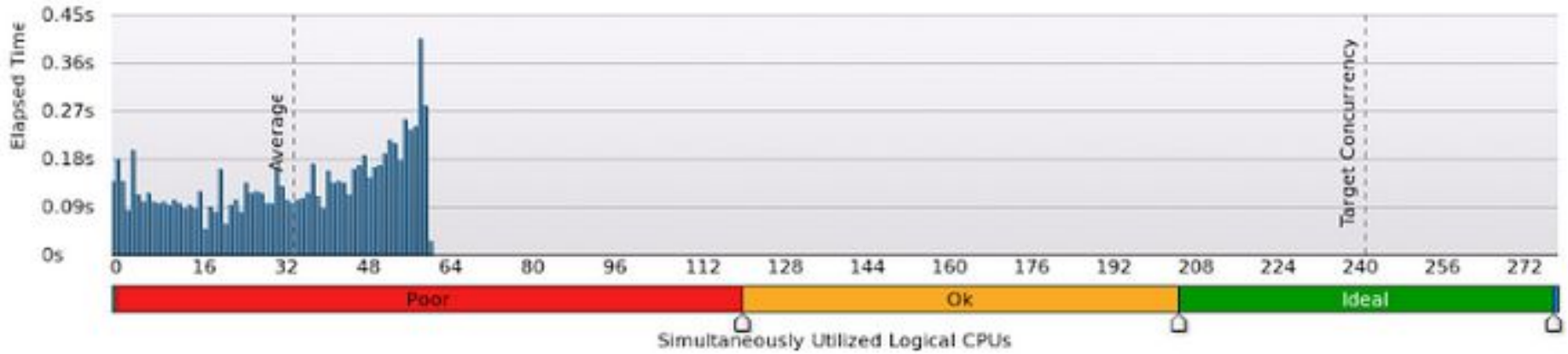
2. Identify the key arrays leading to high memory bandwidth usage and make sure they are/will-be allocated in HBM on Cori.

   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-phase=vec`

Extra Slides

Steps:

0. Use NX To Login Onto Cori

https://www.nersc.gov/users/connecting-to-nersc/using-nx/

1. Get Code:

% git clone https://github.com/NERSC/train.git

2. Build Code:

```
% cd training/csgf-hpc-day/hack-a-kernel
%ftn -g -debug inline-debug-info -O2 -qopenmp \
     -dynamic -parallel-source-info=2 \
     -qopt-report-phase=vec,openmp \
     -o hack-a-kernel-vtune.ex hack-a-kernel.f90
```

3. Run Code (interactively):

```
% salloc -N 1 --reservation-csgftrain -C knl -t 30
% ./hack-a-kernel.ex
```

4. Collect hotspots with vtune (in an interactive session):

```
% module load vtune
% amplxe-cl -collect hotspots -r test_1 -- ./bgw.x
```

5. To View Vtune Results do:

% amplxe-gui

Question 1 - Can you make the code faster by adding OpenMP to any hot loop?

!$OMP PARALLEL do private (...) ...

6. Collect collect bandwidth information d (in an interactive session):

% amplxe-cl -collect bandwidth -r *test_2* -- ./bgw.x
… then view the output in the GUI

Question 2 - Is the code memory bandwidth bound?

Question 3 - Can you improve the code performance further through any optimization strategy described at the beginning of the session?

Steps:

0. Use NX To Login Onto Babbage

https://www.nersc.gov/users/connecting-to-nersc/using-nx/

1. Get Code:

% git clone https://github.com/NERSC/training.git

2. Build Code:

% cd training/hackathon-201502/BGW
% ifort -g -O3 -xAVX -openmp bgw.f90 -o bgw.x

3. Run Code (interactively):

% salloc --nodes=1 --time=00:30:00
wait....
% srun ./bgw.x

4. Collect hotspots with vtune (in an interactive session):

% module load vtune
% srun amplxe-cl -collect hotspots -r *test_1* -- ./bgw.x

5. To View Vtune Results do:

% [srun] amplxe-gui

Question 1 - Can you make the code faster by adding OpenMP to any hot loop?

