LA-UR-18-27941
Approved for public release; distribution is unlimited.

Title: Improving Capsaicin Sweep Performance Using Memory Prefetching

Author(s): Halvic, Ian William

Intended for: Presentation to CCS-2

Issued: 2018-08-20

Improving Capsaicin Sweep Performance Using Memory Prefetching

Ian Halvic

8/16/18

Texas A&M University

Mentor: Kent Budge

NNSA
National Nuclear Security Administration

Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

Agenda

8/16/2018

- Prefetching Overview
- Target Routine
- Target Architecture
- Testing Method
- Haswell Results
- KNL Results
- Wrap-Up

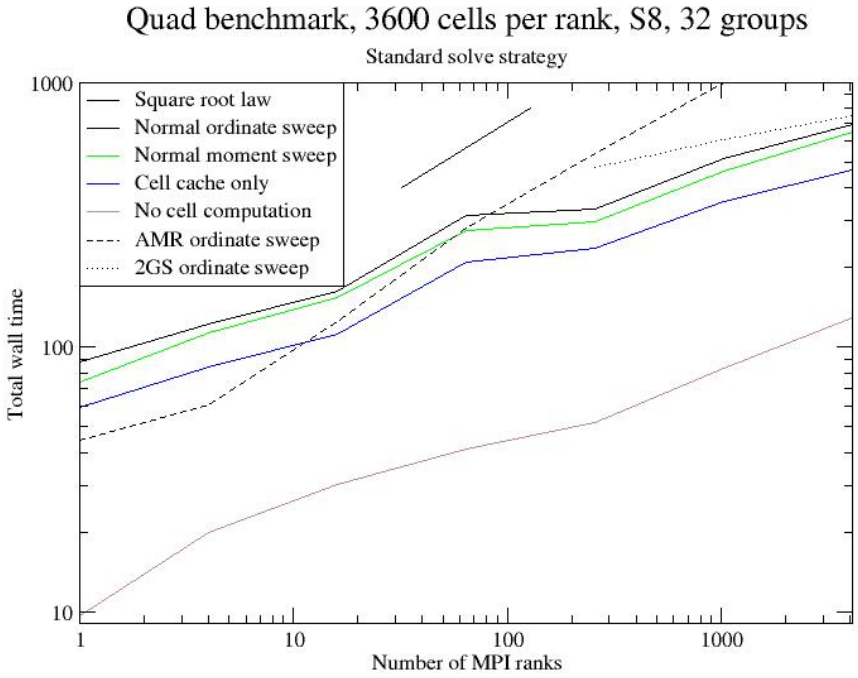
Prefetching

- **Software Prefetching offers a straightforward method to improve runtime**
  - Fetch data from memory to cache before use (reduce latency)
  - Better utilize memory bandwidth
  - Take advantage of compute heavy sections to have memory working
  - Minimal restructuring of code (no logic changes)

- **Limitations**
  - Prefetch location in code
    - Too late: Wasted overhead
    - Too early: Eviction, Cache pollution
  - Memory bandwidth
  - Cache size
  - Overhead
  - Limited number of fetch “slots”
  - Have to keep hardware (and compiler) prefetching in mind

Target

- **Attempt to apply technique to Capsaicin Sn solver**
  - Using slightly modified code with fixed number of sweeps and no source update (Rocotillo)
  - Target is the lumped DFEM Quad grid sweep
  - Targeted variables:
    - scattering source
    - σ values
    - incident flux



Difference between blue (Cache only) and pink (No cell computation) shows room for memory improvement.

