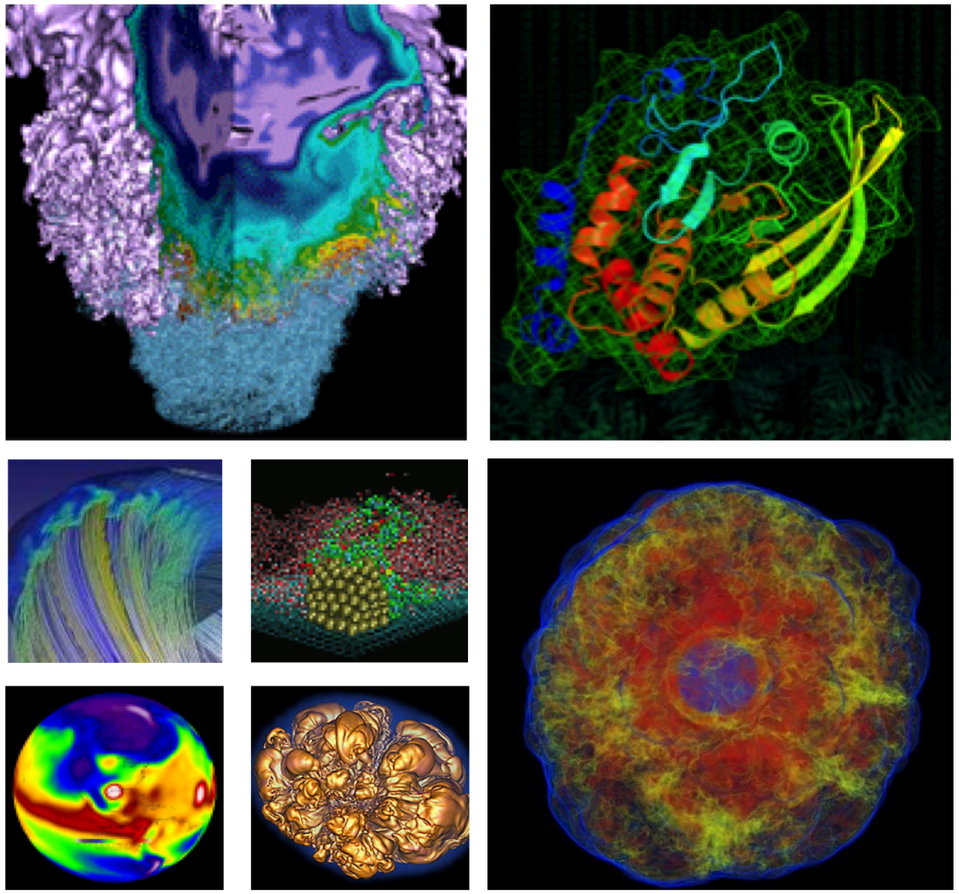
Guiding Optimization on with the Roofline Model



NERSC

February, 2020

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

NERSC's Challenge

How to Enable NERSC's diverse community of 7,000 users, 750 projects, and 700 codes to run on advanced architectures like Cori, Perlmutter and beyond?

What was different about Cori?

Edison ("Ivy Bridge):

- 5576 nodes
- 24 physical cores per node
- 48 virtual cores per node
- 2.4 - 3.2 GHz
- 8 double precision ops/cycle
- 64 GB of DDR3 memory (2.5 GB per physical core)
- ~100 GB/s Memory Bandwidth

Cori ("Knights Landing"):

- 9304 nodes
- 68 physical cores per node
- 272 virtual cores per node
- 1.4 - 1.6 GHz
- 32 double precision ops/cycle
- 16 GB of fast memory
  96GB of DDR4 memory
- Fast memory has 400 - 500 GB/s
- No L3 Cache

Optimization Challenges For Scientists

Science teams need a simple way to wrap their heads around performance when main focus is scientific productivity:

1. Need a sense of absolute performance when optimizing applications.
   - How Do I know if My Performance is Good?
   - Why am I not getting peak performance advertised
   - How Do I know when to stop?

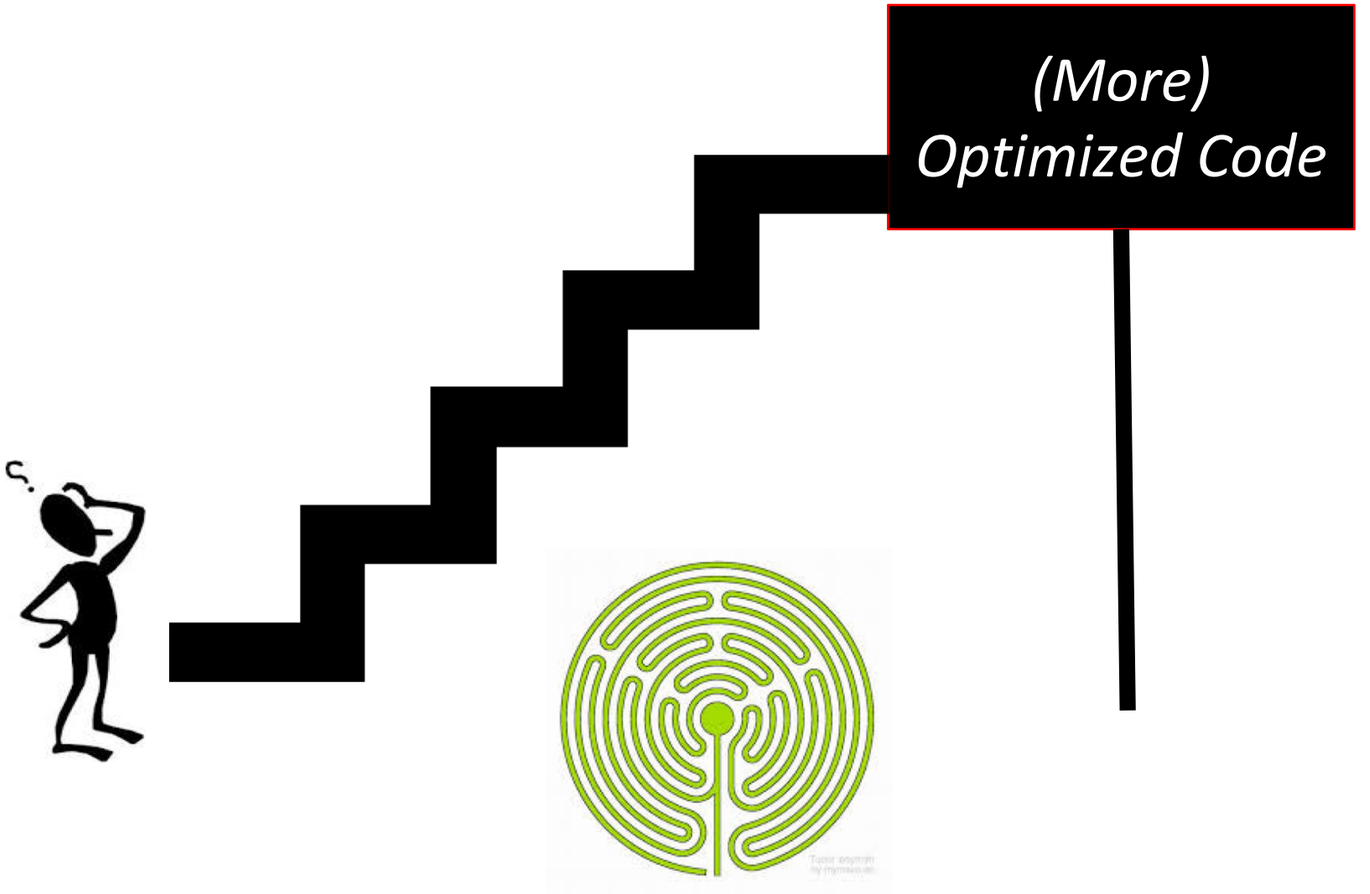
2. Many potential optimization directions:
   - How do I know which to apply?
   - What is the limiting factor in my app's performance?
   - Again, how do I know when to stop?

Optimizing Code For Cori is like:

A. A Staircase ?

A. A Labyrinth ?

A. A Space Elevator?



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

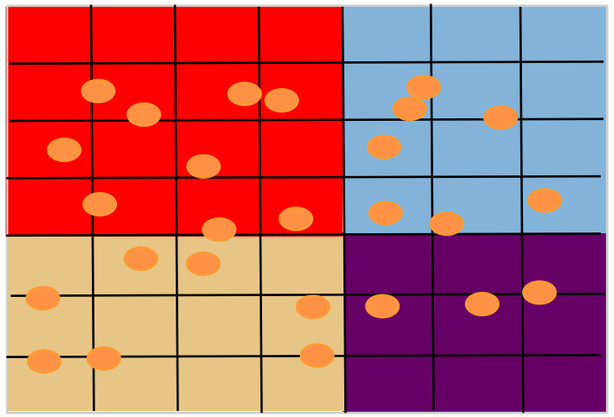
Utilize High-Level IO-Libraries. Consult with NERSC about use of Burst Buffer.

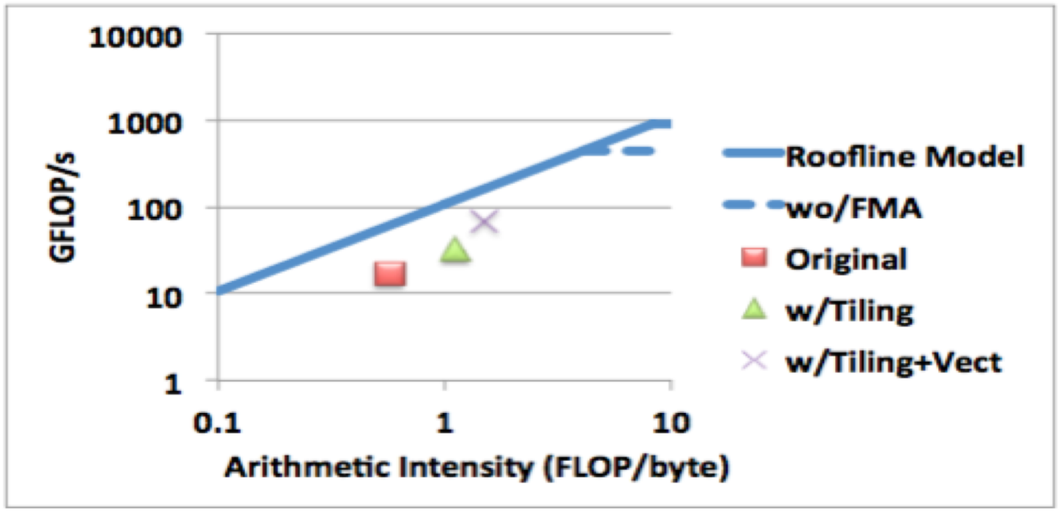
The Dungeon: Simulate kernels on KNL. Plan use of on package memory, vector instructions.

Roofline helps visualize this information! Guides optimizations

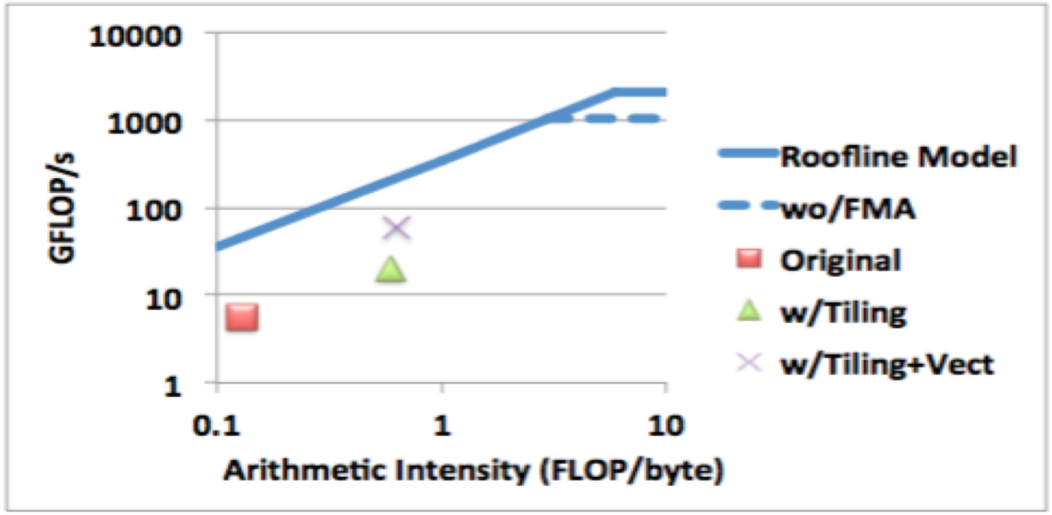
WARP Optimizations:

1. Add tiling over grid targeting L2 cache on both Xeon-Phi Systems
1. Add particle sorting to further improve locality and memory access pattern
1. Apply vectorization over particles



