Roofline on CPU-based Systems

Samuel Williams
Computational Research Division
Lawrence Berkeley National Lab
SWWilliams@lbl.gov

With slides from Charlene Yang, Doug Doerfler, and Zakhar Matveev
Acknowledgements

- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This material is based upon work supported by the DOE RAPIDS SciDAC Institute.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.
Machine Characterization
Machine Characterization

- “Theoretical Performance” numbers can be highly optimistic…
  - Pin BW vs. sustained bandwidth
  - TurboMode / Underclock for AVX
  - compiler failings on high-AI loops.

- LBL developed the Empirical Roofline Toolkit (ERT)…
  - Characterize CPU/GPU systems
  - Peak Flop rates
  - Bandwidths for each level of memory
  - MPI+OpenMP/CUDA == multiple GPUs

https://bitbucket.org/berkeleylab/cs-roofline-toolkit/
https://github.com/cyanguwa/nersc-roofline/
https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/
# ERT Configuration

<table>
<thead>
<tr>
<th>Kernel.c</th>
<th>config.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop over ntrials</td>
<td>ERT_FLOPS 1,2,4,8,16,32,64</td>
</tr>
<tr>
<td>distribute dataset on threads and each computes ERT_FLOPS</td>
<td>ERT_MPI_PROCS 2,4,8,16,32,64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel.h</th>
<th>ERT_OPENMP_THREADS 1-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERT_FLOPS=1: a = b + c</td>
<td>ERT_MEMORY_MAX 1073741824</td>
</tr>
<tr>
<td>ERT_FLOPS=2: a = a x b + c</td>
<td>ERT_WORKING_SET_MIN 1</td>
</tr>
<tr>
<td></td>
<td>ERT_TRIALS_MIN 1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver.c (uses some Macros from config.txt)</th>
<th>Job script</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize MPI, OpenMP</td>
<td>./ert config.txt</td>
</tr>
<tr>
<td>loop over dataset sizes &lt;= ERT_MEMORY_MAX</td>
<td>ert (Python)</td>
</tr>
<tr>
<td>loop over trial sizes &gt;= ERT_TRIALS_MIN</td>
<td>create directories</td>
</tr>
<tr>
<td>start timer</td>
<td>loop over ERT_FLOPS, MPI_PROCS/OMP_THREADS</td>
</tr>
<tr>
<td>call kernel</td>
<td>call driver, kernel</td>
</tr>
<tr>
<td>end timer</td>
<td></td>
</tr>
</tbody>
</table>
ERT Caveats

- Nominally, ERT runs a series of benchmarks
  - Read-modify-write Polynomial of degree-K on a vector of size N
  - Trivially auto-vectorized
  - Demands a unroll-and-jam or large OOO window to hit peak.
  - 1:1 Read:Write ratio
  - Varies both K and N

- From these it extrapolates cache capacities and bandwidths
  - By convention it labels the largest/slowest ‘DRAM’ and the smallest/fastest ‘L1’
  - If N<LLC size, then it will identify the LLC ‘DRAM’ (e.g. on KNL, N>16GB)
  - On architectures that don’t cache writes in the L1 (or are WT), ERT will label L2 as ‘L1’
  - On architectures that have a 2:1 read:write cache bandwidth, ERT will underestimate aggregate cache bandwidth (it uses a 1:1 benchmark)
Application Characterization
Measuring AI

To characterize execution with Roofline we need…

- Time
- Flops (=> flop’s / time)
- Data movement between each level of memory (=> Flop’s / GB’s)

We can look at the full application…

- Coarse grained, 30-min average
- Misses many details and bottlenecks

or we can look at individual loop nests…

- Requires auto-instrumentation on a loop by loop basis
- Moreover, we should probably differentiate data movement or flops on a core-by-core basis.
## How Do We Count Flop’s?