Example Prefetch Call

```
#pragma omp simd
  for (unsigned e = 0; e < emax; e+=4)
      {
        __builtin_prefetch(&Source[e+(0*offset)],0,3);
        __builtin_prefetch(&Source[e+(1*offset)],0,3);
        __builtin_prefetch(&Source[e+(2*offset)],0,3);
        __builtin_prefetch(&Source[e+(3*offset)],0,3);

        __builtin_prefetch (&sigma[e+(0*offset)],0,3);
        __builtin_prefetch (&sigma[e+(1*offset)],0,3);
        __builtin_prefetch (&sigma[e+(2*offset)],0,3);
        __builtin_prefetch (&sigma[e+(3*offset)],0,3);
      }
```

Target Architectures

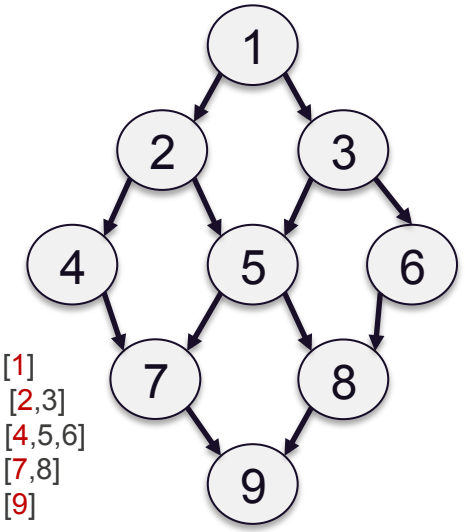
- **Tested on Trinitite system (2 architectures)**
- **Intel Haswell E5-2698-v3 processors**
  - 16 cores per socket, 2 sockets per node
  - 40MB L3 per socket (shared).
  - Prefetching into L3 cache
- **Intel Knights Landing co-processors (KNL)**
  - 68 cores per node
  - 34MB L2 (Shared 1MB per 2 cores)
  - High Bandwidth Memory
    - High bandwidth, similar to cache
    - DRAM regime latency
  - Prefetch into L2 cache
  - Lower clock compered to Haswell

Testing Method

- **Test problem scales mesh with number of ranks used**
  - 32 groups, N=8
  - Each rank gets an equal number of cells
    - Small problem – 6000 cells per rank
    - Large problem – 89600 cells per rank
      - Memory allocation issues with large problems using this scaling mesh
- **1 set := 5 Baseline & 5 PF runs. Averages used to compute % runtime change**
- **Intel V-tune used to determine cache misses. Typically only run for 1 rank**
- **Scaling of successful prefetch schemes with respect to MPI rank tested out to 4 full nodes**

Prefetching for next cell (Haswell)

- **Pull data for next cell while computing current cell**
  - Extra overhead to store a link to the next cell
  - Relies on a queue of “computable” cells being populated
  - Can take advantage of current cell matrix-solve compute time
- **Showed promise for single rank**
  - Miss reduction from ~2.2B to 0.5B
  - 5% runtime improvement
- **Poor results for parallel runs**
  - More frequently emptying computable cell stack
  - Potential Bandwidth issues
  - Runtime degradation



1st: [1]
2nd: [2,3]
3rd: [4,5,6]
4th: [7,8]
5th: [9]

Compute stack refreshes for above graph, scheme cannot prefetch for the first element of refresh (shown in red).