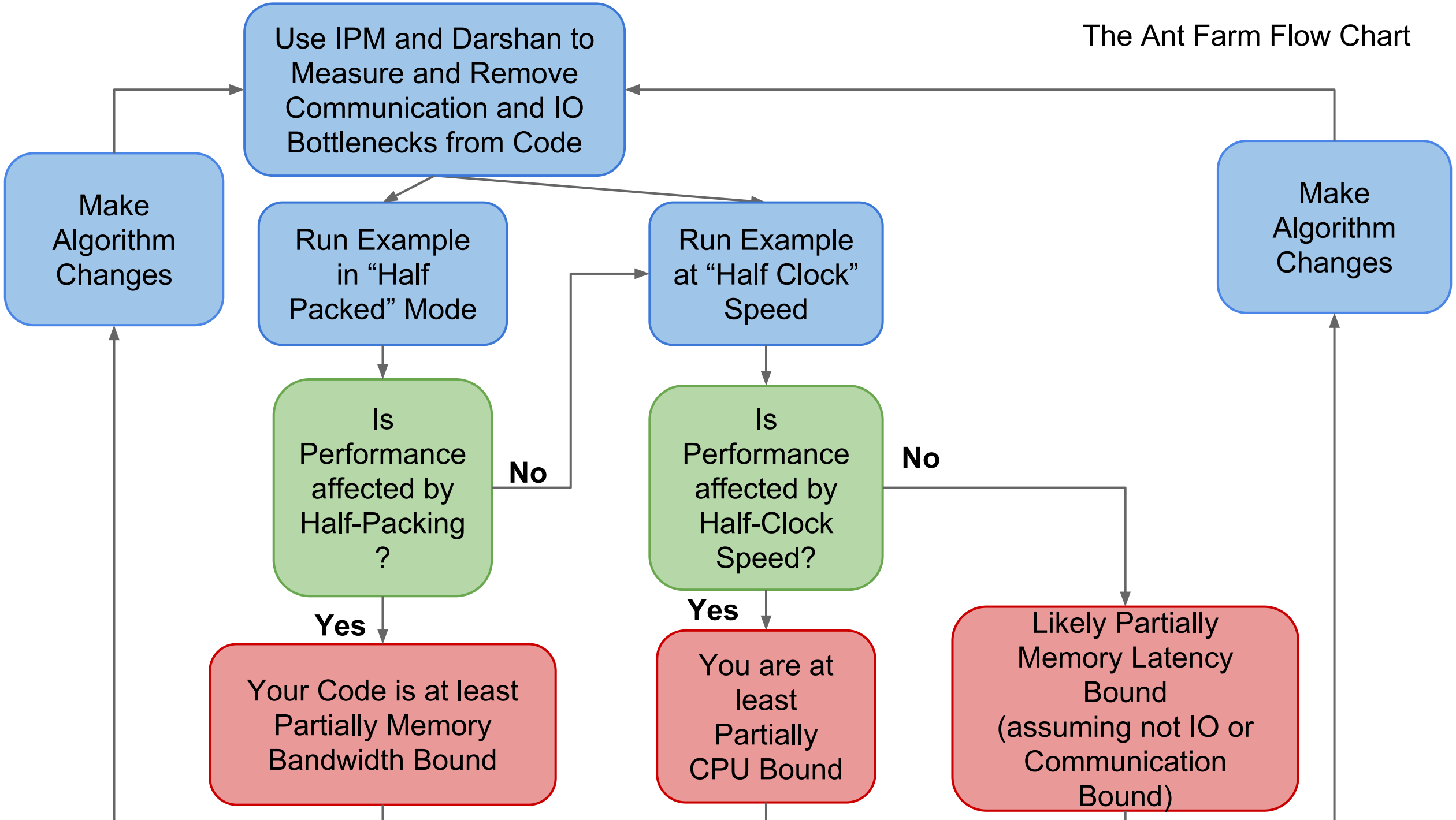
!$OMP PARALLEL do private (...) ...

