Using VTune on Cori KNL Nodes

Zhengji Zhao
User Engagement Group

Cori KNL Training (Spring 2017)
June 9, 2017, Berkeley CA
VTune Introduction

• **An Intel profiling tool**
  – Focus on on-node performance
  – Works with serial and parallel codes

• **Provides both command line interface “amplxe-cl” and a gui “amplxe-gui”**
  – Recommended to use the command line tool to collect data via batch jobs, and then use the gui to display data later on a login node
  – NX is recommended to speed up the gui display

• **Available via the vtune modules on Cori**
  – The current default is 2017.up2 on Cori
  – VTune interfaces could be different from different versions
Running VTune on Cori KNL nodes

1. Compile your codes with `-g` and link the code dynamically with the `-dynamic` option for ftn, cc or CC
   - The `-debug inline-debug-info` is recommended to generate enhanced debug information for the inlined codes.
   - Use optimization flags as in the production use (`-O0` is not required)
   - Intel compilers are recommended

```
module swap craype-haswell craype-mic-knl
ftn -dynamic -qopenmp -O3 -g -debug inline-debug-info jacobi.f90 -o jac.x
```
2. To run VTune, you have to:

- Use `#SBATCH --perf=vtune` or `salloc --perf=vtune` with a batch or interactive job
- Load a `vtune` module before submitting jobs or using `salloc` to request nodes
- Use a Lustre file system, `$SCRATCH`

```
#Continued ...
mkdir $SCRATCH/vtune_runs
cp -p jac.x $SCRATCH/vtune_runs
cd $SCRATCH/vtune_runs
Module load vtune
salloc -N2 -C knl,quad,cache --qos=interactive -t 1:00:00 --perf=vtune
wait ...
```
Running VTune on Cori KNL nodes  --continued

• Sample job script to run VTune on Cori KNL

```bash
#!/bin/bash
#SBATCH -J vtune_test
#SBATCH -N 2
#SBATCH -C knl,quad,cache  #running in quad,cashe mode
#SBATCH --p debug
#SBATCH -t 00:30:00
#SBATCH --perf=vtune
#SBATCH -o myjob.o%j

module load vtune

srun -n 128 -c 4 amplxe-cl -collect general-exploration -r res_dir -trace-mpi -finalization-mode=none  -- ./a.out
```
Running VTune on Cori KNL nodes --continued

3. Collect data (on compute nodes)

#Continued …

module load vtune
export OMP_PROC_BIND=true
export OMP_PLACES=threads
export OMP_NUM_THREADS=8

# to run memory-access analysis:
srun -n 8 -c 32 --cpu_bind=cores amplxe-cl –collect memory-access -r
res_dir -trace-mpi –finalization-mode=none -- ./jac.x

– The -finalization-mode=none is used so to finalize data later on a faster
  node (e.g., on a login node)
– The -data-limit=0 option can be used to get >500MB profiling data
– do amplxe-cl –help collect for the available analysis types, and command
  line options; do amplxe-cl –help collect <analysis name> for the knob
  option fore the analysis.
Running VTune on Cori KNL nodes --continued

• **Available analyses**

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>advanced-hotspots</td>
<td>Advanced Hotspots</td>
</tr>
<tr>
<td>concurrency</td>
<td>Concurrency</td>
</tr>
<tr>
<td>cpugpu-concurrency</td>
<td>CPU/GPU Concurrency</td>
</tr>
<tr>
<td>disk-io</td>
<td>Disk Input and Output</td>
</tr>
<tr>
<td>general-exploration</td>
<td>General Exploration</td>
</tr>
<tr>
<td>gpu-hotspots</td>
<td>GPU Hotspots</td>
</tr>
<tr>
<td>hotspots</td>
<td>Basic Hotspots</td>
</tr>
<tr>
<td>hpc-performance</td>
<td>HPC Performance Characterization</td>
</tr>
<tr>
<td>locksandwaits</td>
<td>Locks and Waits</td>
</tr>
<tr>
<td>memory-access</td>
<td>Memory Access</td>
</tr>
<tr>
<td>sgx-hotspots</td>
<td>SGX Hotspots</td>
</tr>
<tr>
<td>tsx-exploration</td>
<td>TSX Exploration</td>
</tr>
<tr>
<td>tsx-hotspots</td>
<td>TSX Hotspots</td>
</tr>
</tbody>
</table>