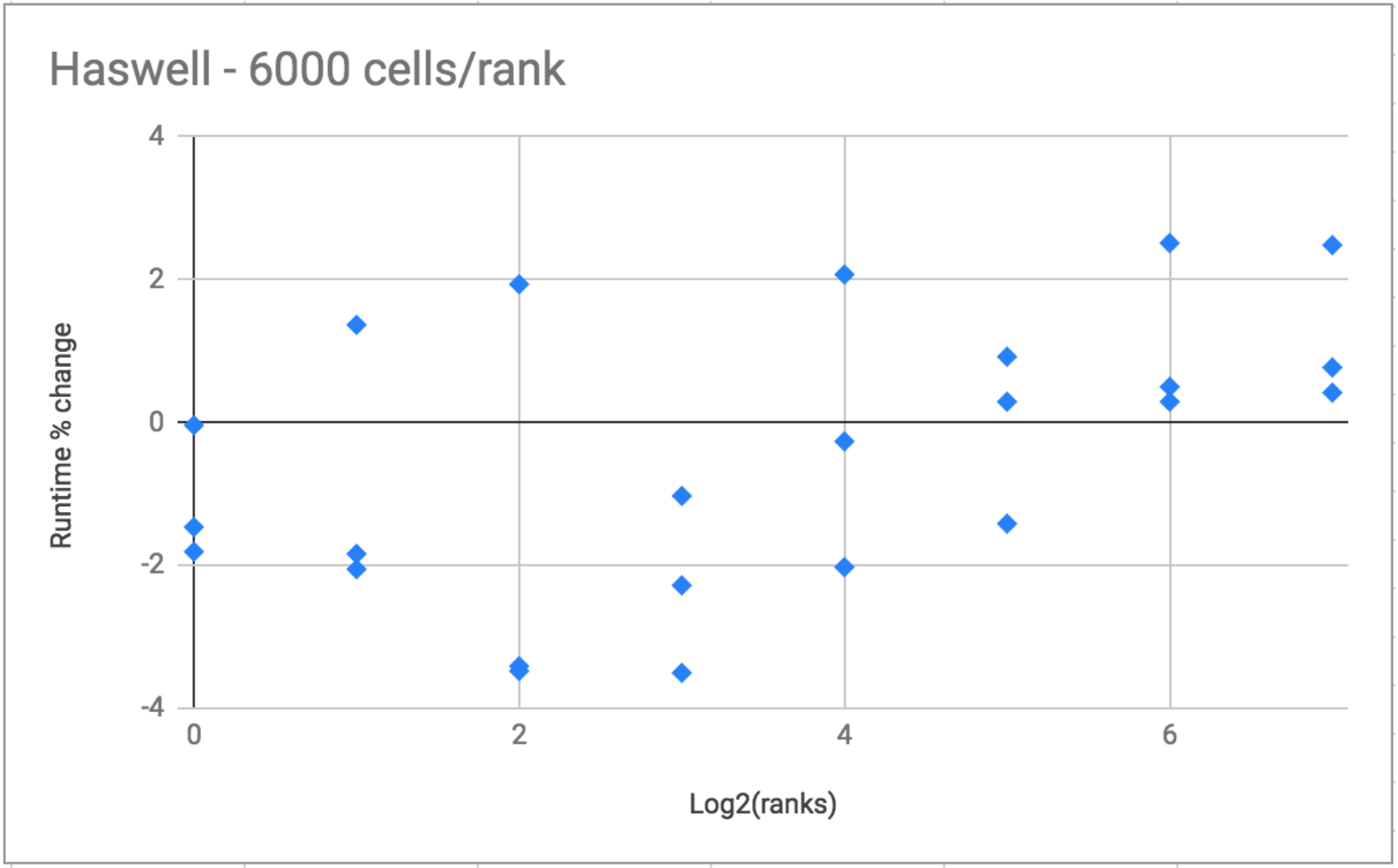
This method has issues with parallel scaling, graph dependence.

Prefetching within cell (Haswell)

- **Focus primarily on improving performance within a single cell**
- **Numerous configurations tested**
  - Many which reduce misses but offer no runtime improvement
- **Best performing scheme (1 rank)**
  - Miss reduction from ~2.2B to 0.5B (similar to next cell PF method)
  - Runtime improvement of ~3%
- **Expanding to parallel reduces runtime impact**
  - Issues set in very quickly, ~8 to 16 ranks
  - Runtimes became more varied with higher ranks, difficult to analyze