<table>
<thead>
<tr>
<th>Manual Counting</th>
<th>Perf. Counters</th>
<th>Binary Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go thru each loop nest and count the number of FP operations</td>
<td>Read counter before/after</td>
<td>Automated inspection of assembly at run time</td>
</tr>
<tr>
<td>Works best for deterministic loop bounds</td>
<td>More Accurate</td>
<td>Most Accurate</td>
</tr>
<tr>
<td>or parameterize by the number of iterations (recorded at run time)</td>
<td>Low overhead (&lt;%) == can run full MPI applications</td>
<td>FMA-, VL-, and mask-aware</td>
</tr>
<tr>
<td>Not scalable</td>
<td>Can detect load imbalance</td>
<td>Can count instructions by class/type</td>
</tr>
<tr>
<td></td>
<td>Requires privileged access</td>
<td>Can detect load imbalance</td>
</tr>
<tr>
<td></td>
<td>Requires manual instrumentation (+overhead) or full-app characterization</td>
<td>Can include effects from non-FP instructions</td>
</tr>
<tr>
<td></td>
<td>Broken counters = garbage</td>
<td>Automated application to multiple loop nests</td>
</tr>
<tr>
<td></td>
<td>May not differentiate FMADD from FADD</td>
<td>&gt;10x overhead (short runs / reduced concurrency)</td>
</tr>
<tr>
<td></td>
<td>No insight into special pipelines</td>
<td></td>
</tr>
</tbody>
</table>
How Do We Measure Data Movement?

**Manual Counting**
- Go thru each loop nest and estimate how many bytes will be moved
- Use a mental model of caches
  - Works best for simple loops that stream from DRAM (stencils, FFTs, spare, ...)
  - N/A for complex caches
  - Not scalable

**Perf. Counters**
- Read counter before/after
  - Applies to full hierarchy (L2, DRAM,)
  - Much more Accurate
  - Low overhead (<%) == can run full MPI applications
  - Can detect load imbalance
  - Requires privileged access
  - Requires manual instrumentation (+overhead) or full-app characterization

**Cache Simulation**
- Build a full cache simulator driven by memory addresses
  - Applies to full hierarchy and multicore
  - Can detect load imbalance
  - Automated application to multiple loop nests
  - Ignores prefetchers
  - >10x overhead (short runs / reduced concurrency)
Performance Counter Issues
Performance Counter Limitations

- Capture aspects architects (not programmers) think are important
- May lack important detail
- Not standardized (vendor-specific)
- Not required to be functional or correct (not part of the ISA)
Performance Counters and SIMD

- SIMD instruction sets are ever evolving.
- Today, they can incorporate…
  - Different Vector Lengths (VL)… 128b, 256b, 512b, …
  - Different precisions… double, single, half, … 8x64b, 16x32b, or 32x16b
  - Use of FMA (1 or 2 flops per element)
  - Use of masks (predicates) to disable execution on certain lanes.
- Thus, a performance counter might be:
  - VL-aware (#operations scales with VL)
  - Precision-aware (#operations increases with reduced precision)
  - FMA-aware (FMAs are 2 flops per element vs. 1)
  - Mask-aware (#operations only incudes unmasked operations)
Roofline with LIKWID
LIKWID

- LIKWID provides easy to use wrappers for measuring performance counters...
  - Works on NERSC production systems
  - Distills counters into user-friendly metrics (e.g. MCDRAM Bandwidth)
  - Minimal overhead (<1%)
  - Scalable in distributed memory (MPI-friendly)
  - Fast, high-level characterization
  - No timing breakdowns
  - Suffers from Garbage-in/Garbage Out
    (i.e. hardware counter must be sufficient and correct)

https://github.com/RRZE-HPC/likwid
<table>
<thead>
<tr>
<th>Likwid Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>likwid-topology</td>
<td>node topology</td>
</tr>
<tr>
<td>likwid-pin</td>
<td>process/thread affinity</td>
</tr>
<tr>
<td>likwid-memsweeper</td>
<td>cleanup memory &amp; LLC</td>
</tr>
<tr>
<td>likwid-powermeter</td>
<td>power measurements</td>
</tr>
<tr>
<td>likwid-setFrequencies</td>
<td>CPU/uncore frequency manipulation</td>
</tr>
<tr>
<td>likwid-perfctr</td>
<td>hardware counter measurements</td>
</tr>
<tr>
<td>likwid-mpirun</td>
<td>hardware counter + MPI</td>
</tr>
<tr>
<td>likwid-bench</td>
<td>micro-benchmarking</td>
</tr>
<tr>
<td>likwid-agent</td>
<td>system monitoring</td>
</tr>
<tr>
<td>likwid-genTopoCfg</td>
<td>generate and store topology file</td>
</tr>
<tr>
<td>HWThread</td>
<td>Thread</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
</tbody>
</table>
### Cache Topology