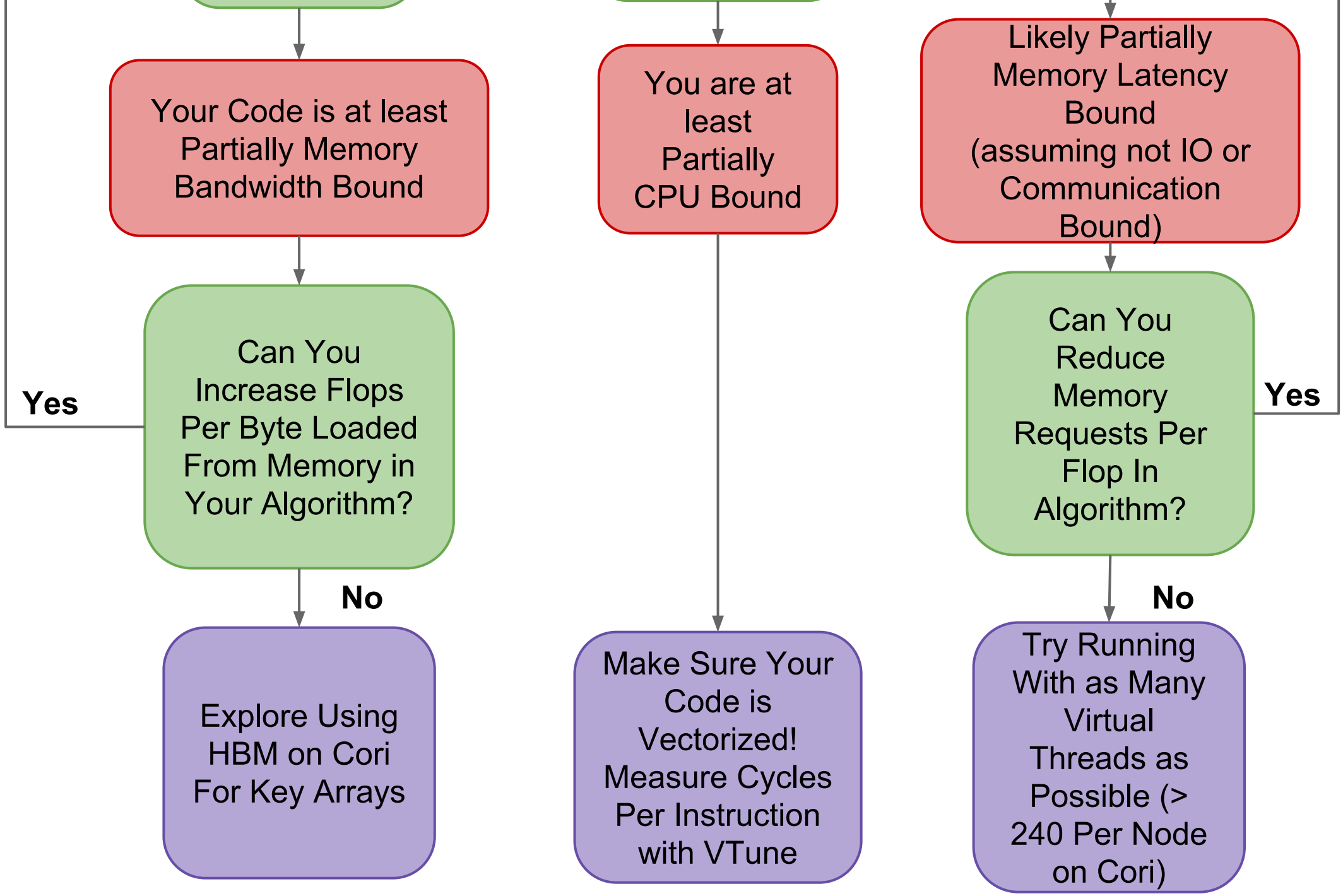
6. Collect collect bandwidth information d (in an interactive session):

% srun amplxe-cl -collect bandwidth -r *test_2* -- ./bgw.x
… then view the output in the GUI

Question 2 - Is the code memory bandwidth bound?

Question 3 - Can you improve the code performance further through any optimization strategy described at the beginning of the session?