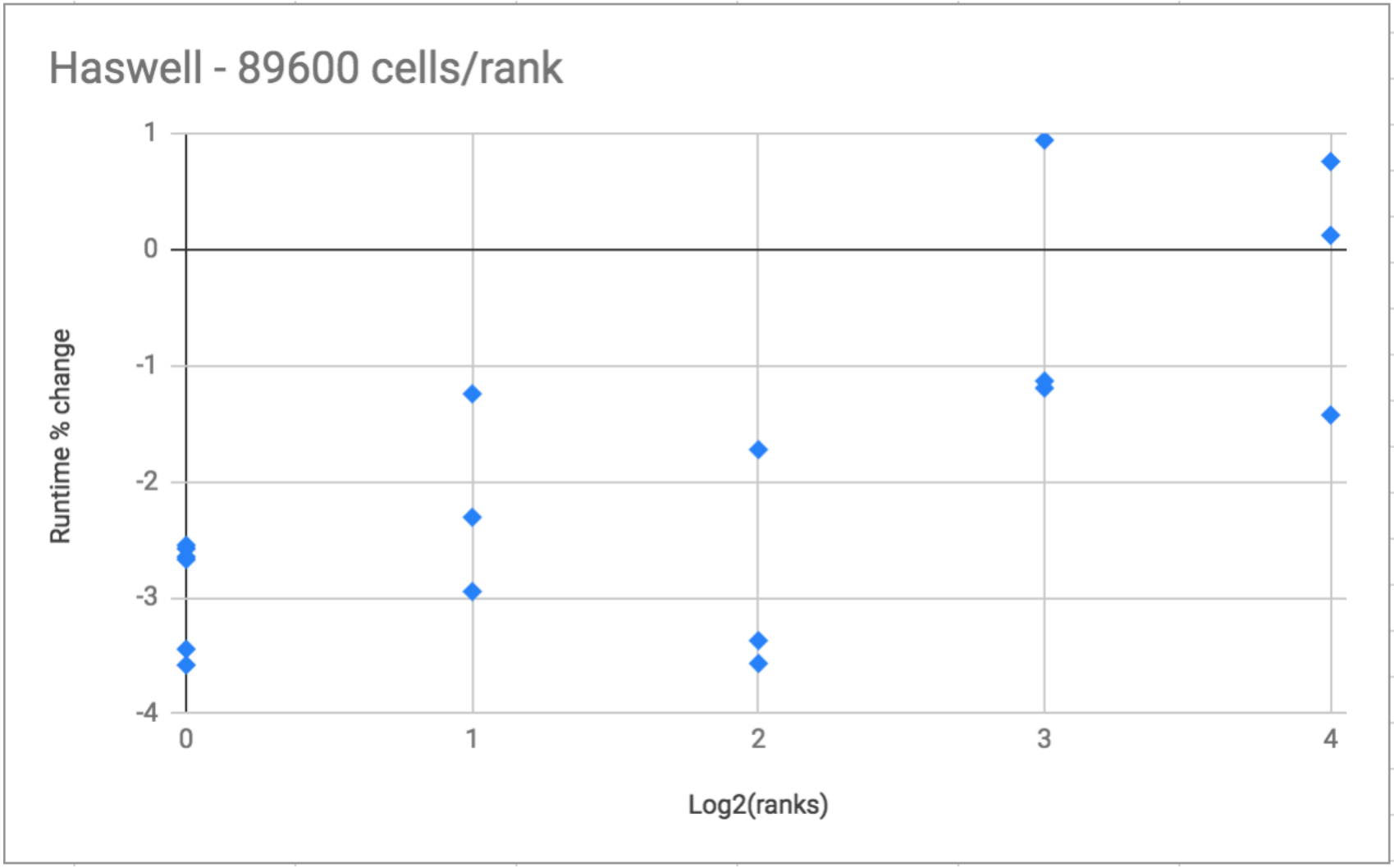
Haswell: Runtime Change Scaling

Small problem tested from 1 to 128 ranks (4 nodes)