<table>
<thead>
<tr>
<th>Level: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size: 32 KB</td>
</tr>
<tr>
<td>Cache groups:</td>
</tr>
<tr>
<td>( 0 68 136 284 ) ( 1 69 137 285 ) ( 2 78 138 286 ) ( 3 71 139 287 ) ( 4 72 140 288 ) ( 5 73 141 289 ) ( 6 74 142 288 ) ( 7 75 143 282 ) ( 8 76 144 224 ) ( 9 77 145 213 ) ( 10 78 146 214 ) ( 11 79 147 215 ) ( 12 80 148 216 ) ( 13 81 149 217 ) ( 14 82 150 218 )</td>
</tr>
<tr>
<td>( 38 106 174 242 ) ( 39 107 175 243 ) ( 40 108 176 244 ) ( 41 109 177 245 ) ( 42 110 178 246 ) ( 43 111 179 247 ) ( 44 112 180 248 ) ( 45 113 181 249 )</td>
</tr>
</tbody>
</table>

### NUMA Topology

<table>
<thead>
<tr>
<th>Level: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size: 1 MB</td>
</tr>
<tr>
<td>Cache groups:</td>
</tr>
<tr>
<td>( 0 68 136 284 1 69 137 285 2 70 138 286 3 71 139 287 4 72 140 288 5 73 141 289 6 74 142 288 7 75 143 282 8 76 144 224 9 77 145 213 10 78 146 214 11 79 147 215 12 80 148 216 13 81 149 217 14 82 150 218 15 83 151 219 16 84 152 220 17 85 153 221 18 86 154 222 19 87 155 223 20 88 156 224 21 89 157 225 22 90 158 226 23 91 159 227 24 92 160 228 25 93 161 229 26 94 162 230 27 95 163 231 28 96 164 232 29 97 165 233 30 98 166 234 31 99 167 235 32 100 168 236 33 101 169 237 34 102 170 238 35 103 171 239 36 104 172 240 37 105 173 241 38 106 174 242 39 107 175 243 40 108 176 244 41 109 177 245 42 110 178 246 43 111 179 247 44 112 180 248 45 113 181 249</td>
</tr>
</tbody>
</table>

### Numa domains

<table>
<thead>
<tr>
<th>Domain: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors:</td>
</tr>
<tr>
<td>( 0 68 136 284 1 69 137 285 2 70 138 286 3 71 139 287 4 72 140 288 5 73 141 289 6 74 142 288 7 75 143 281 8 76 144 224 9 77 145 213 10 78 146 214 11 79 147 215 12 80 148 216 13 81 149 217 14 82 150 218 15 83 151 219 16 84 152 220 17 85 153 221 18 86 154 222 19 87 155 223 20 88 156 224 21 89 157 225 22 90 158 226 23 91 159 227 24 92 160 228 25 93 161 229 26 94 162 230 27 95 163 231 28 96 164 232 29 97 165 233 30 98 166 234 31 99 167 235 32 100 168 236 33 101 169 237 34 102 170 238 35 103 171 239 36 104 172 240 37 105 173 241</td>
</tr>
</tbody>
</table>

### Distances

<table>
<thead>
<tr>
<th>Distances: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free memory: 92324.1 MB</td>
</tr>
<tr>
<td>Total memory: 96563.2 MB</td>
</tr>
</tbody>
</table>
likwid-pin

- `likwid-pin -c N:0,8,16,24 ./xthi.x`
- `likwid-pin -c S0:0,8@S1:0,8 ./xthi.x`

Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
Hello from rank 0, thread 1, on nid00028. (core affinity = 8)
Hello from rank 0, thread 2, on nid00028. (core affinity = 16)
Hello from rank 0, thread 3, on nid00028. (core affinity = 24)

- `likwid-pin -c E:N:128:2:4 ./xthi.x`

Hello from rank 0, thread 0, on nid02308. (core affinity = 0)
Hello from rank 0, thread 1, on nid02308. (core affinity = 68)
Hello from rank 0, thread 2, on nid02308. (core affinity = 1)
Hello from rank 0, thread 3, on nid02308. (core affinity = 69)
* snip *
Hello from rank 0, thread 126, on nid02308. (core affinity = 63)
Hello from rank 0, thread 127, on nid02308. (core affinity = 131)

- `likwid-perfctr` takes the same specification as its processor list
Profiling with LIKWID

- likwid-perfctr (threaded) + likwid-mpirun (MPI/hybrid)