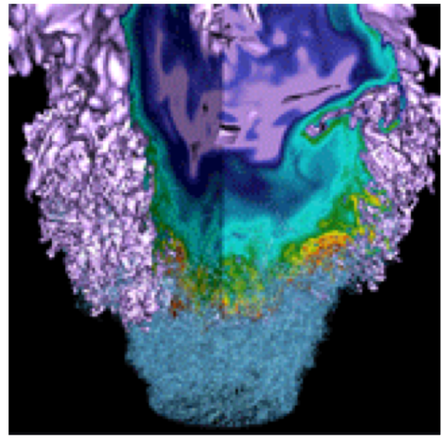


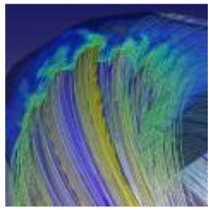
(a) Haswell Roofline

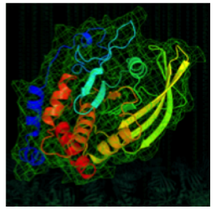


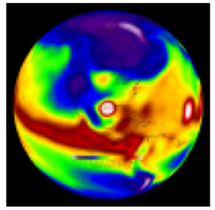
(b) KNL Roofline

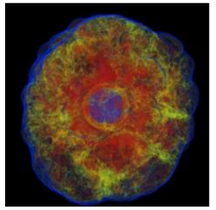
NESAP Example

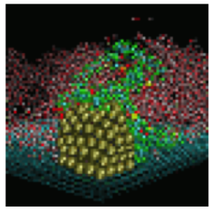












NERSC 40 YEARS at the FOREFRONT

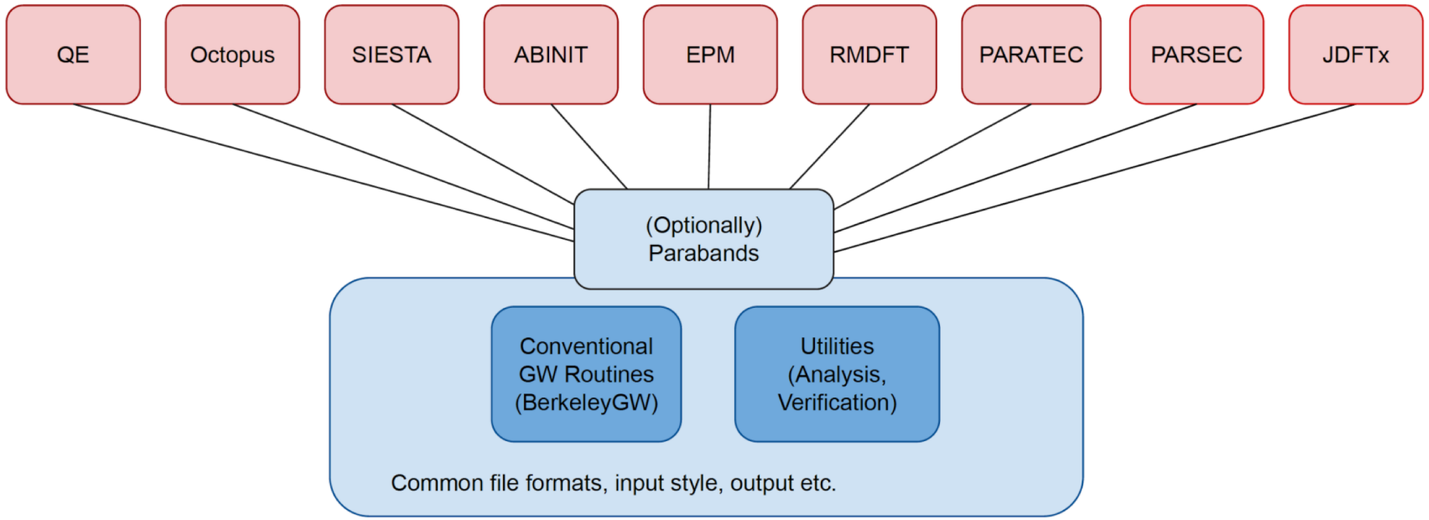
U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

BerkeleyGW

- A massively parallel package for GW calculations
- Sits on top of DFT codes

- Computational motifs
  - FFTs
  - Dense linear algebra
  - Large reductions



BerkeleyGW

C2SEPEM

Sigma-GPP

Pseudo Code

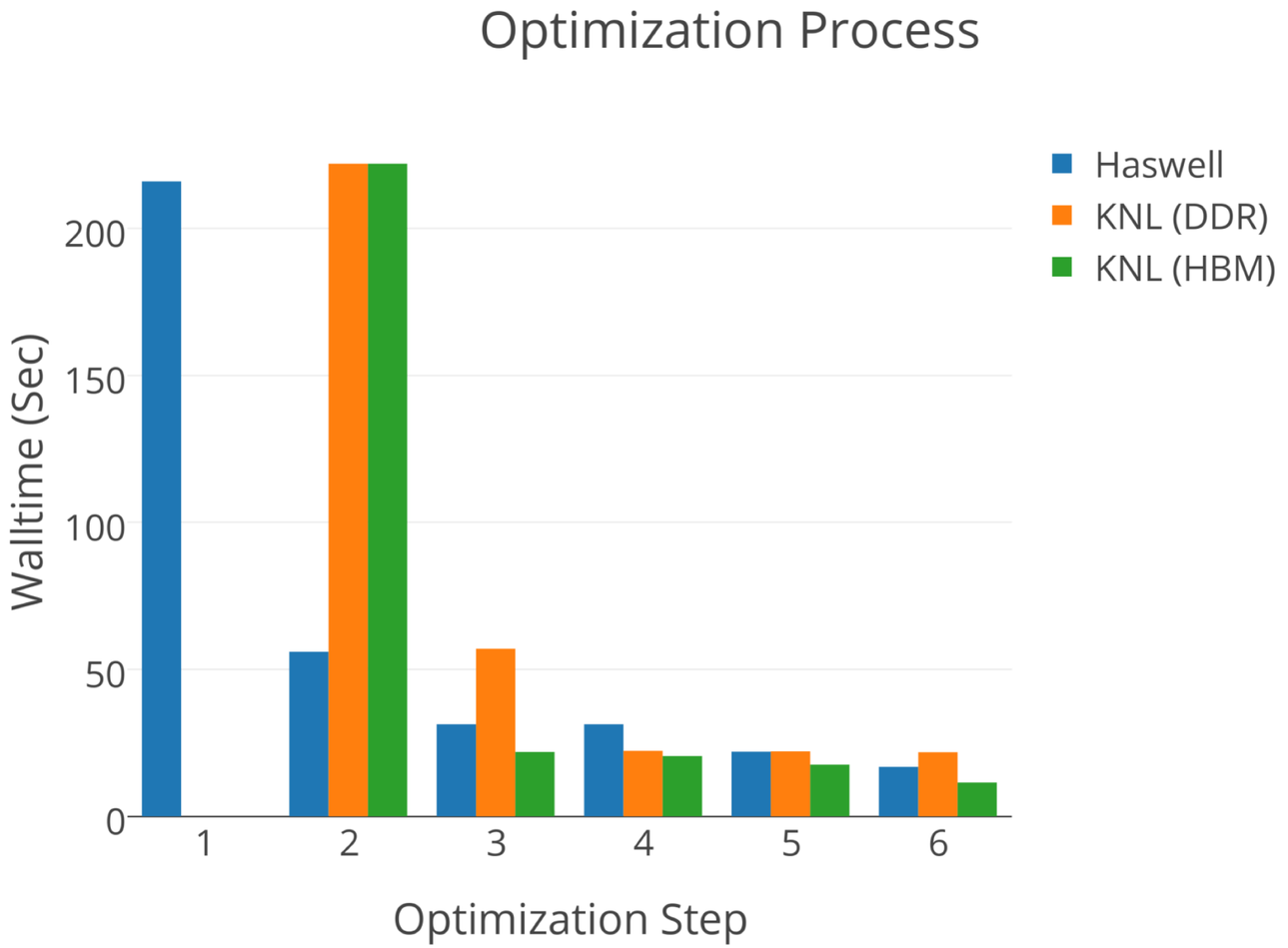
```
do n1 = 1, nbands              n' e.g. 2763
   do igp = 1, ngpown          G' e.g. 6633
      do ig = 1, ncouls        G  e.g. 26529
         do iw = 1, nw         E  e.g. 3
            compute:   1. mixed data types
                          e.g. complex double, double, integer
                       2. various memory access patterns
                          e.g. (ig,igp)(ig,n1)(igp,n1)(iw,n1)(n1)
                       3. complex number divisions
                       4. nw is very small, will be unrolled

            reduction: 1. complex numbers
                       2. all top 3 loops, billions of iterations
```

BerkeleyGW NESAP Project Optimization Path

Optimization process for GPP Kernel

1. Add OpenMP
2. Initial Vectorization (loop reordering, conditional removal)
3. Cache-Blocking
4. Improved Vectorization (Divides)
5. 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
```

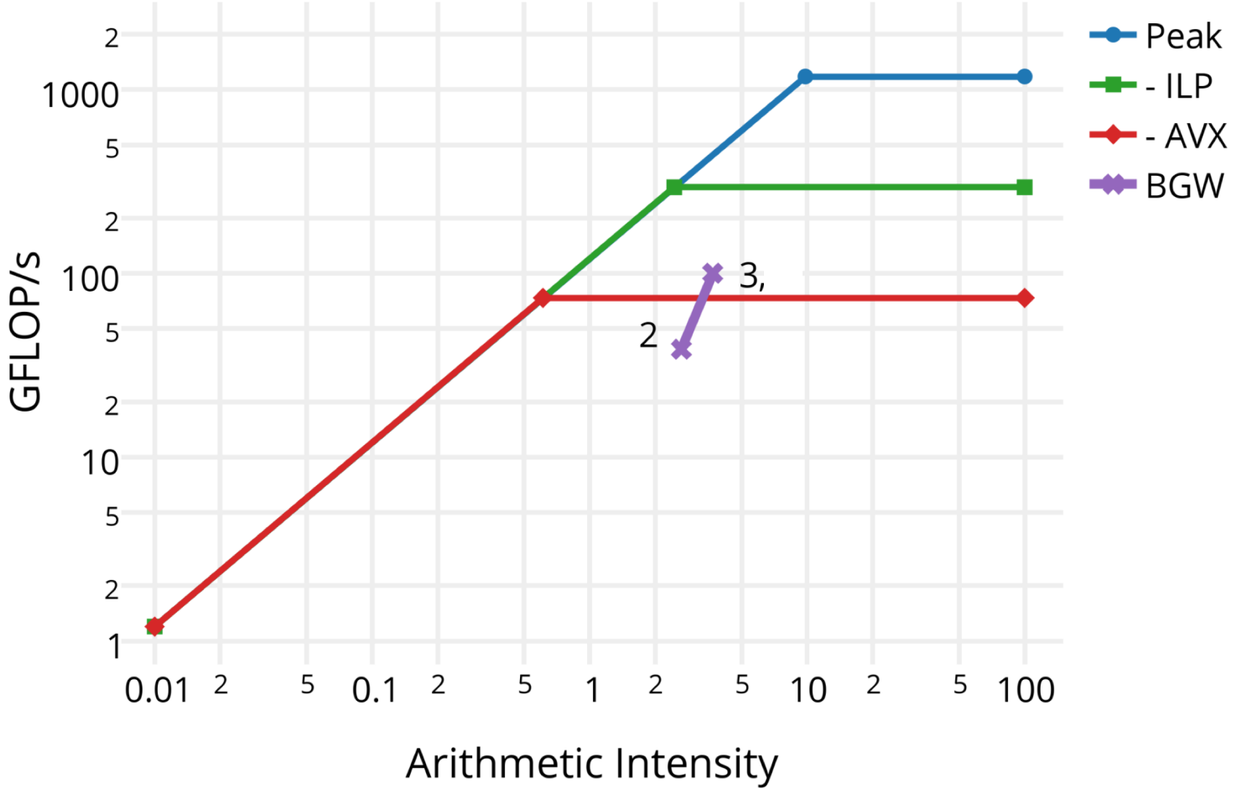
ngpown typically in 100's to 1000s. Good for many threads.

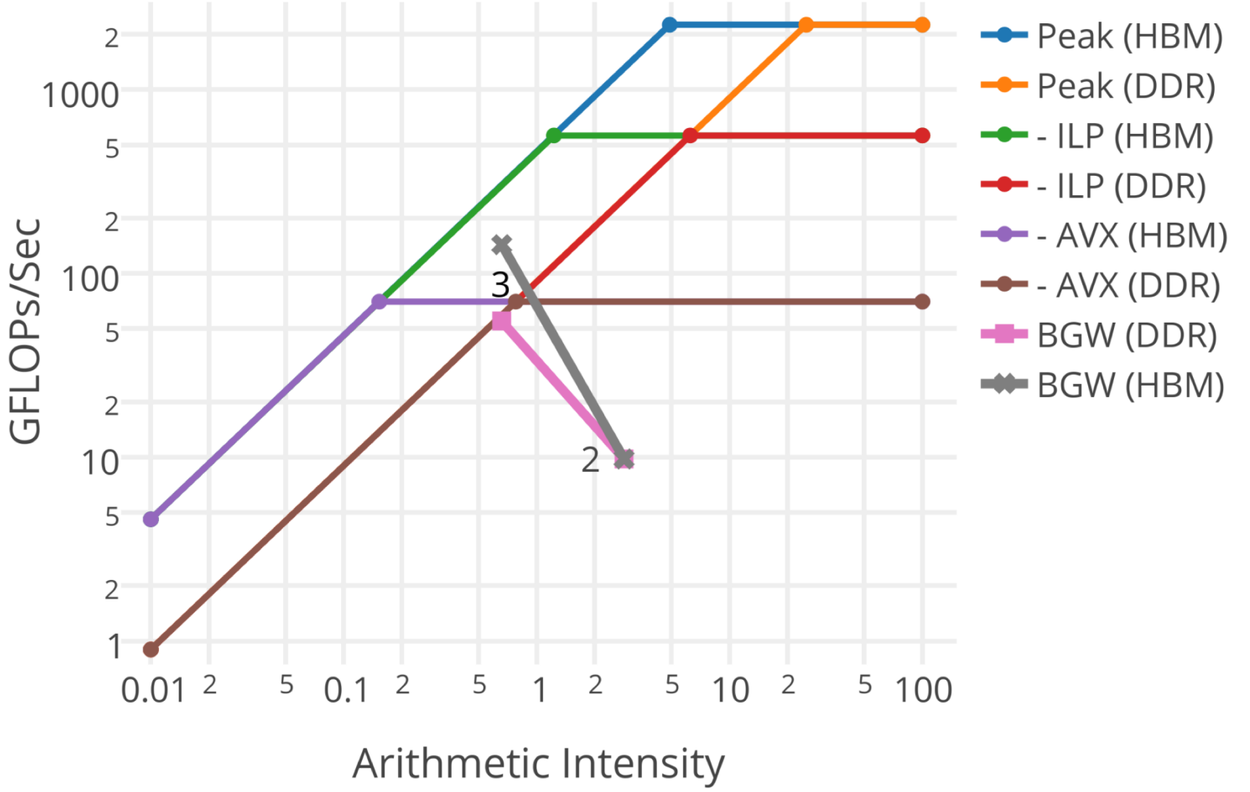
Original inner loop. Too small to vectorize!

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

Attempt to save work breaks vectorization and makes code slower.

Change in Roofline





The loss of L3 on MIC makes locality more important.

Why KNC worse than Haswell for GPP Kernel?

```
!$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 KNL and Haswell. But, Haswell has L3 to catch us.