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
Run Example at “Half Clock” Speed
Make Algorithm Changes
Is Performance affected by Half-Packing ?
No
Is Performance affected by Half-Clock Speed?
No
Yes
Yes
Your Code is at least Partially Memory Bandwidth Bound
You are at least Partially CPU Bound
Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)


Your Code is at least Partially Memory Bandwidth Bound
You are at least Partially CPU Bound
Likely Partially Memory Latency Bound (assuming not IO or Communication Bound)
Can You Increase Flops Per Byte Loaded From Memory in Your Algorithm?
Yes
No
Can You Reduce Memory Requests Per Flop In Algorithm?
Yes
No
Explore Using HBM on Cori For Key Arrays
Make Sure Your Code is Vectorized! Measure Cycles Per Instruction with VTune
Try Running With as Many Virtual Threads as Possible (> 240 Per Node on Cori)

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
}
```

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.

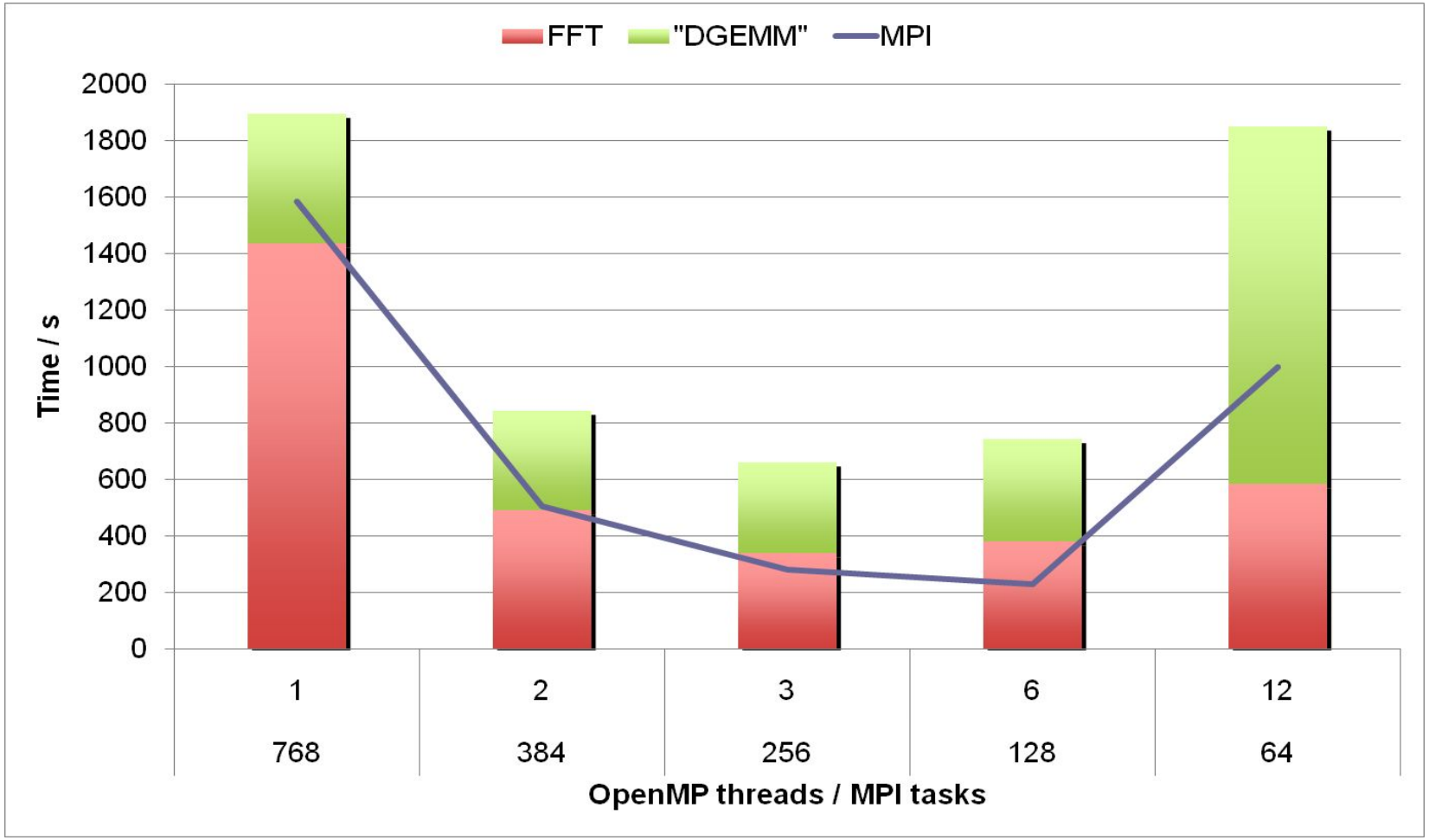


Figure Courtesy of Andrew Canning