- no GUI
- low overhead -> SDE, VTune, etc
- no code instrumentation required -> CrayPat-tracing
- no root access required -> VTune
- no extra modules required to be installed -> VTune

- use Linux ‘msr’ module to access MSR (Model Specific Register) files

- Cori:
  module load vtune
  sbatch/salloc --perf=likwid
  module load likwid
Profiling with LIKWID (2)

- Alternately, one can construct a script and monitor only process 0

```bash
srun -n8 -c32 ./a.out args
srun -n8 -c32 ./perfctr.sh ./a.out args

where perfctr.sh is
#!/bin/bash
let SLURM_MPI_RANK=$SLURM_PROCID
if [ $SLURM_MPI_RANK = 0 ]; then
    # only process 0 runs likwid and it monitors only logical CPUs 0-31
    likwid-perfctr -C 0-31 -g CACHES $@
else
    $@
fi
```
<table>
<thead>
<tr>
<th>Group name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBM_OFFCORE</td>
<td>Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)</td>
</tr>
<tr>
<td>TLB_INSTR</td>
<td>L1 Instruction TLB miss rate/ratio</td>
</tr>
<tr>
<td>FLOPS_SP</td>
<td>Single Precision MFLOP/s</td>
</tr>
<tr>
<td>BRANCH</td>
<td>Branch prediction miss rate/ratio</td>
</tr>
<tr>
<td>L2CACHE</td>
<td>L2 cache miss rate/ratio</td>
</tr>
<tr>
<td>ENERGY</td>
<td>Power and Energy consumption</td>
</tr>
<tr>
<td>FRONTEND_STALLS</td>
<td>Frontend stalls</td>
</tr>
<tr>
<td>ICACHE</td>
<td>Instruction cache miss rate/ratio</td>
</tr>
<tr>
<td>TLB_DATA</td>
<td>L2 data TLB miss rate/ratio</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory bandwidth in MBytes/s</td>
</tr>
<tr>
<td>DATA</td>
<td>Load to store ratio</td>
</tr>
<tr>
<td>L2</td>
<td>L2 cache bandwidth in MBytes/s</td>
</tr>
<tr>
<td>FLOPS_DP</td>
<td>Double Precision MFLOP/s</td>
</tr>
<tr>
<td>CLOCK</td>
<td>Power and Energy consumption</td>
</tr>
<tr>
<td>HBM_CACHE</td>
<td>Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)</td>
</tr>
<tr>
<td>HBM</td>
<td>Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)</td>
</tr>
<tr>
<td>UOPS_STALLS</td>
<td>UOP retirement stalls</td>
</tr>
</tbody>
</table>
Using LIKWID for Roofline

- GPP kernel from BerkeleyGW
- Arithmetic Intensity = FLOPS / Bytes (= SDE / VTune)
  = FLOPS/sec / Bytes/sec
  = FLOPS\textsubscript{DP} / Bandwidth

- AI (DRAM) = FLOPS\textsubscript{DP} / Bandwidth (DRAM)
- AI (MCDRAM) = FLOPS\textsubscript{DP} / Bandwidth (MCDRAM)
- AI (L2) = FLOPS\textsubscript{DP} / Bandwidth (L2)
- AI (L1) = FLOPS\textsubscript{DP} / Bandwidth (L1)

- Performance = FLOPS\textsubscript{DP}
GFlop/s

- **GPP kernel on KNL:** 171.960 GFLOPS/sec
  - UOPS RETIRED_PACKED_SIMD
  - UOPS RETIRED_SCALAR_SIMD

- **likwid-perfctr -C 0-63 -g FLOPS_DP ./gpp.knl.exe 512 2 32768 20**
  - 8*UOPS RETIRED_PACKED_SIMD+UOPS RETIRED_SCALAR_SIMD