Why KNC worse than Haswell for GPP Kernel?

```
!$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:

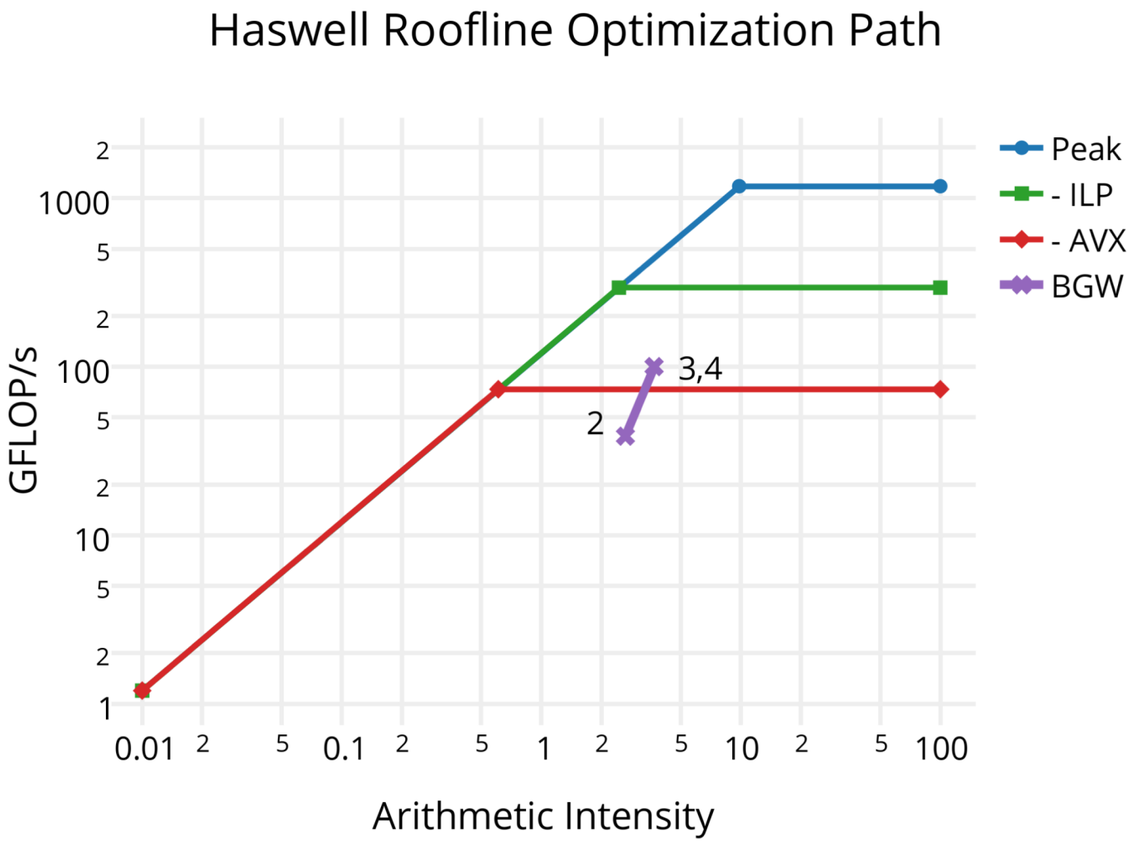
1536 KB

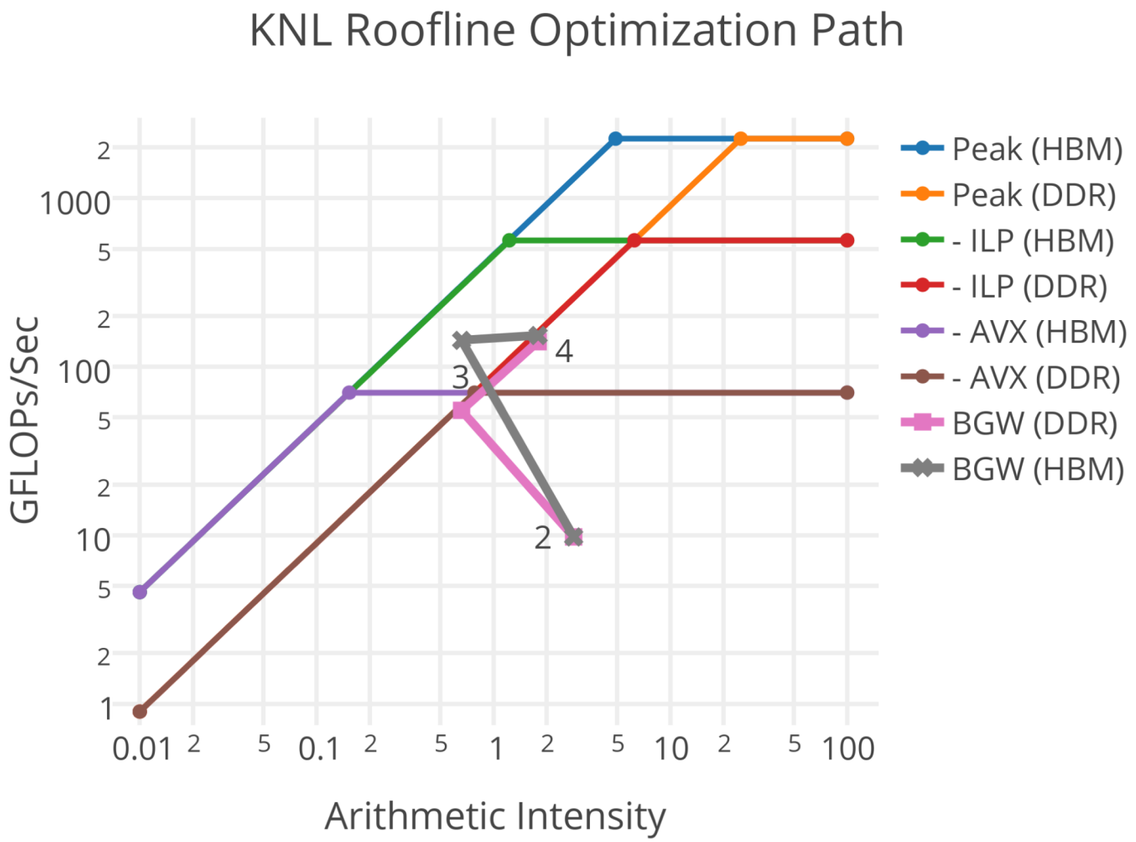
L2 on KNL is 512 KB per core
L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core

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

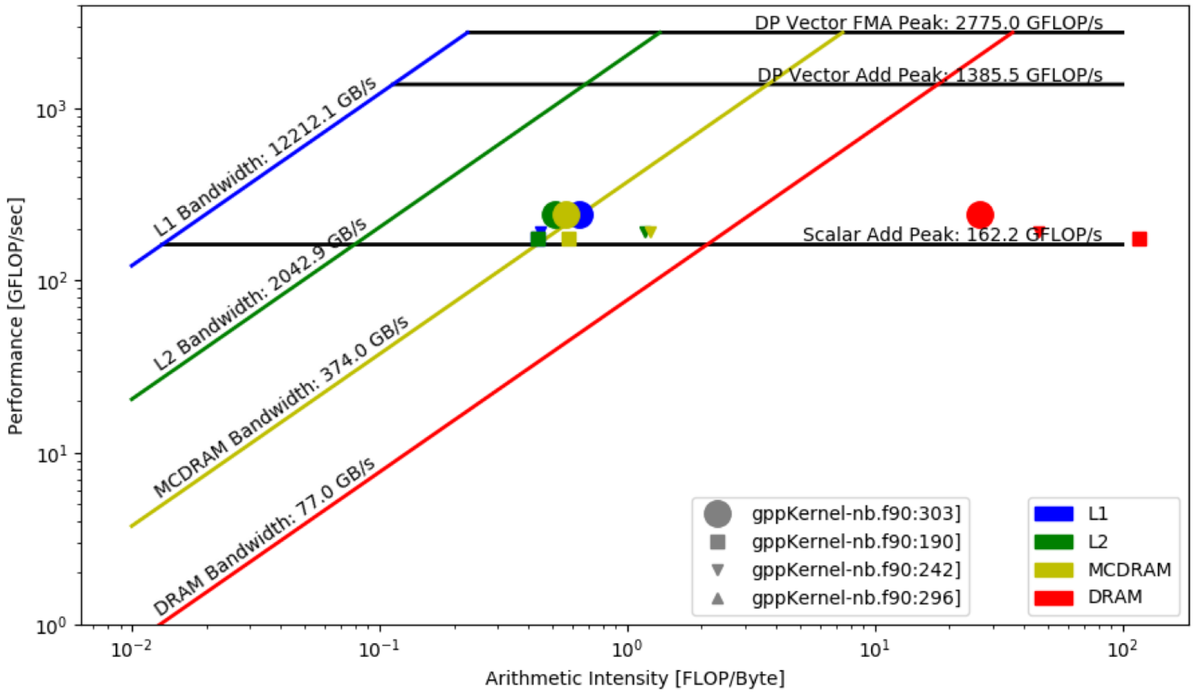
Cache Blocking Optimization



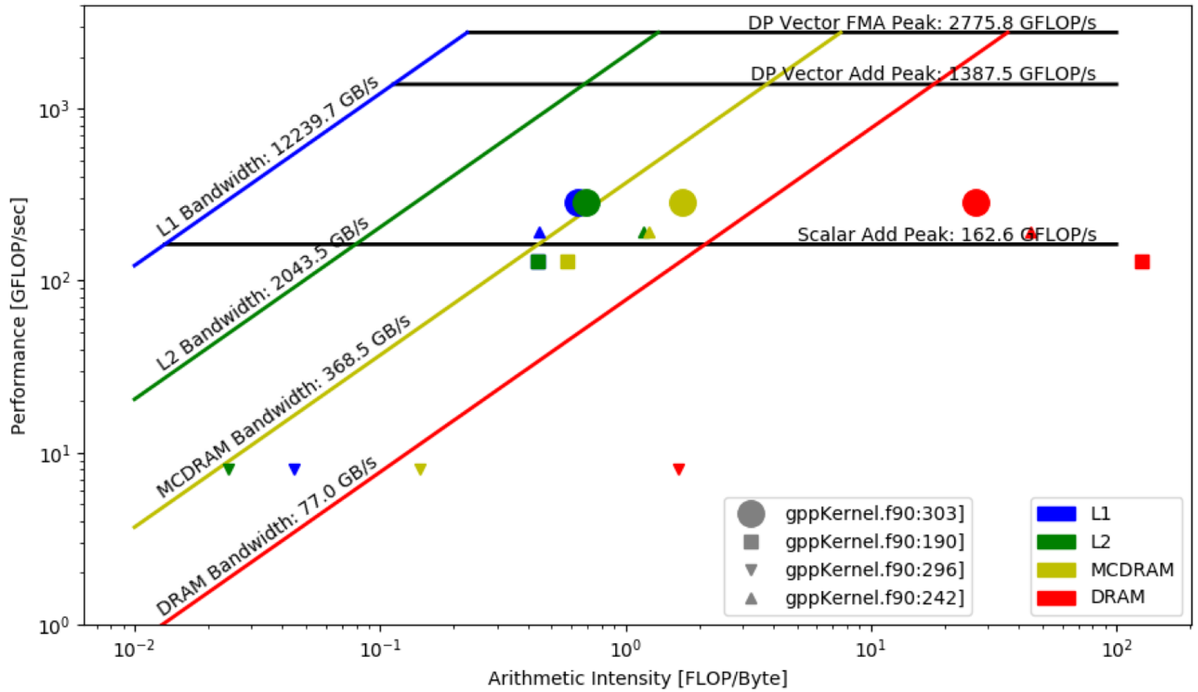


Cache Blocking Optimization (Hierarchical Roofline)

Original Code

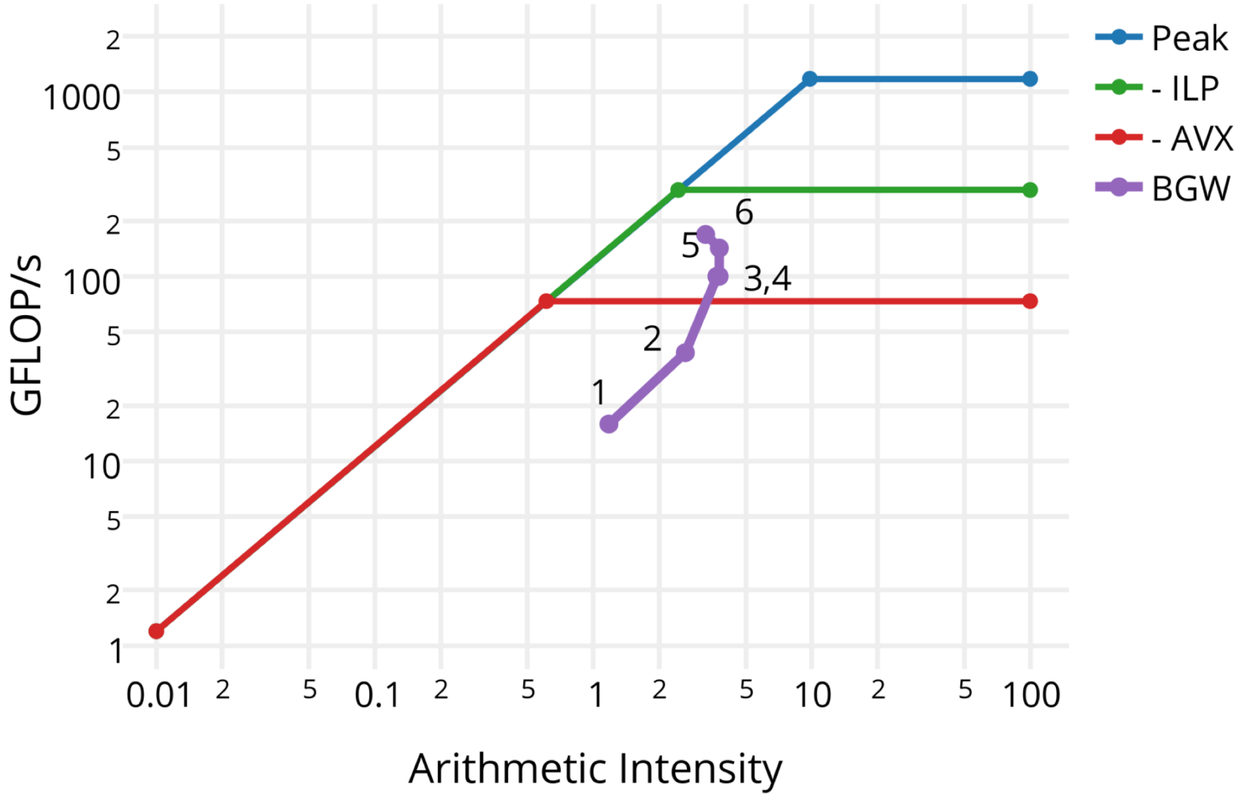


Cache-Blocking Code

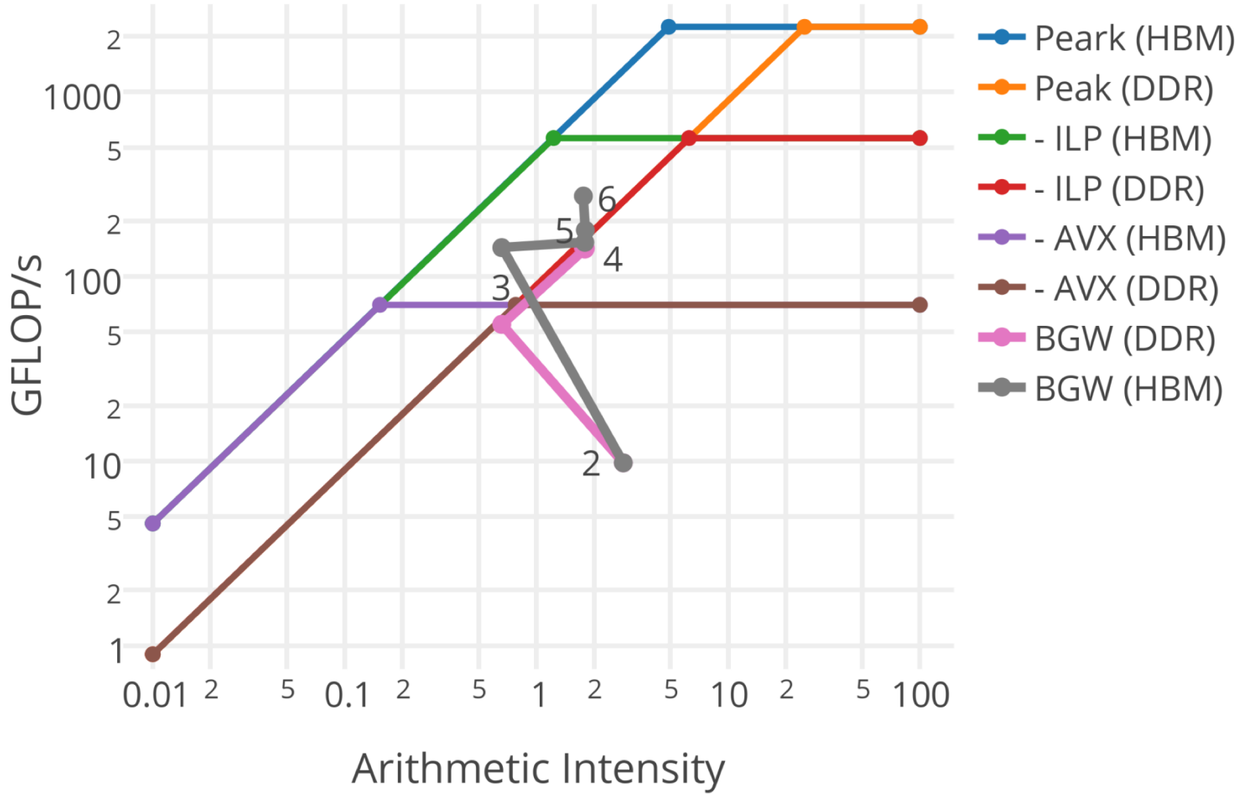


Additional Speedups from Hyperthreading

Haswell Roofline Optimization Path



KNL Roofline Optimization Path

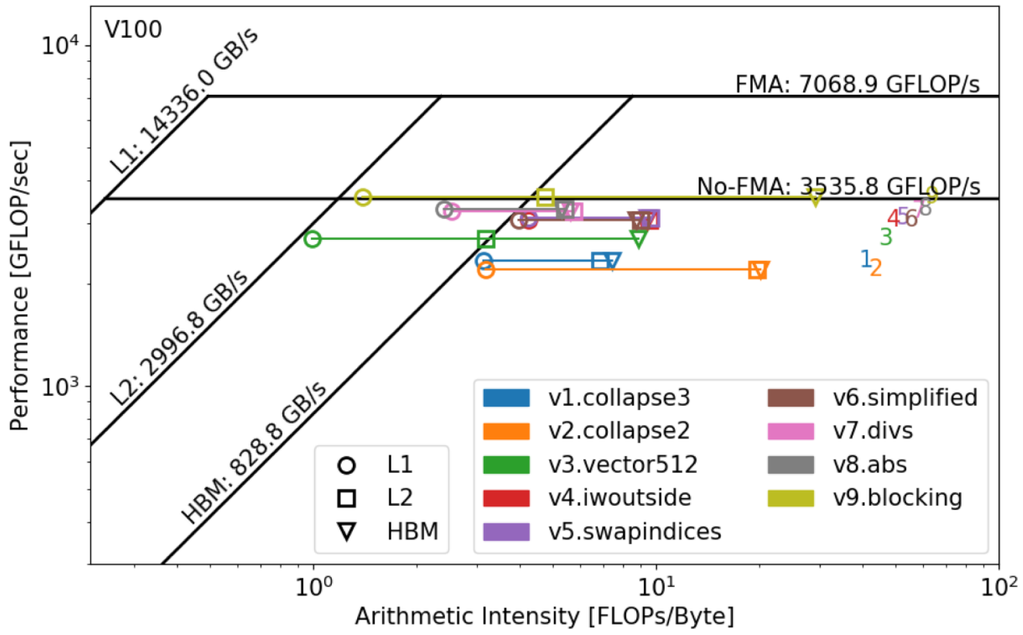


GPP on GPUs in 8 Steps

1. Collapse n', G', and G loops
2. Bring n' loop in; collapse only G' and G
3. Adjust threadblock size
4. Reduce branching; pull iw loop outside
5. Swap indices to suite parallelisation
6. Simplify code
7. Replace div. with rcp. and mul.
8. Replace abs with power of 2
9. Cache blocking

	TFLOPs	Time	TFLOP/s
v1.collapse3	3.71	1.63	2.27
v9.block	2.00	0.57	3.50

3x !!



V1. Naïve Implementation

- Collapse the first 3 loops to gain parallelism