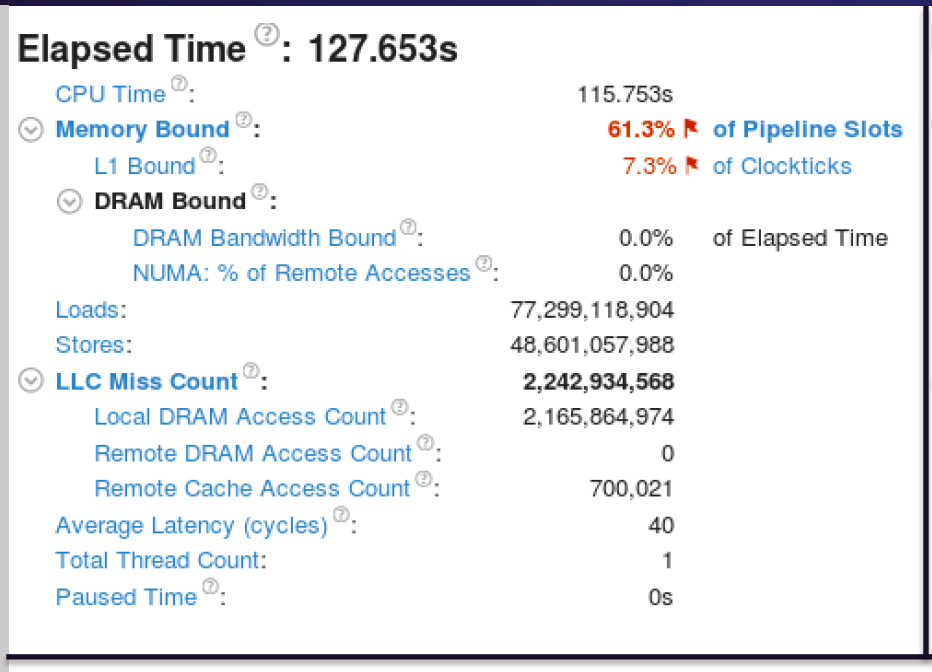


Haswell: Runtime Change Scaling

Larger problem tested from 1 to 16 ranks

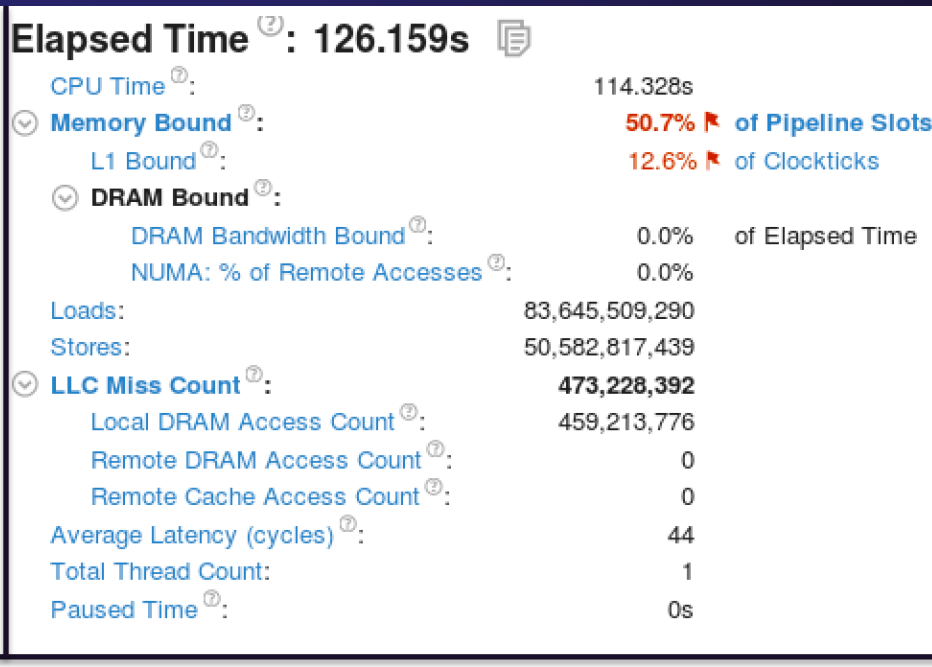


Haswell: V-tune and Runtime Comparison



Elapsed Time: 127.653s

CPU Time:	115.753s	
Memory Bound:	**61.3%**	**of Pipeline Slots**
L1 Bound:	7.3%	of Clockticks
DRAM Bound:		
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
NUMA: % of Remote Accesses:	0.0%	
Loads:	77,299,118,904	
Stores:	48,601,057,988	
LLC Miss Count:	**2,242,934,568**	
Local DRAM Access Count:	2,165,864,974	
Remote DRAM Access Count:	0	
Remote Cache Access Count:	700,021	
Average Latency (cycles):	40	
Total Thread Count:	1	
Paused Time:	0s	