---

<table>
<thead>
<tr>
<th>Metric</th>
<th>Sum</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s] STAT</td>
<td>940.8064</td>
<td>14.7001</td>
<td>14.7001</td>
<td>14.7001</td>
</tr>
<tr>
<td>Clock [MHz] STAT</td>
<td>96000.0155</td>
<td>1499.9955</td>
<td>1500.0007</td>
<td>1500.0002</td>
</tr>
<tr>
<td>CPI STAT</td>
<td>86.0772</td>
<td>1.3396</td>
<td>1.5850</td>
<td>1.3450</td>
</tr>
<tr>
<td>DP MFLOP/s (SSE assumed) STAT</td>
<td>44456.2105</td>
<td>688.9334</td>
<td>729.9324</td>
<td>694.6283</td>
</tr>
<tr>
<td>DP MFLOP/s (AVX assumed) STAT</td>
<td>86957.6422</td>
<td>1347.4354</td>
<td>1429.2337</td>
<td>1358.7132</td>
</tr>
<tr>
<td>DP MFLOP/s (AVX512 assumed) STAT</td>
<td>171960.5065</td>
<td>2664.4393</td>
<td>2827.8362</td>
<td>2686.8829</td>
</tr>
<tr>
<td>Packed MUOPS/s STAT</td>
<td>21250.7162</td>
<td>329.2510</td>
<td>349.6506</td>
<td>332.0424</td>
</tr>
<tr>
<td>Scalar MUOPS/s STAT</td>
<td>1954.7786</td>
<td>30.4313</td>
<td>30.6312</td>
<td>30.5434</td>
</tr>
</tbody>
</table>
MCDRAM and DDR GB/s

- kernel on KNL: DDR 2.59GB/s + MCDRAM 63.71GB/s
  - MC_CAS_READS/MC_CAS_WRITES
  - EDC_RPQ_INSERTS/EDC_WPQ_INSERTS
  - EDC_MISS_CLEAN/EDC_MISS_DIRTY

- likwid-perfctr -C 0-63 -g HBM_CACHE ./gpp.knl.ex 512 2 32768 20

<table>
<thead>
<tr>
<th>Metric</th>
<th>Sum</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s] STAT</td>
<td>896.4352</td>
<td>14.0068</td>
<td>14.0068</td>
<td>14.0068</td>
</tr>
<tr>
<td>Clock [MHz] STAT</td>
<td>95979.5220</td>
<td>1499.6763</td>
<td>1499.6807</td>
<td>1499.6800</td>
</tr>
<tr>
<td>CPI STAT</td>
<td>83.4239</td>
<td>1.2985</td>
<td>1.5496</td>
<td>1.3035</td>
</tr>
<tr>
<td>MCDRAM Memory read bandwidth [MBytes/s] STAT</td>
<td>63246.3054</td>
<td>0</td>
<td>63246.3054</td>
<td>988.2235</td>
</tr>
<tr>
<td>MCDRAM Memory read data volume [GBytes] STAT</td>
<td>885.8769</td>
<td>0</td>
<td>885.8769</td>
<td>13.8418</td>
</tr>
<tr>
<td>MCDRAM Memory writeback bandwidth [MBytes/s] STAT</td>
<td>468.4857</td>
<td>0</td>
<td>468.4857</td>
<td>7.3201</td>
</tr>
<tr>
<td>MCDRAM Memory writeback data volume [GBytes] STAT</td>
<td>6.5620</td>
<td>0</td>
<td>6.5620</td>
<td>0.1025</td>
</tr>
<tr>
<td>MCDRAM Memory bandwidth [MBytes/s] STAT</td>
<td>63714.7910</td>
<td>0</td>
<td>63714.7910</td>
<td>995.5436</td>
</tr>
<tr>
<td>MCDRAM Memory data volume [GBytes] STAT</td>
<td>892.4389</td>
<td>0</td>
<td>892.4389</td>
<td>13.9444</td>
</tr>
<tr>
<td>DDR Memory read bandwidth [MBytes/s] STAT</td>
<td>2569.3065</td>
<td>0</td>
<td>2569.3065</td>
<td>40.1454</td>
</tr>
<tr>
<td>DDR Memory read data volume [GBytes] STAT</td>
<td>35.9877</td>
<td>0</td>
<td>35.9877</td>
<td>0.5623</td>
</tr>
<tr>
<td>DDR Memory writeback bandwidth [MBytes/s] STAT</td>
<td>21.1772</td>
<td>0</td>
<td>21.1772</td>
<td>0.3309</td>
</tr>
<tr>
<td>DDR Memory writeback data volume [GBytes] STAT</td>
<td>0.2966</td>
<td>0</td>
<td>0.2966</td>
<td>0.0046</td>
</tr>
<tr>
<td>DDR Memory bandwidth [MBytes/s] STAT</td>
<td>2590.4837</td>
<td>0</td>
<td>2590.4837</td>
<td>40.4763</td>
</tr>
<tr>
<td>DDR Memory data volume [GBytes] STAT</td>
<td>36.2843</td>
<td>0</td>
<td>36.2843</td>
<td>0.5669</td>
</tr>
</tbody>
</table>
L2 GB/s