```
!$ACC PARALLEL LOOP COLLAPSE(3) REDUCTION(+: )
do n1 = 1, nbands
   do igp = 1, ngpown
      do ig = 1, ncouls
         do iw = 1, nw      #unrolled
            compute and reduction
```

	TFLOPs	Time (sec)	TFLOP/s
v1.collapse3	3.71	1.63	2.27

V2. More Compute Per Thread

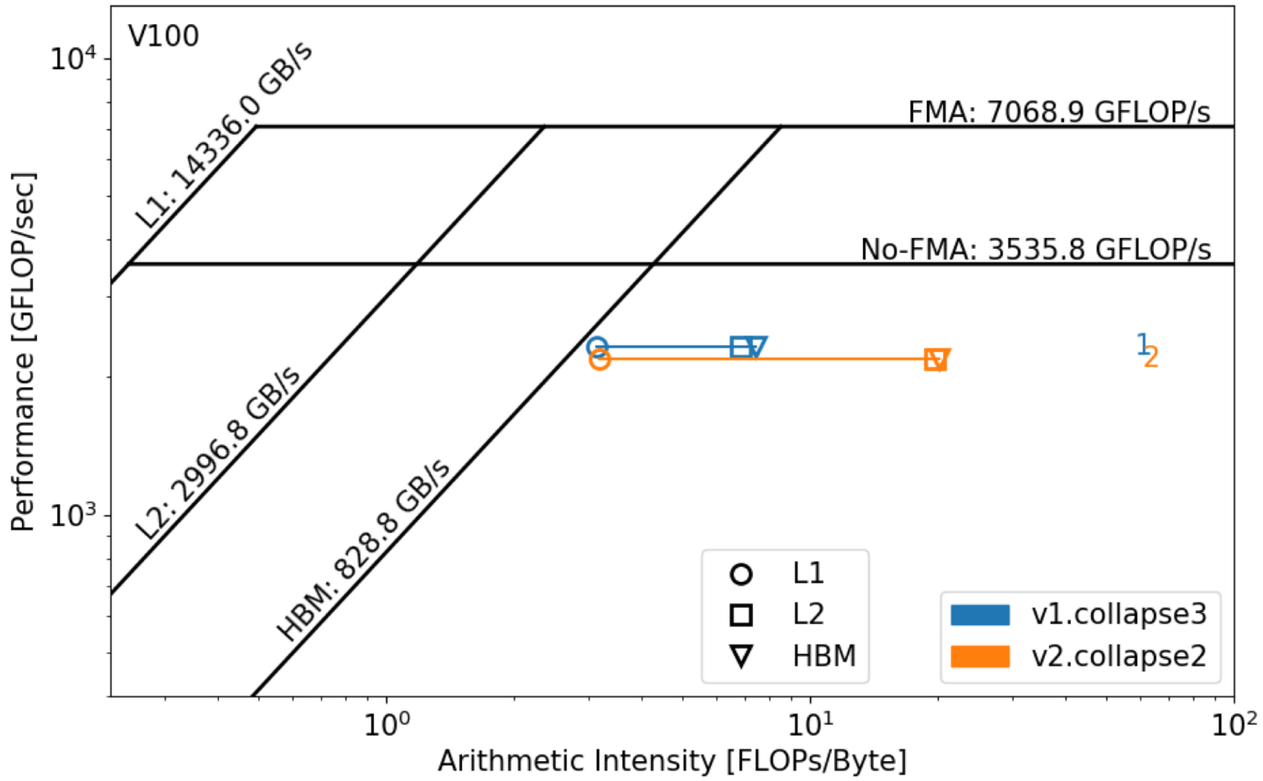
- ~~Move n' loop in, and collapse the first 2 loops~~

```
!$ACC PARALLEL LOOP COLLAPSE(2) REDUCTION(+: )
do igp = 1, ngpown
   do ig = 1, ncouls
      do n1 = 1, nbands      #unrolled too!
         do iw = 1, nw       #unrolled
           compute and reduction
```

	TFLOPs	Time	TFLOP/s
v1.collapse3	3.71	1.63	2.27
v2.collapse2	3.71	1.73	2.15

V2. More Compute Per Thread

- **L2/HBM AI increases!**
- **Register count at 186**
  - **Very low occupancy**
  - **8 warps per SM**
- **Need more warps to hide la**



Active Warps Per Scheduler [warp]	2.00	Instructions Per Active Issue Slot [inst/cycle]	1
Eligible Warps Per Scheduler [warp]	0.35	No Eligible [%]	70.01
Issued Warp Per Scheduler	0.30	One or More Eligible [%]	29.99

V3. Increase Threadblock Size

- Force threadblock size to be 512, instead of the default 128

```
!$ACC PARALLEL LOOP COLLAPSE(2) VECTOR_LENGTH(512) REDUCTION(+: )
```

- Register spills but performance may not be bad!

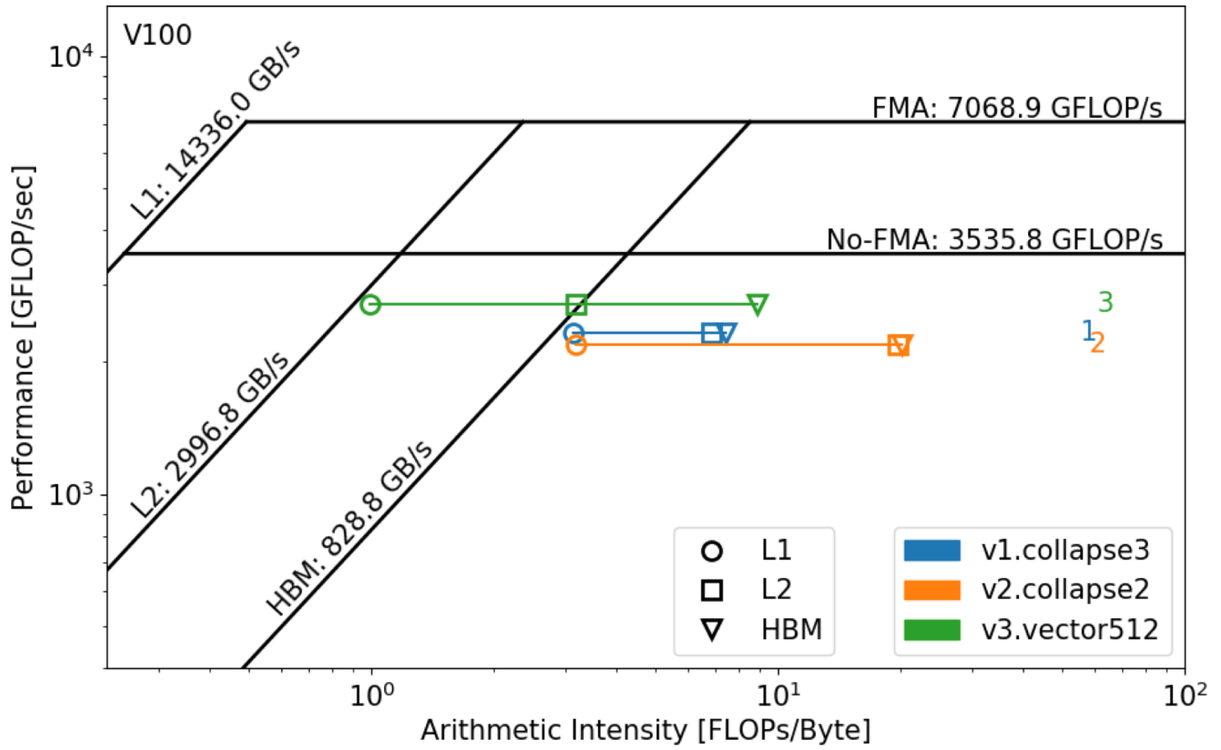
```
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 186 registers, 624 bytes cmem[0], 32 bytes cmem[2]