**Example amplxe-cl command line with more knob options:**

```
srun -n 4 -c 24 amplxe-cl -collect memory-access -knob analyze-openmp=true -knob analyze-mem-objects=true -r res_dir -trace-mpi -- ./a.out```

Running VTune on Cori KNL nodes  --continued

4. Display the collected data using the VTune gui on a login node
   – amplxe-gui -- > Open results, then the Vtune gui finalize the data first and then display
   – Or you can finalize the data outside the gui by

   amplxe-cl -finalize -finalization-mode=full -r <your result dir>
   -search-dir <path to your binary>
Memory access – Memory usage viewpoint

Memory Bandwidth Usage

- Memory Bound:
  - L2 Hit Rate: 89.0%
  - L2 Hit Bound: 9.6% of Clockticks
  - L2 Miss Bound: 16.2% of Clockticks
  - MCDRAM Cache Bandwidth Bound: 4.0%
  - MCDRAM Flat Bandwidth Bound: 4.9%
  - DRAM Bandwidth Bound: 0.0% of Elapsed Time

- L2 Miss Count: 7,038,211,140
- MCDRAM Hit Rate: 98.9%
- MCDRAM HIRM Rate: 94.3%
- Total Thread Count: 65
- Paused Time: 0s

Bandwidth Utilization

- Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

- Bandwidth Domain: DRAM, GB/sec

- Bandwidth Utilization Histogram
  - This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Memory Usage

This view uses hardware event-based metrics to characterize how effectively your application uses memory resources and identify potential memory access related issues like excessive access to remote memory on NUMA platforms, hitting DRAM or Intel QPI bandwidth limit, and others. It provides various performance metrics for both the application code and memory objects arrays.

- Show the analysis description when result opens
Memory Access – Memory usage bottom-up viewpoint
Memory access – Hotspots viewpoint

**Elapsed Time**: 146.850s
- CPU Time: 6696.230s
- Instructions Retired: 6,609,218,000,000
- CPI Rate: 1.517
- CPU Frequency Ratio: 1.089
- Total Thread Count: 65
- Paused Time: 0s

**Top Hotspots**
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>libbiomp5.so</td>
<td>libbiomp5.so</td>
<td>2982.694s</td>
</tr>
<tr>
<td>vasp_std</td>
<td>vasp_std</td>
<td>2559.634s</td>
</tr>
<tr>
<td>vmlinux</td>
<td>vmlinux</td>
<td>538.931s</td>
</tr>
<tr>
<td>libmpich_intel.so.0.6.1</td>
<td>libmpich_intel.so.0.6.1</td>
<td>304.980s</td>
</tr>
<tr>
<td>libugni.so.0.6.0</td>
<td>libugni.so.0.6.0</td>
<td>157.160s</td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td>151.856s</td>
</tr>
</tbody>
</table>

**Average Bandwidth**

<table>
<thead>
<tr>
<th>Package</th>
<th>Total, GB/sec</th>
<th>Read, GB/sec</th>
<th>Write, GB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>package_0</td>
<td>1.197</td>
<td>0.710</td>
<td>0.419</td>
</tr>
</tbody>
</table>

**CPU Usage Histogram**
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU usage value.

**Hotspots**
This view uses hardware event-based collection to display code regions (modules, functions, and so on) that consume a lot of CPU time (hotspots).

Use this view to:
- Identify and analyze hotspots with the highest CPU time values and high CPI rate.
- Filter hotspots to identify hotspots for a specific time region.
- Analyze the source by double-clicking a hotspot and identify its most critical code lines.

Press F1 for help on each window.

Show the analysis description when result opens.
General exploration – General exploration viewpoint

**General Exploration**

This view uses **hardware event-based metrics** to show code regions that experienced potentially significant architectural bottlenecks. Hover over a metric name in the **grid** for the metric description.

**Use this view to:**
- Identify code regions (modules, functions, and so on) with the highest execution time.
- Analyze detected **hardware issues**, highlighted by pink cells and get tuning recommendations.

Press **F1** for help on each window.

**Show the analysis description when result opens**

---

**Elapsed Time**: 93.154s

- **Clockticks**: 6,401,472,000,000