- Kernel on KNL: L2 96.80GB/s
  - L2_REQUESTS_REFERENCE
  - OFFCORE_RESPONSE_0_OPTIONS
- likwid-perfctr -C 0-63 -g L2 ./gpp.knl.ex 512 2 32768 20

![Image of a table with various metrics and their statistics]
L1 GB/s

- Kernel on KNL: **L1 170.77GB/s**
  - MEM_UOPS RETIRED_ALL LOADS
  - MEM_UOPS RETIRED_ALL STORES

- likwid-perfctr -C 0-63 -g DATA ./gpp.knl.ex 512 2 32768 20
  - (MEM_UOPS RETIRED_ALL LOADS + MEM_UOPS RETIRED_ALL STORES)*64/runtime
  - -g DATA is for load-to-store ratio, but can be used to estimate L1 bandwidth (assume all loads are vector loads)
Resultant Roofline

- AI (DRAM): 66.39
- AI (MCDRAM): 2.70
- AI (L2): 1.78
- AI (L1): 1.01
- Performance: 171.960 GFLOPS/s
LIKWID on AMReX apps

- Used LIKWID to characterize AMReX applications
- Measured cache and DRAM bytes.
  - Averaged over 30min executions and 32 processes
  - Only 2 applications (not counting HPGMG proxy) used >50% of memory bandwidth on average
  - Used this data to estimate average AI for each level of the memory hierarchy
  - Used this data to infer requisite cache tapering
Likwid-mpirun

- `srun -n 2 -c 32 --cpu-bind=cores likwid-perfctr -C 0,8 -g MEM -o test_%h_%p_%r.txt ./xthi.x`
  - `%h` hostname
  - `%p` process ID
  - `%r` MPI rank

- `likwid-mpirun -pin S0:0,8_S1:0,8 -g MEM ./xthi.x`
  - `Hello from rank 0, thread 0, on nid00191. (core affinity = 0)`
  - `Hello from rank 0, thread 1, on nid00191. (core affinity = 8)`
  - `Hello from rank 1, thread 0, on nid00191. (core affinity = 16)`
  - `Hello from rank 1, thread 1, on nid00191. (core affinity = 24)`

- Uncore counters are measured on a per-socket basis
Marking Specific Regions

```c
#include <likwid.h>

LIKWID_MARKER_INIT;
#pragma omp parallel {
    LIKWID_MARKER_THREADINIT;
}
#pragma omp parallel {
    LIKWID_MARKER_START("foo");
    #pragma omp for
    for(i = 0; i < N; i++) {
        data[i] = omp_get_thread_num();
    }
    LIKWID_MARKER_STOP("foo");
}
LIKWID_MARKER_CLOSE;
```

- `cc -qopenmp -DLIKWID_PERFMON -I$LIKWID_INCLUDE -L$LIKWID_LIB -llikwid -dynamic test.c -o test.x`
- `likwid-perfctr -C 0-3 -g MEM -m ./test.x`

focus on specific code regions
FLOP Roofline vs. VUOP Roofline

- Nominally, Roofline is based on Flop/s, GB/s, and Flop/Byte
- Such metrics make sense from the user perspective.

- On SIMD machines, one might consider vuop/s instead of flop/s
  - vuop/s (scalar + vector) can easily be mapped to vector unit utilization
  - 100% vector unit utilization can bottleneck performance
  - Performance counters give vuop/s and not flop/s
  - 100% vector unit utilization does not imply 100% of peak (FMA, scalar vs. vector)
FLOP Roofline

- With performance counters alone, it's hard to deduce why performance is well-below the FLOP Roofline.
  - VL?
  - Precision?
  - FMA?
  - Masks?
  - Non-FP vector instructions
- Moreover, one might conclude a code is memory bound when in reality is compute-bound.
**VUOP Roofline**

- In a VUOP KNL Roofline
  - Machine peak (VUOP/s) is 16x lower
  - Machine balance is 16x lower (0.375)
- Consider an example where all flops are scalar adds (VADDSD)
  - 1 FLOP / VUOP
  - AI = 3 FLOPs/Byte = 3 VUOPS/Byte
- Although FLOP/s was far from its Roofline, VUOP/s is 16x closer to its peak