    104 bytes stack frame, 188 bytes spill stores, 168 bytes spill loads
ptxas info    : Used 128 registers, 624 bytes cmem[0], 32 bytes cmem[2]
```

V3. Increase Threadblock Size

- More bandwidth bound now bu latency hiding is successful!

	TFLOPs	Time	TFLOP/s
v2.collapse2	3.71	1.73	2.15
v3.vector512	3.71	1.40	2.65



Active Warps Per Scheduler [warp]	4.00	Instructions Per Active Issue Slot [inst/cycle]	1
Eligible Warps Per Scheduler [warp]	0.67	No Eligible [%]	58.21
Issued Warp Per Scheduler	0.42	One or More Eligible [%]	41.79

V4. Reduce Branching

- Bring iw loop outside of the kernel

```
do iw = 1, nw                              #reduce branching
!$ACC PARALLEL LOOP COLLAPSE(2) VECTOR_LENGTH(512) REDUCTION(+: )
do igp = 1, ngpown
   do ig = 1, ncouls
      do n1 = 1, nbands     #unrolled
       compute and reduction
```

- Fewer variables to be reduced -> lower register pressure

```
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 122 registers, 600 bytes cmem[0], 32 bytes cmem[2]
```

V4. Reduce Branching

- Aggregated data for all kerne

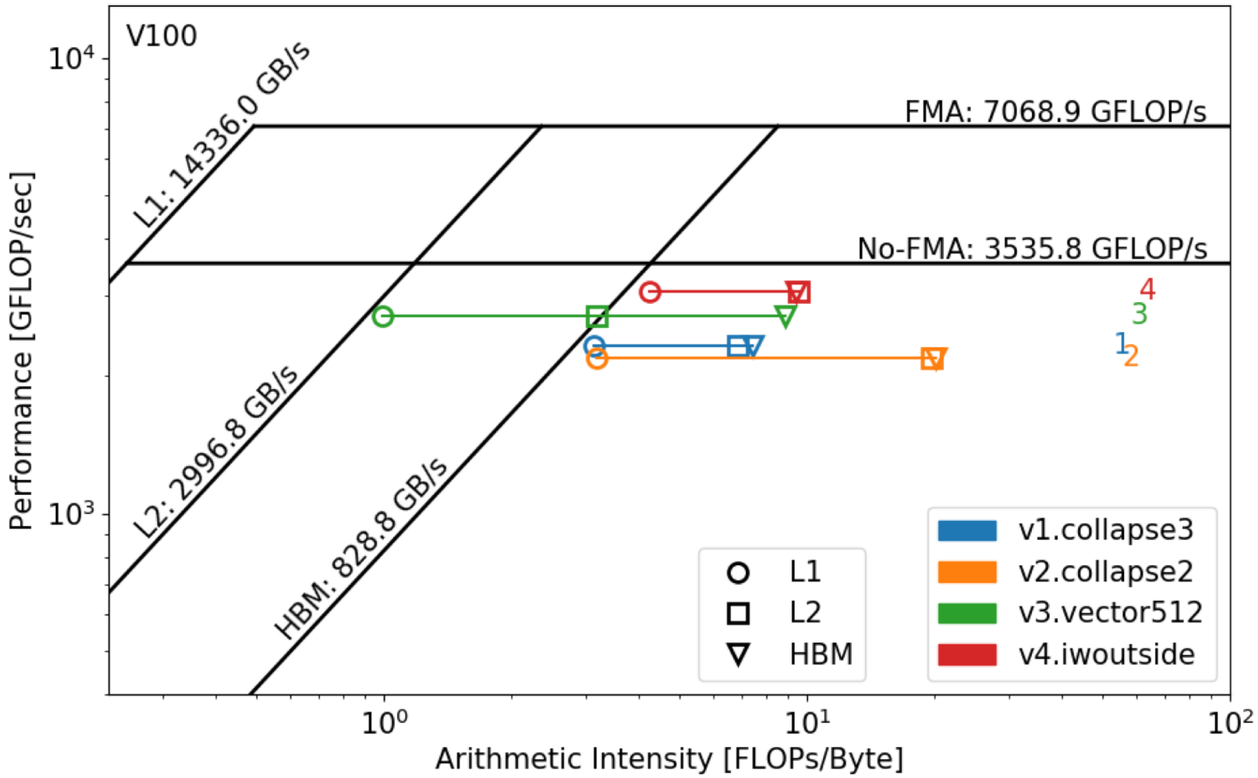
- BRA instruction coun

  14,278,897,053

  5,975,051,812 x 2

16%

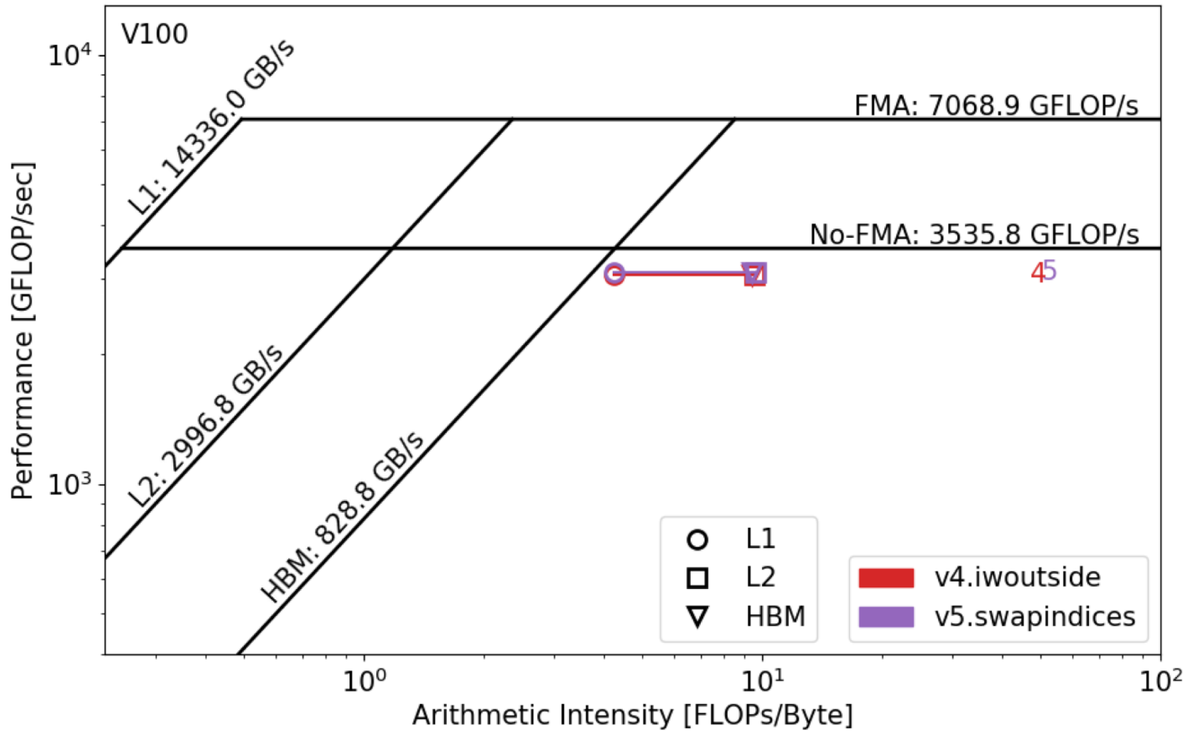
	TFLOPs	Time	TFLOP/s
v3.vector512	3.71	1.40	2.65
v4.iwoutside	3.52	1.17	3.00



V5. Swap Indices

```
do iw = 1, nw
!$ACC PARALLEL LOOP
do igp = 1, ngpown
   do ig = 1, ncouls
      do n1 = 1, nbands
       wx_array(iw,n1) to (n1,iw)
```

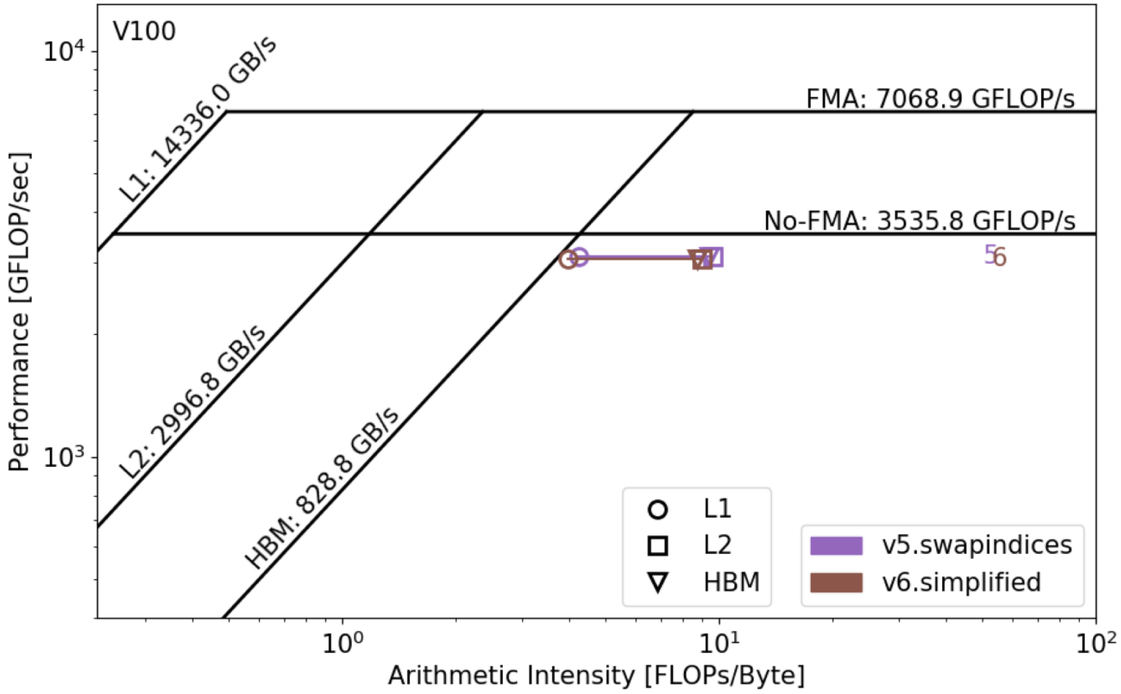
	TFLOPs	Time	TFLOP/s
v4.iwoutside	3.52	1.17	3.00
v5.swapindices	3.52	1.16	3.03



V6. Simplify Code

- Fewer instructions -> less work
  - Pull repeated instructions outside the loop
  - Use temporary variables to hold intermediate values for reuse
- Less branches -> better programmin
  - 3 branches is more than 1 branc
    worse than 2 branches!

	TFLOPs	Time	TFLOP/s
v5.swapindices	3.52	1.16	3.03
v6.simplify	3.30	1.10	3.00

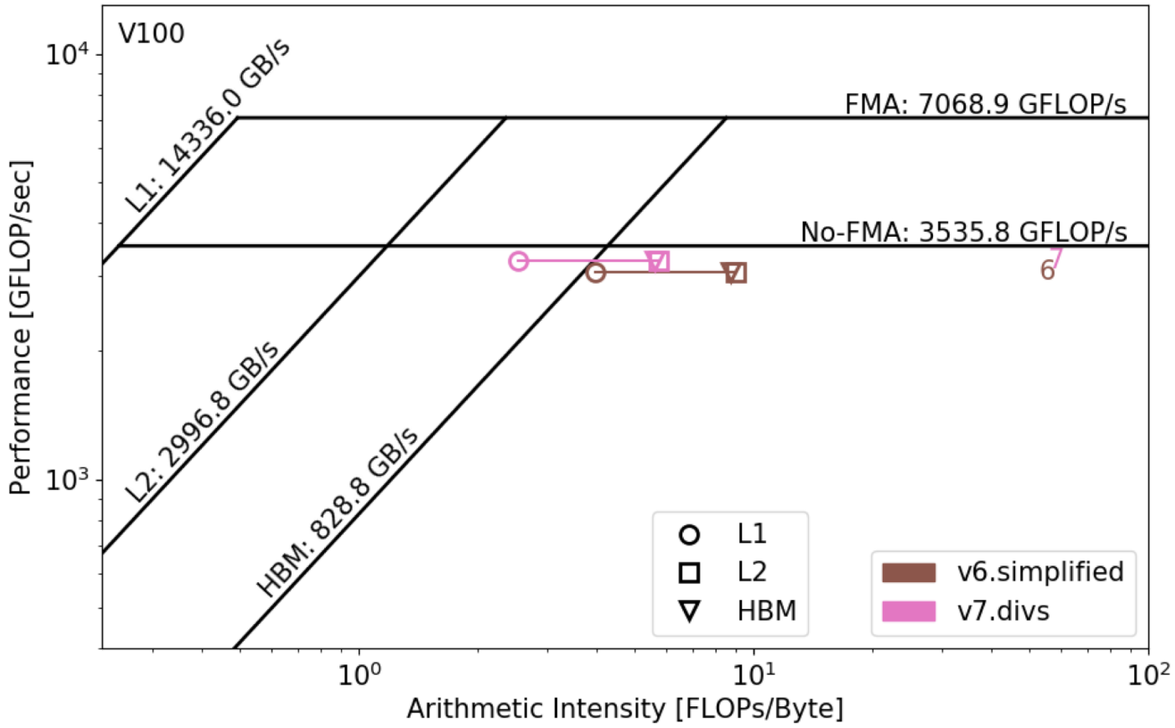


V7. Replace Divides

- Replace (complex) div. with (double) rcp. and (complex) mul.
- Lower instruction count: 40%

- More bandwidth bound now!

	TFLOPs	Time	TFLOP/s
v6.simplify	3.30	1.10	3.00
v7.divs	2.09	0.66	3.18

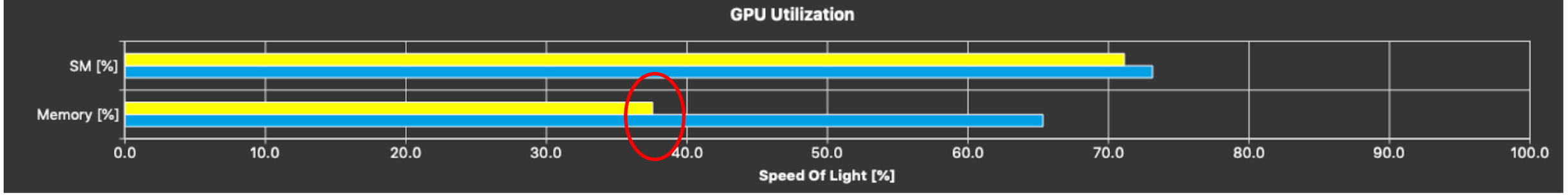


V7. Replace Divides

- Can be confirmed by Nsight Compute profiles

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.

SOL SM [%]	73.13	(+2.82%)	Duration [msecond]	350.19	(-43.07%)
SOL Memory [%]	65.35	(+73.98%)	Elapsed Cycles [cycle]	459634385	(-43.08%)
SOL TEX [%]	19.81	(+57.58%)	SM Active Cycles [cycle]	458943655.99	(-43.08%)
SOL L2 [%]	22.93	(+73.60%)	SM Frequency [cycle/nsecond]	1.31	(-0.02%)
SOL FB [%]	65.35	(+73.98%)	Memory Frequency [cycle/usecond]	877.45	(-0.01%)



V8. Replace abs(x) with x**2

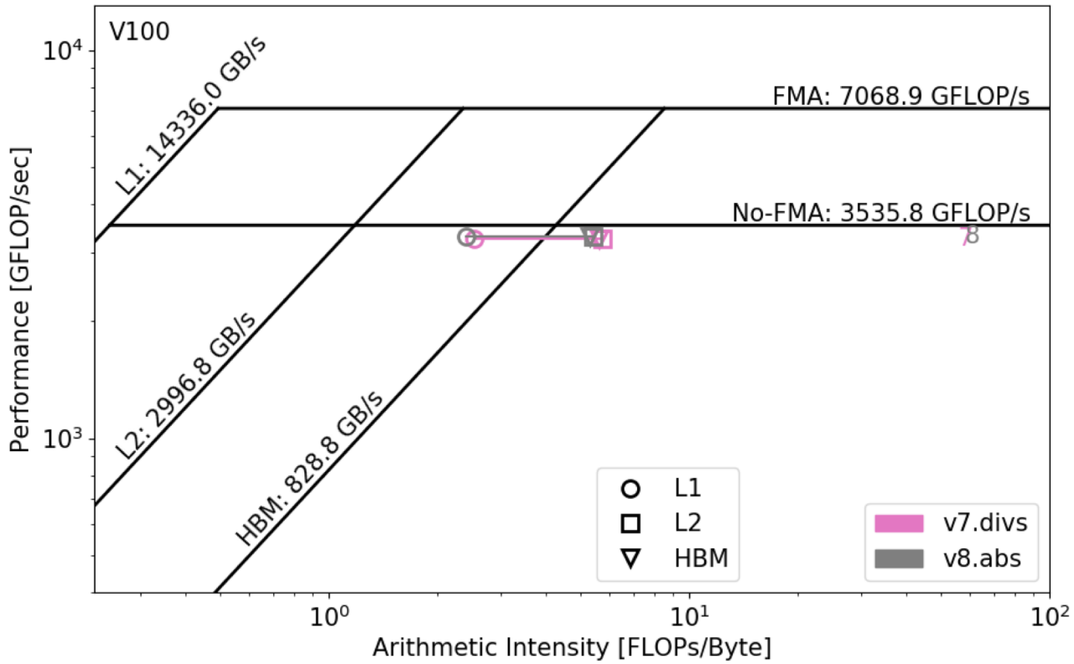
```
complex(DP) ssx
if (abs(ssx) .le. ssxcutoff) then
```

→