Elapsed Time: 126.159s

CPU Time:	114.328s	
Memory Bound:	**50.7%**	**of Pipeline Slots**
L1 Bound:	12.6%	of Clockticks
DRAM Bound:		
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
NUMA: % of Remote Accesses:	0.0%	
Loads:	83,645,509,290	
Stores:	50,582,817,439	
LLC Miss Count:	**473,228,392**	
Local DRAM Access Count:	459,213,776	
Remote DRAM Access Count:	0	
Remote Cache Access Count:	0	
Average Latency (cycles):	44	
Total Thread Count:	1	
Paused Time:	0s	

- **Approximately 75% reduction in LLC misses**
- **Scaling issues may be caused by shared L3 cache and related resources**
  - L2 prefetching tested but resulted in runtime degradation

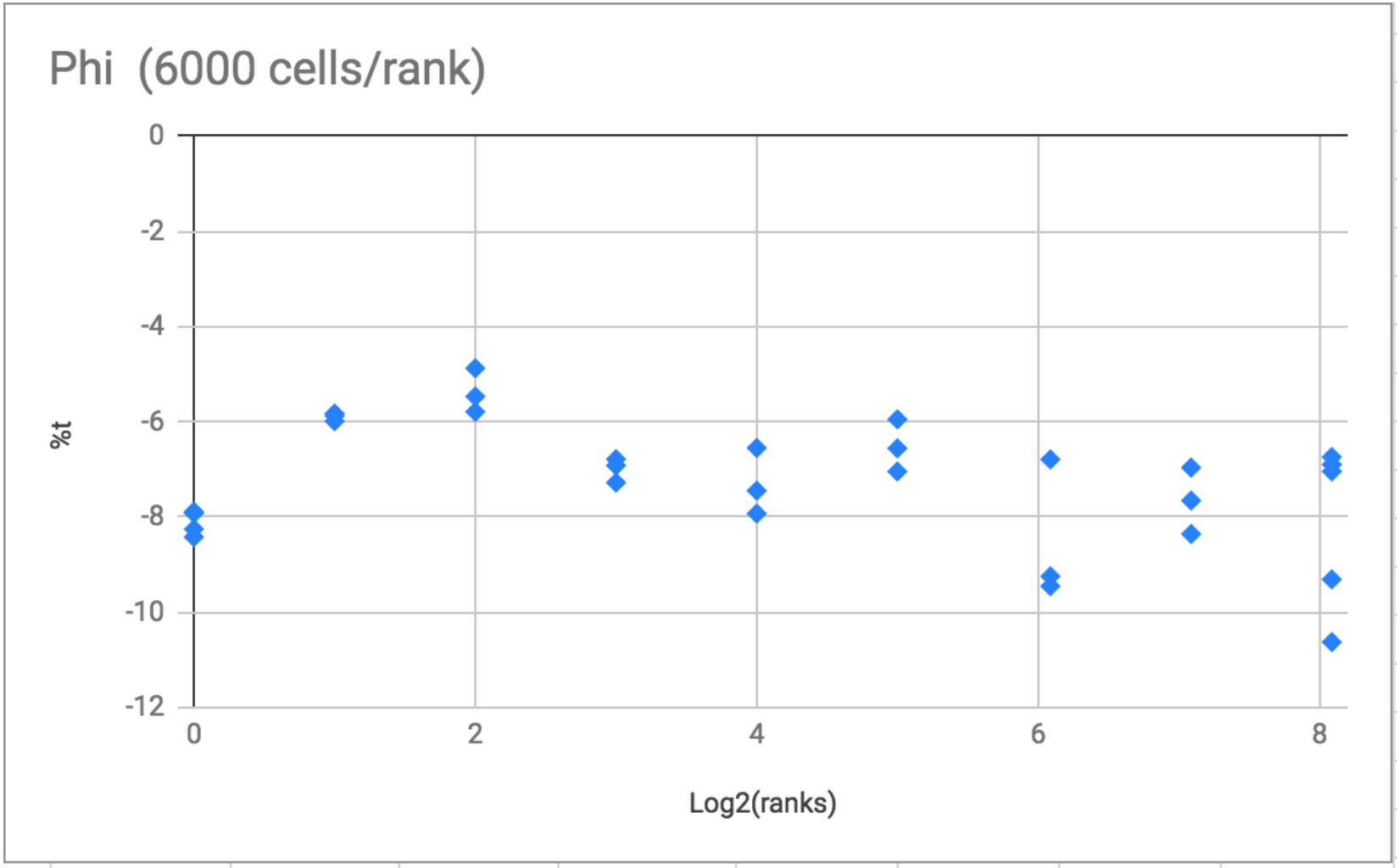
Even single cell PF schemes can have issues with scaling