- Need source code analysis to understand VL, FMA, … issues
FLOP Roofline

- Use of FMA doesn’t change Arithmetic Intensity (FMA == FMUL+FADD == 2 FLOPs)
- Use of SIMD doesn’t change Arithmetic Intensity
- Presence of vector integer operations doesn’t change Arithmetic Intensity
- Moving from 64b to 32b data types doubles AI
- High fraction of Roofline implies high performance

VUOP Roofline

- Use of FMA cuts Arithmetic Intensity in half (half the number of VUOPS)
- Use of SIMD reduces Arithmetic Intensity by a factor of Vector Length (e.g. cuts number of VUOPS by 8x)
- Presence of vector integer operations increases Arithmetic Intensity
- Moving from 64b to 32b data types doesn’t change Arithmetic Intensity
- High fraction of Roofline implies high vector unit utilization (but not necessarily high performance)
Roofline with SDE
Why isn’t LIKWID good enough?

- LIKWID counts vector uops
- KNL vuop counters aren’t…
  - VL-aware
  - precision-aware
  - mask-aware
  - FMA-aware
- Counters don’t differentiate instruction types (FP, int, shuffle, …)
- **Flop counters were broken on Haswell.**
- Thus, LIKWID might be a good starting point, but it’s not perfect.

> Need tools that actually count flops correctly and ones that can be used to understand nuances of instruction mixes.
Intel Software Development Emulator (SDE)

- Dynamic instruction tracing
  - Accounts for actual loop lengths and branches
  - Counts instruction types, lengths, etc…
  - Can mark individual regions
  - Support for MPI+OpenMP
  - Can be used to calculate FLOPs (VL-, FMA-, and precision-aware)
  - Post processing can be expensive.
  - No insights into cache behavior or DRAM data movement
  - X86 only

Compiling with SDE at NERSC

- **Makefile...**

  ```
  MPICC = cc
  CFLAGS = -g -O3 -dynamic -qopenmp -restrict -qopt-streaming-stores always \ 
  -DSTREAM_ARRAY_SIZE=400000000 -DNTIMES=50 \ 
  -I$(VTUNE_AMPLIFIER_XE_2018_DIR)/include
  LDFLAGS = -L$(VTUNE_AMPLIFIER_XE_2018_DIR)/lib64 -littnotify
  
  stream_mpi.exe: stream_mpi.c Makefile
  $(MPICC) $(CFLAGS) stream_mpi.c -o stream_mpi.exe $(LDFLAGS)
  
  clean:
  rm -f stream_mpi.exe
  ```

- **module load sde**

- **make**

[https://bitbucket.org/dwdoerf/stream-ai-example.git](https://bitbucket.org/dwdoerf/stream-ai-example.git)
Running with SDE at NERSC

```
srun -n 4 -c 6 sde -ivb -d -iform 1 -omix my_mix.out -i -global_region -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat -- foo.exe
```

- `ivb` is used to target Edison's Ivy Bridge ISA (for Cori use `-hsw` for Haswell or `-knl` for KNL processors)
- `-d` specifies to only collect dynamic profile information
- `-iform 1` turns on compute ISA iform mix
- `-omix` specifies the output file (and turns on `-mix`)
- `-i` specifies that each process will have a unique file name based on process ID (needed for MPI)
- `-global_region` will include any threads spawned by a process (needed for OpenMP)

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/
When the job completes, you’ll have a series of files prefixed with “sde_”.

Parse the output to summarize the results...

```
$ ./parse-sde.sh sde_2p16t*
Search stanza is "EMIT_GLOBAL_DYNAMIC_STATS"
elements_fp_single_1 = 0
elements_fp_single_2 = 0
elements_fp_single_4 = 0
elements_fp_single_8 = 0
elements_fp_single_16 = 0
elements_fp_double_1 = 2960
elements_fp_double_2 = 0
elements_fp_double_4 = 999999360
elements_fp_double_8 = 0
---Total single-precision FLOPs = 0
---Total double-precision FLOPs = 4000000400

---Total FLOPs = 4000000400
mem-read-1 = 8618384
mem-read-2 = 1232
mem-read-4 = 137276433
mem-read-8 = 149329207
mem-read-16 = 1999998720
mem-read-32 = 0
mem-read-64 = 0
mem-write-1 = 264992
mem-write-2 = 560
mem-write-4 = 285974
mem-write-8 = 14508338
mem-write-16 = 0
mem-write-32 = 499999680
mem-write-64 = 0
---Total Bytes read = 33752339756
---Total Bytes written = 16117466472
---Total Bytes = 49860980228
```
Marking Regions of Interest for SDE