```
real(DP) ssxpower
if (ssxpower .le. ssxcutoff **2) then
```

- $\sqrt{x^2 + y^2} < z$ → $x^2 + y^2 < z^2$
- sqrt(complex) vs power of 2
- Causing pipeline to wait

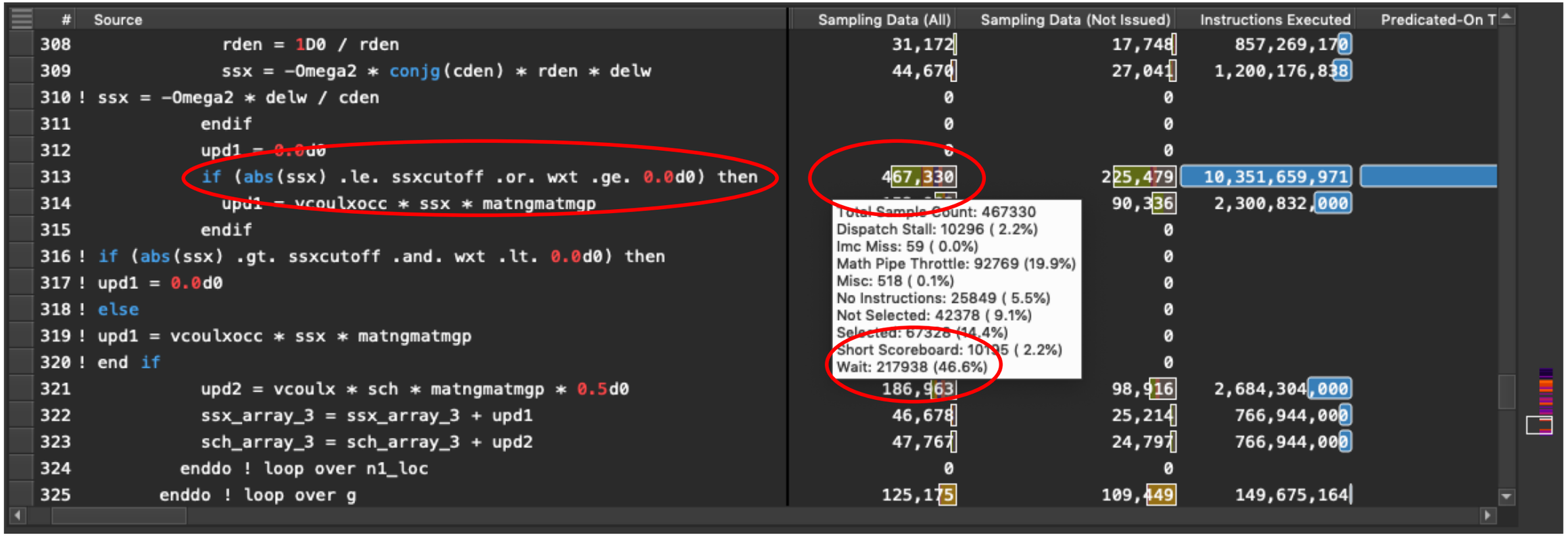
	TFLOPs	Time	TFLOP/s
v7.divs	2.09	0.66	3.18
v8.abs	1.99	0.62	3.23



V8. Replace abs(x) with x**2

Before:

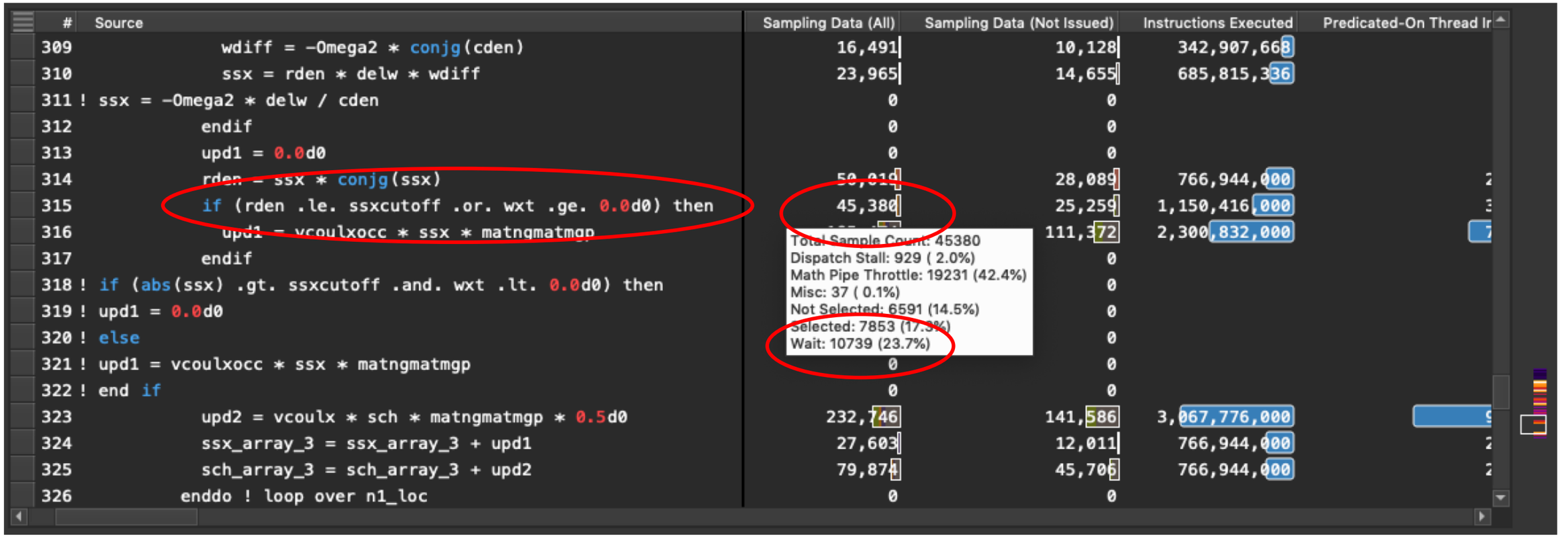
- **Wait:** warp stalled waiting on a fixed latency execution dependency



V8. Replace abs(x) with x**2

After:

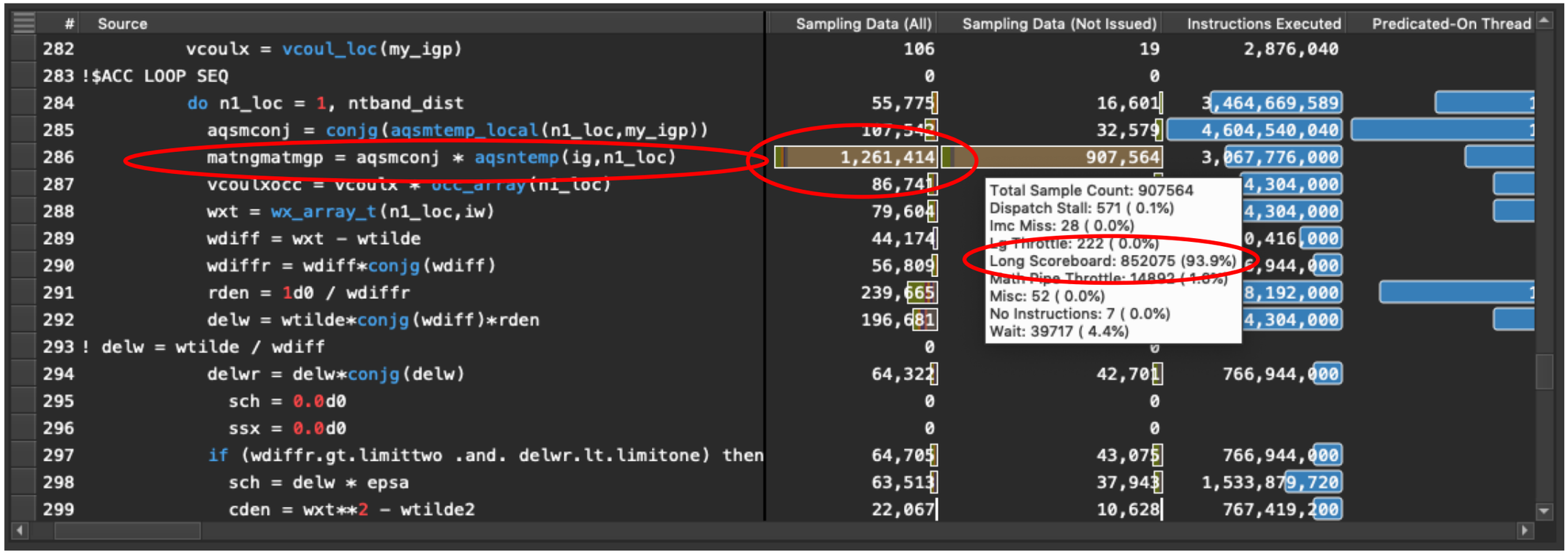
- Wait: 46.6% -> 23.7%



#	Source	Sampling Data (All)	Sampling Data (Not Issued)	Instructions Executed	Predicated-On Thread Ir
309	wdiff = -Omega2 * conjg(cden)	16,491	10,128	342,907,668	
310	ssx = rden * delw * wdiff	23,965	14,655	685,815,336	
311	! ssx = -Omega2 * delw / cden	0	0		
312	endif	0	0		
313	upd1 = 0.0d0	0	0		
314	rden = ssx * conjg(ssx)	50,019	28,089	766,944,000	2
315	if (rden .le. ssxcutoff .or. wxt .ge. 0.0d0) then	45,380	25,259	1,150,416,000	3
316	upd1 = vcoulxocc * ssx * matngmatmgp		111,372	2,300,832,000	7
317	endif		0		
318	! if (abs(ssx) .gt. ssxcutoff .and. wxt .lt. 0.0d0) then		0		
319	! upd1 = 0.0d0		0		
320	! else		0		
321	! upd1 = vcoulxocc * ssx * matngmatmgp	0	0		
322	! end if	0	0		
323	upd2 = vcoulx * sch * matngmatmgp * 0.5d0	232,746	141,586	3,067,776,000	9
324	ssx_array_3 = ssx_array_3 + upd1	27,603	12,011	766,944,000	2
325	sch_array_3 = sch_array_3 + upd2	79,874	45,706	766,944,000	2
326	enddo ! loop over n1_loc	0	0		

V9. Cache Blocking

- **Non-coalesced memory access for `aqsntemp`**
- **Causing Long Scoreboard Warp State**
  - Warp stalled waiting for L1TEX (local, global, surface, tex) memory operation



#	Source	Sampling Data (All)	Sampling Data (Not Issued)	Instructions Executed	Predicated-On Thread
282	vcoulx = vcoul_loc(my_igp)	106	19	2,876,040	
283	!$ACC LOOP SEQ	0	0		
284	do n1_loc = 1, ntband_dist	55,775	16,601	3,464,669,589	
285	aqsmconj = conjg(aqsmtemp_local(n1_loc,my_igp))	107,543	32,579	4,604,540,040	
286	matngmatmgp = aqsmconj * aqsntemp(ig,n1_loc)	1,261,414	907,564	3,067,776,000	
287	vcoulxocc = vcoulx * occ_array(n1_loc)	86,741		4,304,000	
288	wxt = wx_array_t(n1_loc,iw)	79,604		4,304,000	
289	wdiff = wxt - wtilde	44,174		0,416,000	
290	wdiffr = wdiff*conjg(wdiff)	56,809		2,944,000	
291	rden = 1d0 / wdiffr	239,665		8,192,000	
292	delw = wtilde*conjg(wdiff)*rden	196,681		4,304,000	
293	! delw = wtilde / wdiff	0	0		
294	delwr = delw*conjg(delw)	64,322	42,701	766,944,000	
295	sch = 0.0d0	0	0		
296	ssx = 0.0d0	0	0		
297	if (wdiffr.gt.limittwo .and. delwr.lt.limitone) then	64,705	43,075	766,944,000	
298	sch = delw * epsa	63,513	37,943	1,533,879,720	
299	cden = wxt**2 - wtilde2	22,067	10,628	767,419,200	

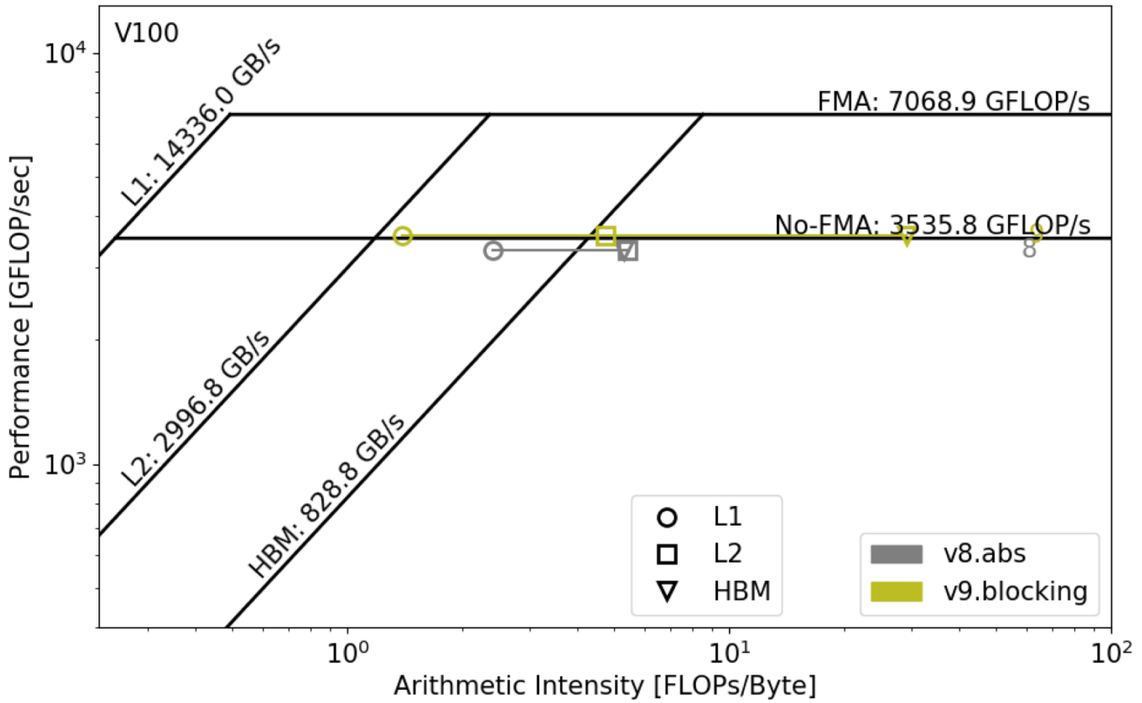
Total Sample Count: 907564
Dispatch Stall: 571 (0.1%)
Imc Miss: 28 (0.0%)
Lg Throttle: 222 (0.0%)
Long Scoreboard: 852075 (93.9%)
Math Pipe Throttle: 14892 (1.6%)
Misc: 52 (0.0%)
No Instructions: 7 (0.0%)
Wait: 39717 (4.4%)

V9. Cache Blocking

- Break loops into chunks and reuse data across threadblocks
- Increase L2 hit rate

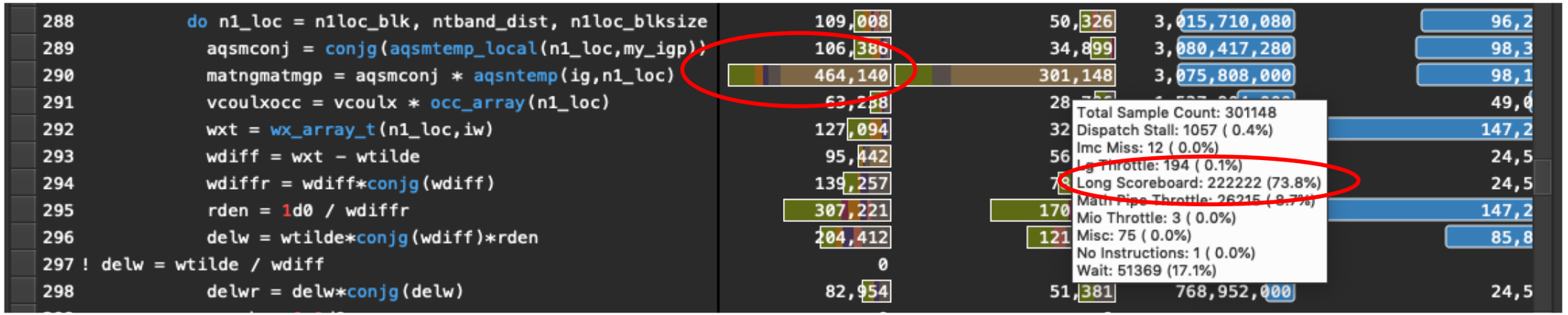
```
!$ACC LOOP GANG VECTOR
do ig_blk = 1, ig_blksize
!$ACC LOOP SEQ
  do ig = ig_blk, ncouls, ig_blksize
```

	TFLOPs	Time	TFLOP/s
v8.abs	1.99	0.62	3.23
v9.block	2.00	0.57	3.50



V9. Cache Blocking

- **Less Long Scoreboard samples and higher L2/L1 hit rate**



Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy).

Memory Throughput [Gbyte/second]	131.85 (−78.99%)	Mem Busy [%]	31.10 (+29.96%)
L1 Hit Rate [%]	71.38 (+27.27%)	Max Bandwidth [%]	28.44 (−59.32%)
L2 Hit Rate [%]	88.89 (+3272.88%)	Mem Pipes Busy [%]	12.84 (+59.44%)

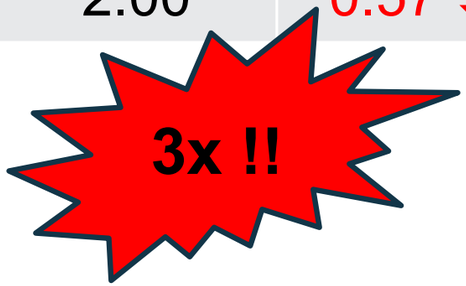
Summary

8 Steps to Optimize Sigma-GPP

1. Collapse n', G', and G loops
2. Bring n' loop in; collapse only G' and G
3. Adjust threadblock size
4. Reduce branching; pull iw loop outside
5. Swap indices to suite parallelisation
6. Simplify code
7. Replace div. with rcp. and mul.
8. Replace abs with power of 2
9. Cache blocking