- **Instructions Retired**: 3,850,862,200,000
- **CPI Rate**: 1.662
- **MUX Reliability**: 0.999

1. **Front-End Bound**: 32.39% of Pipeline Slots
2. **Bad Speculation**: 2.5% of Pipeline Slots
3. **Back-End Bound**: 29.11% of Pipeline Slots
4. **Retiring**: 36.19% of Pipeline Slots

- **Total Thread Count**: 69
- **Paused Time**: 0s

---

**CPU Usage Histogram**

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU usage.
General exploration – hotspots

Elapsed Time: 93.154s

CPU Time: 4267.914s
Instructions Retired: 3,850,662,200,000
CPU Rate: 1.662
CPU Frequency Ratio: 1.071
Total Thread Count: 69
Paused Time: 0s

Top Hotspots
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>libiomp5.so</td>
<td>libiomp5.so</td>
<td>1955.410s</td>
</tr>
<tr>
<td>[vasp_std]</td>
<td>vasp_std</td>
<td>1821.137s</td>
</tr>
<tr>
<td>[vmlinux]</td>
<td>vmlinux</td>
<td>139.823s</td>
</tr>
<tr>
<td>[libmpich_intel.so.3.0.1]</td>
<td>libmpich_intel.so.3.0.1</td>
<td>117.726s</td>
</tr>
<tr>
<td>[libm.so]</td>
<td>libm.so</td>
<td>115.019s</td>
</tr>
<tr>
<td>[Others]</td>
<td></td>
<td>118.799s</td>
</tr>
</tbody>
</table>

CPU Usage Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.
General Exploration – Hotspots bottom-up viewpoint
HPC Performance Characterization viewpoint

HPC Performance Characterization

- Elapsed Time: 98.250s
- CPU Utilization: 17.2%
  - Average CPU Usage: 16.831 Out of 272 logical CPUs
- MPI Rank on the Critical Path:
  - MPI Busy Wall Time
  - Serial Time (outside any parallel region)
  - OpenMP Potential Gain
- CPU Usage Histogram
- Back-End Bound: 28.1%
  - L2 Hit Bound: 8.8% of Clockticks
  - L2 Miss Bound: 23.3% of Clockticks
  - MCDRAM Cache Bandwidth Bound: 0.1%
  - MCDRAM Flat Bandwidth Bound: 0.1%
  - DRAM Bandwidth Bound: 0.2%
- SIMD Instructions per Cycle: 0.138
  - FP instruction Mix:
    - % of Packed SIMD Instr: 85.2%
    - % of Scalar SIMD Instr: 13.8%
- Top Loops/Functions with FPU Usage by CPU Time
  - This section provides information for the most time consuming loops/functions with floating point operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time</th>
<th>SIMD Instructions per Cycle</th>
<th>Vector instruction Set</th>
<th>Loop Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>libemps.so</td>
<td>2228.553s</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>use_smpf</td>
<td>1825.893s</td>
<td>0.329</td>
<td>0.329</td>
<td></td>
</tr>
<tr>
<td>optimiser</td>
<td>170.298s</td>
<td>0.005</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>libemps_intel_x_2.0.1</td>
<td>135.620s</td>
<td>0.049</td>
<td>0.049</td>
<td></td>
</tr>
<tr>
<td>libemps_intel_x_2.1</td>
<td>113.210s</td>
<td>0.206</td>
<td>0.206</td>
<td></td>
</tr>
<tr>
<td>[Others]</td>
<td>94.890s</td>
<td>N/A*</td>
<td>N/A*</td>
<td></td>
</tr>
</tbody>
</table>

*N/A is applied to non-comparable metrics.

Use this view to:
- Explore OpenMP code optimization Potential Gain by parallelization inefficiencies representing their wall time cost.
- Explore precise Imbalance metrics to identify imbalance of working threads on barriers.
- Analyze barrier-to-barrier region segments to explore performance of OpenMP work-sharing constructs and barrier cost inside a region.
- For hybrid MPI and OpenMP applications using Intel MPI library, analyze per-rank MPI communication busy wait time.

Press F1 for help on each window.
HPC Performance Characterization viewpoint

Elapsed Time: 98.250s

CPU Utilization: 17.2%
- Average CPU Usage: 46.831 out of 272 logical CPUs
- MPI Rank on the Critical Path:
  - MPI Busy Wait Time:
  - Serial Time (outside any parallel region):
  - OpenMP Potential Gain:

CPU Usage Histogram
- This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.
HPC Performance Characterization viewpoint

**Back-End Bound**: 28.1%

- L2 Hit Bound: 98.0% of Clockticks
- L2 Miss Bound: 23.1% of Clockticks
- MCDRAM Cache Bandwidth Bound: 0.1%
- MCDRAM Flat Bandwidth Bound: 0.1%