KNL: Prefetching Within Cell

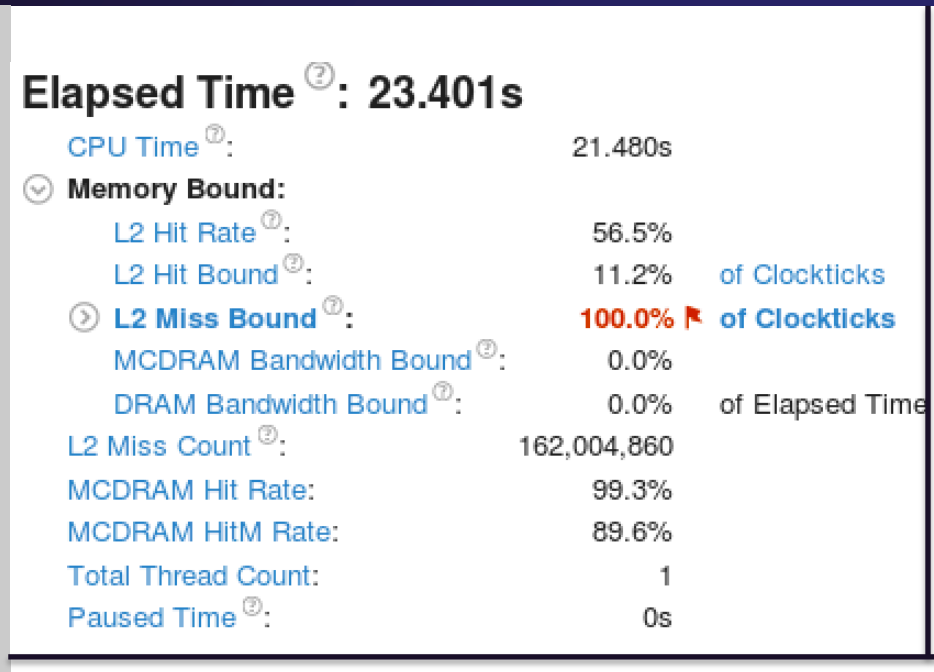
- **Using same prefetch scheme as best performing Haswell**
- **Single rank performance gain**
  - Miss reduction from 160M to 40M
  - Frequent ~7%-12% Runtime improvement (up to ~14% observed)
- **Appears to scale to multirank runs well**
  - Similar runtime improvement observed on 272 ranks (4 nodes)

KNL: Runtime Change Scaling

Small problem tested from 1 to 272 ranks (4 nodes)