```
!$ACC PARALLEL LOOP REDUCTION(+:
)
do n1 = 1, nbands
   do igp = 1, ngpown
      do ig = 1, ncouls
         do iw = 1, nw
         compute and reduction
```

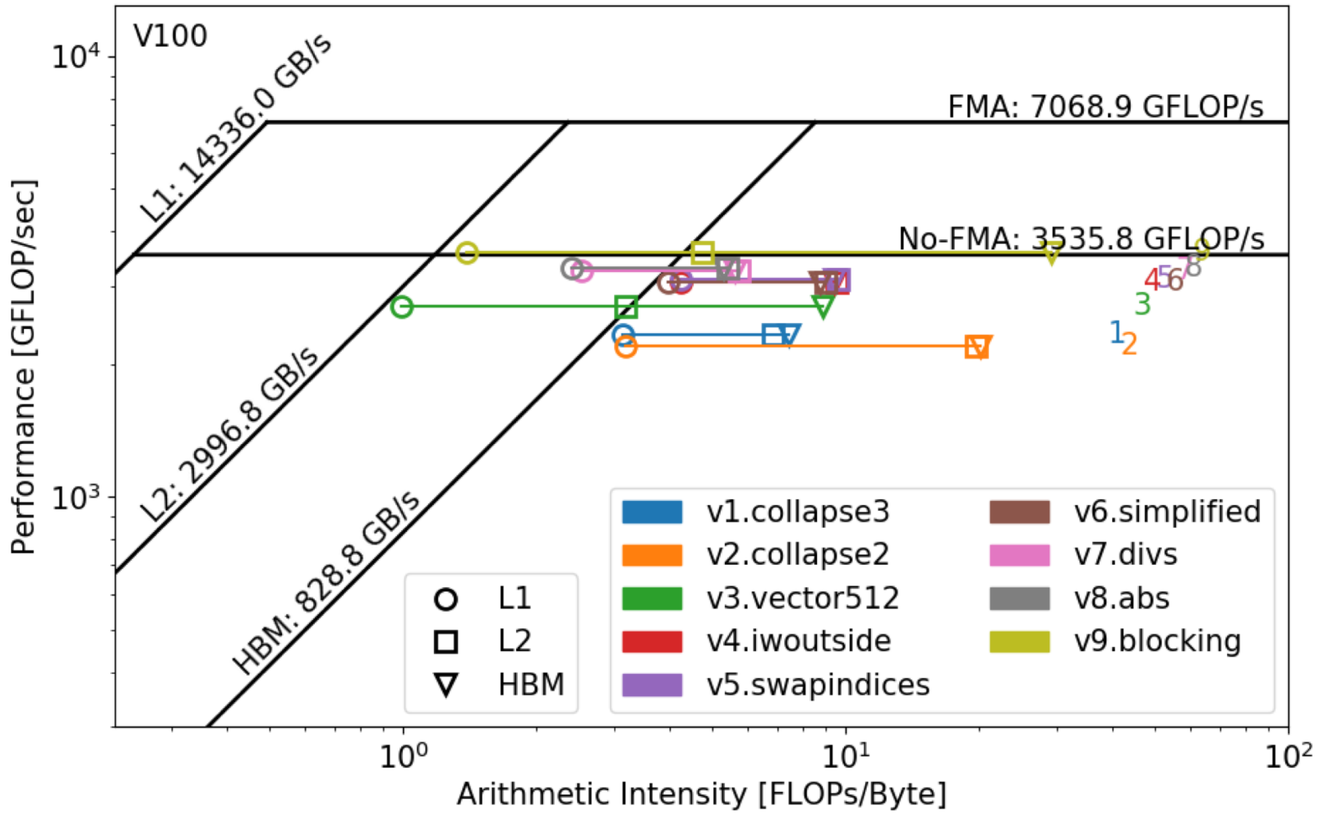
	TFLOPs	Time	TFLOP/s
v1.collapse3	3.71	1.63	2.27
v9.block	2.00	0.57	3.50



Conclusion

- Code is still bandwidth and latencv bound
  - shared memory
  - lower register count
  - improve FMA ratio

- **Together with profilers, Roofline provides the complete solution for your performance analysis and optimization needs!**



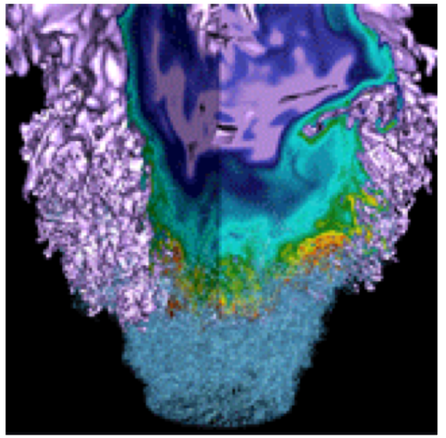
Conclusions

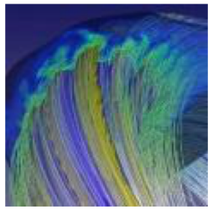
Roofline is a great way to Frame Conversation with Application Teams

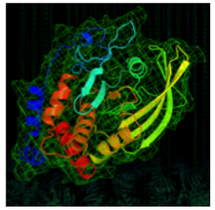
Helps Motivate, Direct and Visualize the optimization process.

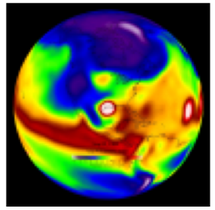
Coming soon to a tool near you!!

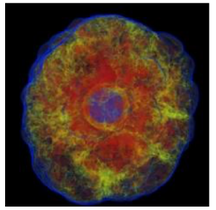
Extras

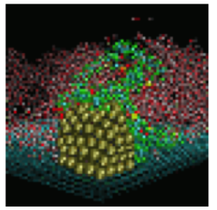












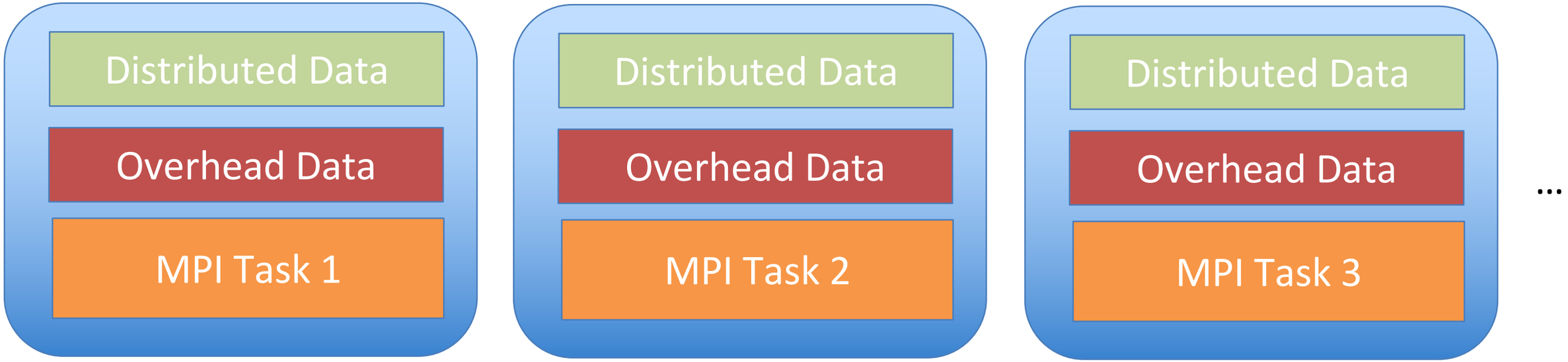
NERSC 40 YEARS at the FOREFRONT

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**.

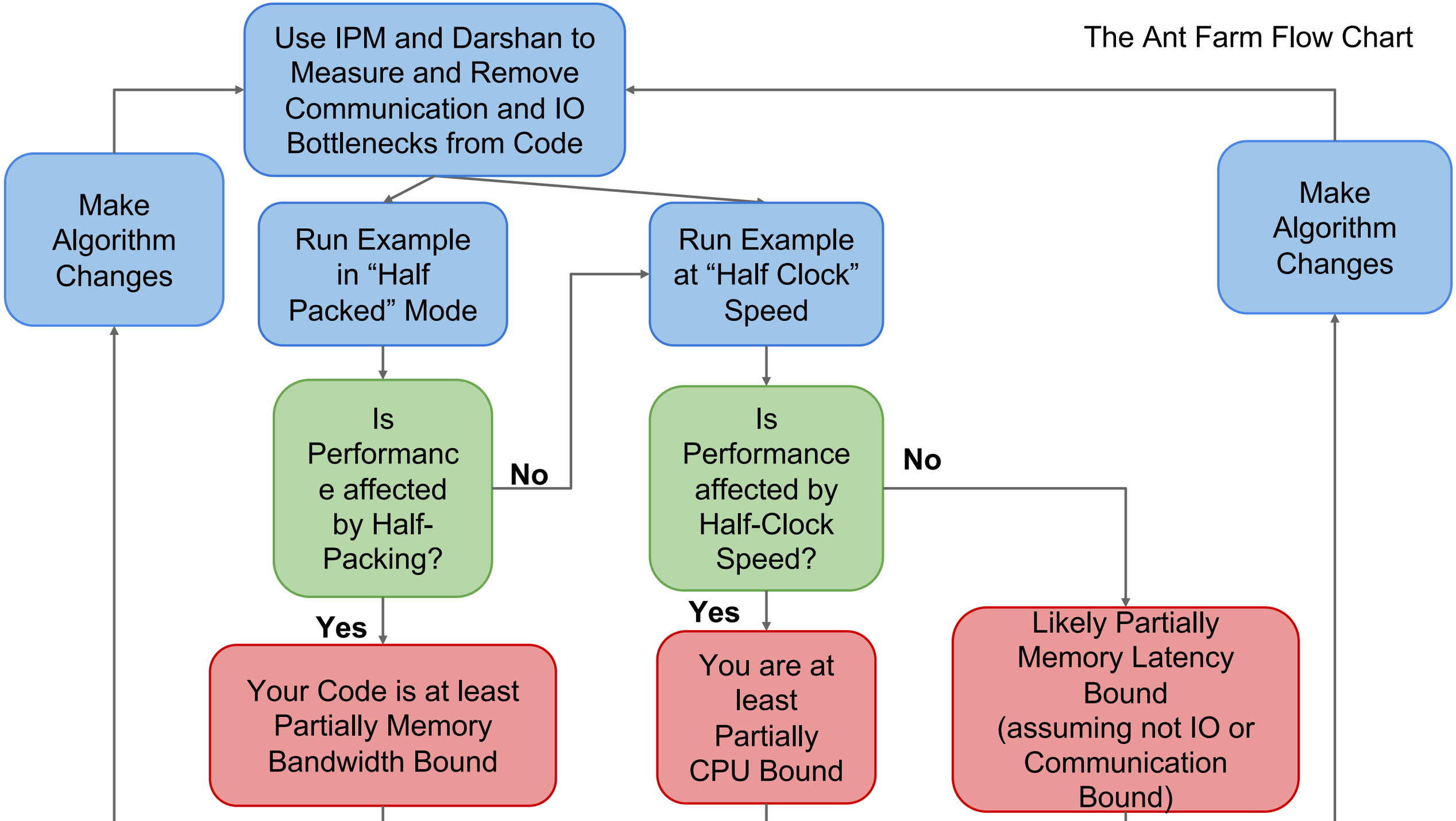


Computational Bottlenecks

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.

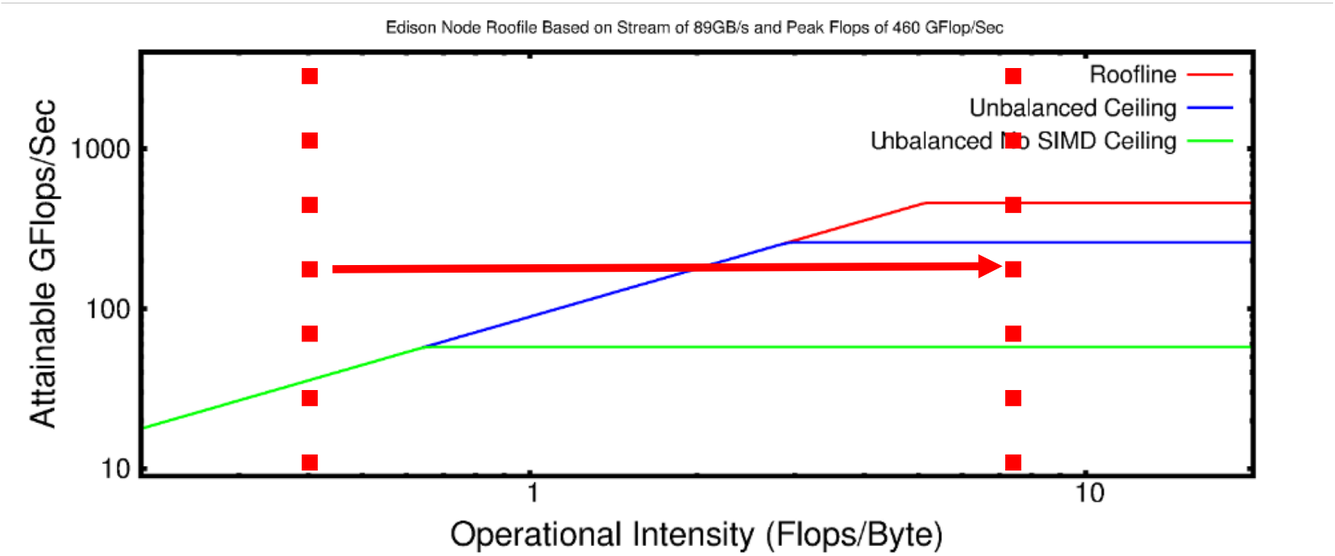
$$\langle n\mathbf{k}|\,\Sigma_{\mathrm{CH}}(E)\,|n'\mathbf{k}\rangle = \frac{1}{2}\sum_{n''}\sum_{\mathbf{q}\mathbf{G}\mathbf{G}'} M^{*}_{n''n}(\mathbf{k},-\mathbf{q},-\mathbf{G})M_{n''n'}(\mathbf{k},-\mathbf{q},-\mathbf{G}') \times \frac{\Omega^{2}_{\mathbf{G}\mathbf{G}'}(\mathbf{q})\,(1-i\tan\phi_{\mathbf{G}\mathbf{G}'}(\mathbf{q}))}{\tilde{\omega}_{\mathbf{G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