- DRAM Bandwidth Bound: 0.0%

**Bandwidth Utilization**
Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

**Bandwidth Utilization Histogram**
This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them, for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

**Top Functions with High Bandwidth Utilization**
This section shows top functions, sorted by LLC Misses that were executing when bandwidth utilization was high for the domain selected in the histogram area.

*No data to show. The collected data is not sufficient.
HPC Performance Characterization viewpoint

Back-End Bound: 28.1%
- L2 Hit Bound: 8.8% of Clockticks
- L2 Miss Bound: 23.1% of Clockticks
- MCDRAM Cache Bandwidth Bound: 0.1%
- MCDRAM Flat Bandwidth Bound: 0.1%
- DRAM Bandwidth Bound: 0.0%

Bandwidth Utilization
Explore bandwidth utilization over time using the histogram and identifymemory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: MCDRAM Fill, GB/sec

Bandwidth Utilization Histogram
This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Top Functions with High Bandwidth Utilization
This section shows top functions, sorted by LLC Misses that were executing when bandwidth utilization was high for the domain selected in the histogram area.

<table>
<thead>
<tr>
<th>Function</th>
<th>LLC Miss Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>[vasp_std]</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
HPC Performance Characterization viewpoint

<table>
<thead>
<tr>
<th>Bandwidth Domain / Bandwidth Utilization Type / Function</th>
<th>CPU Time</th>
<th>Back-End Bound</th>
<th>SIMD Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM, GB/sec</td>
<td>4601.087s</td>
<td>4.750s</td>
<td>L2 Hit Bound</td>
</tr>
<tr>
<td>MCDRAM Flat, GB/sec</td>
<td>4601.087s</td>
<td>4.750s</td>
<td>L2 Miss Bound</td>
</tr>
<tr>
<td>Low</td>
<td>2751.784s</td>
<td>1.130s</td>
<td>% of Packed SIMD Inst.</td>
</tr>
<tr>
<td>Medium</td>
<td>1844.685s</td>
<td>3.840s</td>
<td>% of Scalar SIMD Inst.</td>
</tr>
<tr>
<td>High</td>
<td>4.540s</td>
<td>0s</td>
<td>% of Packed SIMD Inst.</td>
</tr>
</tbody>
</table>

**Elapsed Time:** 98.250s

**CPU Utilization:** 17.2%

**Average CPU Usage:** 45.831 Out

**MPI Rank on the Critical Path:** Serial Time (outside any parallel)

**OpenMP Potential Gain:**

**CPU Usage Histogram**

**Back-End Bound:** 28.1%

- L2 Hit Bound
- L2 Miss Bound
- MCDRAM Cache Bandwidth Bound
- MCDRAM Flat Bandwidth Bound
- DRAM Bandwidth Bound

**SIMD Instructions per Cycle:** 0%

**FP Instruction Mix:**

**% of Packed SIMD Inst.:** 99.2%

**% of Scalar SIMD Inst.:** 13.8%

**Thread**
- Effective Time
- Spin and Overhead
- CPU Time
- Spin and Overhead

**CPU Time**
- Total, GB/sec
- Read, GB/sec
- Write, GB/sec

**MCDRAM Flat Mode**
- MCDRAM Bandwidth
- MCDRAM Read Bandwidth
- MCDRAM Write Bandwidth

**MCDRAM Cache**
- MCDRAM Bandwidth
- MCDRAM Read Bandwidth
- MCDRAM Write Bandwidth
SIMD Instructions per Cycle: 0.138

- FP Instruction Mix:
  - % of Packed SIMD Instr.: 86.2%
  - % of Scalar SIMD Instr.: 13.8%

- Top Loops/Functions with FPU Usage by CPU Time
  This section provides information for the most time consuming loops/functions with floating point operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time (s)</th>
<th>SIMD Instructions per Cycle</th>
<th>Vector Instruction Set</th>
<th>Loop Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[libiomp5.so]</td>
<td>2228.553s</td>
<td>0.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[vasp_std]</td>
<td>1825.883s</td>
<td>0.329</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[vmlinux]</td>
<td>170.380s</td>
<td>0.005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[libmpich_intel.so.3.0.1]</td>
<td>135.620s</td>
<td>0.048</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[libimf.so]</td>
<td>115.210s</td>
<td>0.206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Others]</td>
<td>94.890s</td>