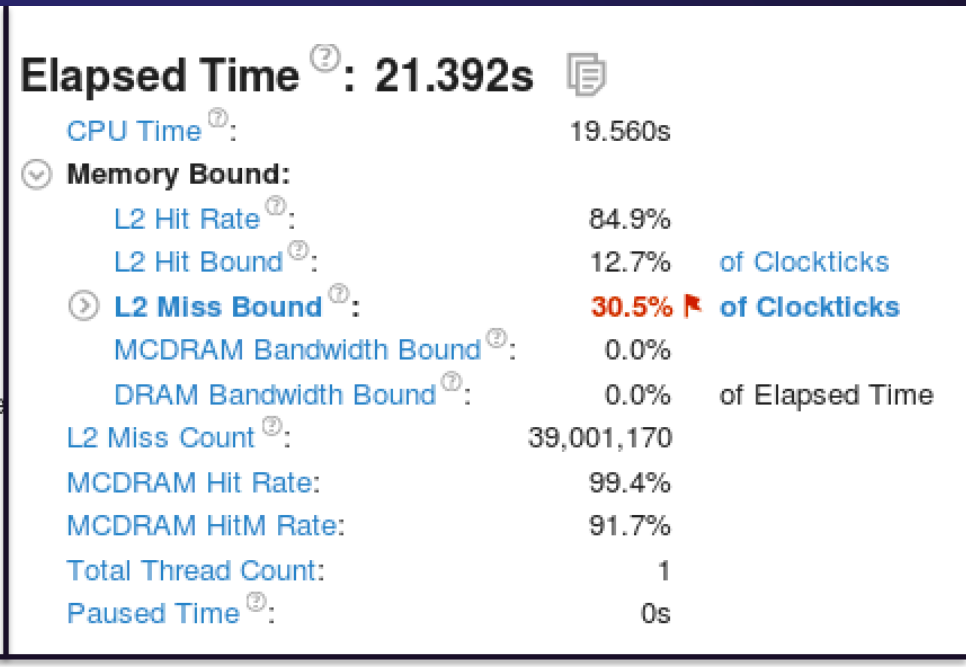


KNL: V-tune and Runtime Comparison



Elapsed Time: 23.401s

CPU Time:	21.480s	
Memory Bound:		
L2 Hit Rate:	56.5%	
L2 Hit Bound:	11.2%	of Clockticks
L2 Miss Bound:	**100.0%**	**of Clockticks**
MCDRAM Bandwidth Bound:	0.0%	
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
L2 Miss Count:	162,004,860	
MCDRAM Hit Rate:	99.3%	
MCDRAM HitM Rate:	89.6%	
Total Thread Count:	1	
Paused Time:	0s	



Elapsed Time: 21.392s

CPU Time:	19.560s	
Memory Bound:		
L2 Hit Rate:	84.9%	
L2 Hit Bound:	12.7%	of Clockticks
L2 Miss Bound:	**30.5%**	**of Clockticks**
MCDRAM Bandwidth Bound:	0.0%	
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
L2 Miss Count:	39,001,170	
MCDRAM Hit Rate:	99.4%	
MCDRAM HitM Rate:	91.7%	
Total Thread Count:	1	
Paused Time:	0s	

- **Once again approximately 75% reduction in LLC misses**
- **Better scaling may be explained by the “less shared” nature of the LLC for the KNL compared to Haswell.**

KNLs show promise for this prefetching scheme

Less Misses =/= Reduced Runtime

- **The vast majority of schemes tested on Haswell reduced misses (often by a significant amount) but had little effect on runtime**
  - Potentially overloading fetch “slots”?
  - Prefetch call overhead?
    - Thought: Put in pointless Prefetch calls and see the effect of overhead
- **Poor scaling despite similar cache miss reduction**
  - At 16 ranks, Haswell still showed ~75% miss reduction
  - More confusing: At 16 ranks KNL only showed ~55% miss reduction

Wrap-Up

- **Reduction in cache misses will not always correspond to decreased runtimes**
  - Could be overtaxing fetching resources
  - Some overhead to consider
- **Prefetching for next graph node results in graph dependence, which complicates scaling to higher ranks.**
- **More performance improvements due to prefetching seem to be available for KNL compared to Haswell**
  - Possibly due to fetching into shared L3 cache for Haswell compared to less-shared L2 for KNL (16 fast cores in 40MB versus 2 slow cores in 1 MB)
- **Implementation of prefetching for complex code seems to require a lot of guess and check**

Future Directions

- **Test prefetching scheme on larger “real” problems with full transport solve**
- **Adapt scheme to other mesh geometries beyond what was tested here**
- **Deeper dive into why a 75% reduction in cache misses does little for Haswell, as well as its poor scaling**
- **Gather more scaling data points (runtime and miss changes) out to higher number of ranks**
  - Can probably script this