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 Performanc e affected by Half-Packing?
No
Yes
Is Performance affected by Half-Clock Speed?
No
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)

So, you are Memory Bandwidth Bound?

What to do?

1. Try to improve memory locality, cache reuse



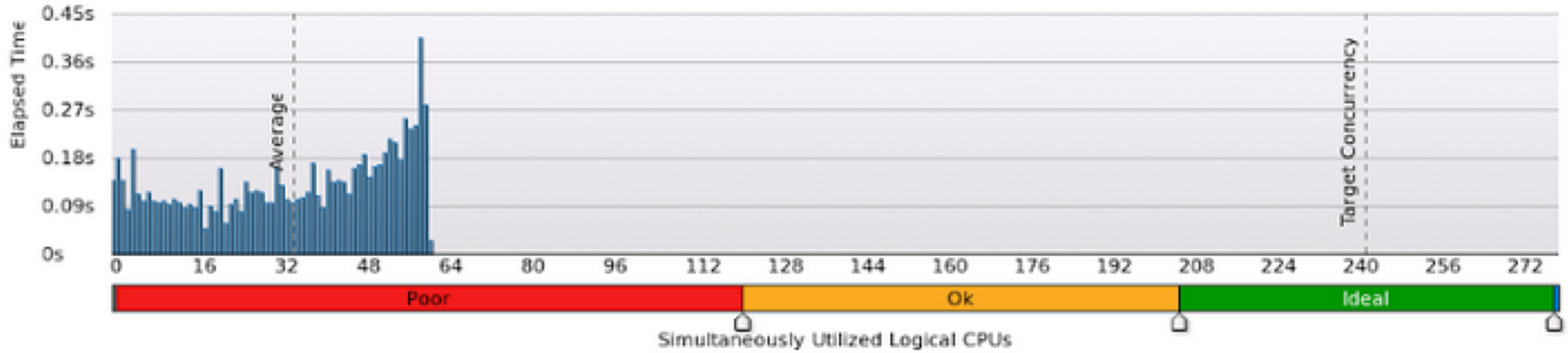
1. Identify the key arrays leading to high memory bandwidth usage and make sure they are/will-be allocated in HBM on Knights Landing.

   Profit by getting ~ 4-5x 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.



1. 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

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 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 Knights-Landing, each core will support up to 4 threads. Use them all.

Are you memory or compute bound? Or both?

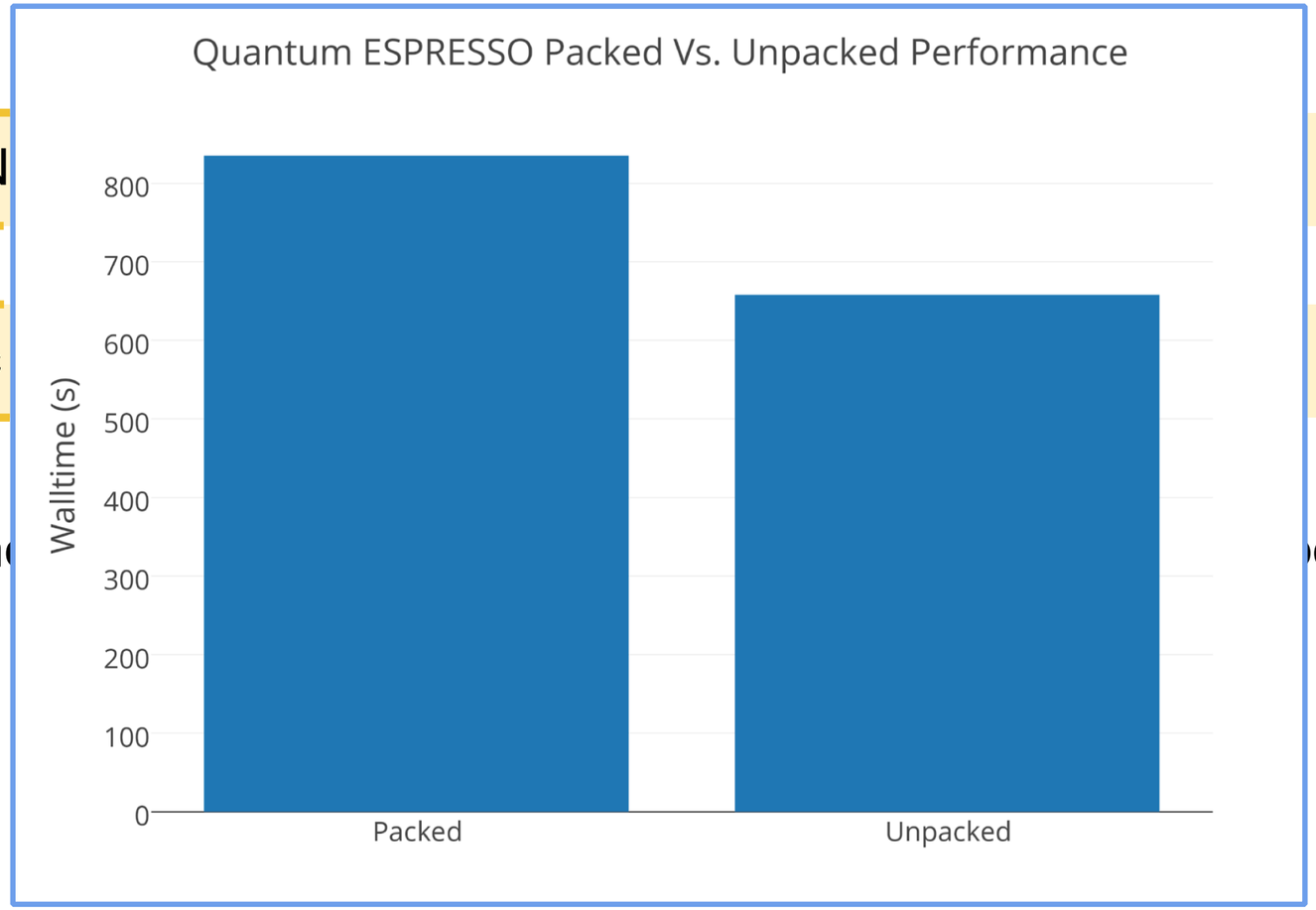
Run Example in “Half Packed” Mode

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 …

srun -N 2 -n 24 -c …

If your performan… …ound



Cori KNL System

Cray XC40 system with 9,600+ Intel Knights Landing (KNL) nodes:

- 68 cores, 272 Hardware Threads
- Up to 32 FLOPs per Cycle, 1.2-1.4 GHz Clock Rate
- Wide (512 Bit) vector Units
- Multiple Memory Tiers: 96 GB DRAM / 16 GB HBM
- NVRAM Burst Buffer 1.5 PB, 1.5 TB/sec