<td>N/A*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*N/A is applied to non-summable metrics.*
Advanced Hotspots – hotspots viewpoint

**Elapsed Time**: 142.760s

- **CPU Time**: 6564.929s
- **Instructions Retired**: 6,561,240,000,000
- **CPI Rate**: 1.498
- **CPU Frequency Ratio**: 1.070
- **Total Thread Count**: 67
- **Paused Time**: 0s

**Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[libomp5.so]</td>
<td>libomp5.so</td>
<td>2981.554s</td>
</tr>
<tr>
<td>[vasp.std]</td>
<td>vasp_std</td>
<td>2532.204s</td>
</tr>
<tr>
<td>[vmlinux]</td>
<td>vmlinux</td>
<td>456.501s</td>
</tr>
<tr>
<td>[libmpich_intel.so.3.0.1]</td>
<td>libmpich_intel.so.3.0.1</td>
<td>362.210s</td>
</tr>
<tr>
<td>[libugni.so.0.6.0]</td>
<td>libugni.so.0.6.0</td>
<td>157.450s</td>
</tr>
<tr>
<td>[Others]</td>
<td></td>
<td>135.110s</td>
</tr>
</tbody>
</table>

**CPU Usage Histogram**

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU usage value.
More resources

• At NERSC
  – NERSC Vtune page, more details about how to run Vtune
  – Various training slides by NERSC and Intel staff, available at NERSC user training web site

• At Intel
Acknowledgement

Thank you!
Application Performance Snapshot

• Intel profiling tool is recently made available on Cori (under beta testing)
  – Application Performance Snapshot provides a quick view into a shared memory or MPI application's use of available hardware (CPU, FPU, and Memory). It analyzes your application's time spent in MPI, MPI and OpenMP imbalance, memory access efficiency, FPU usage, and I/O and memory footprint, and displays basic performance enhancement opportunities for systems using Intel® platform in the standard output and also stores the analysis result in the html format to displays.
APS report

Application Performance Snapshot

221.32s
Elapsed Time

2.41
CPI
(MAX 2.43, MIN 2.35)

Your application is MPI bound. This may be caused by high busy wait time inside the library (imbalance), non-optimal communication schema or MPI library settings. Use MPI profiling tools like Intel® Trace Analyzer and Collector to explore performance bottlenecks.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Current run</th>
<th>Target</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI Time</td>
<td>22.64%</td>
<td>&lt;15%</td>
<td></td>
</tr>
<tr>
<td>OpenMP Imbalance</td>
<td>1.77%</td>
<td>&lt;10%</td>
<td></td>
</tr>
<tr>
<td>Back-End Stalls</td>
<td>70.14%</td>
<td>&lt;20%</td>
<td></td>
</tr>
<tr>
<td>SIMD Instr. per Cycle</td>
<td>0.16%</td>
<td>&gt;1</td>
<td></td>
</tr>
<tr>
<td>I/O Bound</td>
<td>0.01%</td>
<td>&lt;10%</td>
<td></td>
</tr>
</tbody>
</table>

**MPI Time**
22.64% of Elapsed Time (50.10s)

- **MPI Imbalance**: 0.00% of Elapsed Time (0.00s)

<table>
<thead>
<tr>
<th>Top 5 MPI Functions</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allreduce</td>
<td>5.86</td>
</tr>
<tr>
<td>Waitall</td>
<td>5.57</td>
</tr>
<tr>
<td>Barrier</td>
<td>4.93</td>
</tr>
<tr>
<td>Ireduce</td>
<td>2.62</td>
</tr>
<tr>
<td>Bcast</td>
<td>2.23</td>
</tr>
</tbody>
</table>

**I/O Bound**
0.01%
(AVG 4.07, PEAK 0.07)

- **Read**: AVG 7.0 MB, MAX 240.4 MB
- **Write**: AVG 27.9 KB, MAX 453.8 KB

**OpenMP Imbalance**
1.77% of Elapsed Time (3.91s)

**Back-End Stalls**
70.14% of pipeline slots

- **L2 Hit Bound**: 18.46%
- **L2 Miss Bound**: 41.26%

**Memory Footprint**
AVG 10739.15 MB, PEAK 11728.88 MB

**SIMD Instr. per Cycle**
0.16%

- FP Instruction Mix:
  - % of Packed SIMD Instr.: 94.21%
  - % of Scalar SIMD Instr.: 5.79%
To use on Cori

• Link dynamically, and run with the following script

```bash
#!/bin/bash
#SBATCH --J aps_test
#SBATCH -N 2
#SBATCH -C knl,quad,cache  #running in quad,cashe mode
#SBATCH --p debug
#SBATCH -t 00:30:00
#SBATCH --perf=vtune
#SBATCH -o myjob.o%j

module load aps
. $APS_DIR/apsvars.sh

export OMP_NUM_THREADS=8

srun -n32 -c16 --cpu_bind=cores aps.sh jax-dynamic.x
aps.sh --report stat_$rundate  # $rundate looks like this 20170609
```