// Code must be built with appropriate paths for VTune include file (ittnotify.h) and library (-littnotify)
#include <ittnotify.h>

__SSC_MARK(0x111); // start SDE tracing, note it uses 2 underscores
__itt_resume();     // start VTune, again use 2 underscores
for (k=0; k<NTIMES; k++) {
    #pragma omp parallel for
    for (j=0; j<STREAM_ARRAY_SIZE; j++)
        a[j] = b[j]+scalar*c[j];
}

__itt_pause();      // stop VTune
__SSC_MARK(0x222); // stop SDE tracing

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/
LIKWID vs. SDE

- Recall, LIKWID counts vector uops while SDE counts instructions
- Why does this matter?
  - VL-aware: KNL has scalar but treats 128b, 256b, and 512b as 512b
  - precision-aware: User has to know which precision they use
  - mask-aware: KNL counters ignore masks
  - FMA-aware: LIKWID assumes 1 flop per element
  - KNL counts vector integer, stores, NT stores, and gathers as vector uops (and thus as potential flop/s)

- LIKWID’s and SDE’s counts of #FP ops and Gflop/s can be different (very different for linear algebra).
LIKWID vs. SDE/VTune

- **SDE FLOPS:**
  - `sde64 -knl -d -iform 1 -omix my_mix.out -global_region -- ./gpp.knl.ex 512 2 32768 20`
  - `./parse-sde.sh my_mix.out`
  - `--- > Total FLOPs = 2775769815463`

- **VTune Bytes:**
  - `amplxe-cl -collect memory-access -finalization-mode=deferred -r my_vtune/ -- ./gpp.knl.ex 512 2 32768 20`
  - `amplxe-cl -report summary -r my_vtune/ > my_vtune.summary`
  - `./parse-vtune.sh my_vtune.summary`
  - `DDR --- > Total Bytes = 35983553088`
  - `HBM --- > Total Bytes = 963486016448`

Roofline with LIKWID + SDE
Initially Cobbled Together Tools...

- Use tools known/observed to work on NERSC’s Cori (KNL, HSW)...
  - Used **Intel SDE** (Pin binary instrumentation + emulation) to create software Flop counters
  - Used **Intel VTune** performance tool (NERSC/Cray approved) to access uncore counters
- Accurate measurement of Flop’s (HSW) and DRAM data movement (HSW and KNL)
- Used by NESAP (NERSC KNL application readiness project) to characterize apps on Cori...

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/
More Recently…

- Use tools known/observed to work on NERSC’s Cori (KNL, HSW)…
  - Used Intel SDE (Pin binary instrumentation + emulation) to create software Flop counters
  - Used LIKWID performance counter tool (NERSC/Cray approved) to access uncore counters
- Accurate measurement of Flop’s (HSW) and DRAM data movement (HSW and KNL)
- Used by NESAP (NERSC KNL application readiness project) to characterize apps on Cori…

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/
Hierarchical Roofline vs. Cache-Aware Roofline

...understanding different Roofline formulations in Intel Advisor
There are two Major Roofline Formulations:

- **Hierarchical Roofline (original Roofline w/ DRAM, L3, L2, …)**
  - Chapter 4 of “Auto-tuning Performance on Multicore Computers”, 2008
  - Defines multiple bandwidth ceilings and multiple AI’s per kernel
  - Performance bound is the minimum of flops and the memory intercepts (superposition of original, single-metric Rooflines)

- **Cache-Aware Roofline**
  - Defines multiple bandwidth ceilings, but uses a single AI (flop:L1 bytes)
  - As one loses cache locality (capacity, conflict, …) performance falls from one BW ceiling to a lower one at constant AI

- **Why Does this matter?**
  - Some tools use the Hierarchical Roofline, some use cache-aware == Users need to understand the differences
  - Cache-Aware Roofline model was integrated into production Intel Advisor
  - Evaluation version of Hierarchical Roofline\(^1\) (cache simulator) has also been integrated into Intel Advisor

\(^1\)Experimental Feature, the look and feel and exact behavior is subject for change
Hierarchical Roofline

- Captures cache effects
- AI is Flop:Bytes after being *filtered by lower cache levels*
- Multiple Arithmetic Intensities (one per level of memory)
- AI *dependent* on problem size (capacity misses reduce AI)
- Memory/Cache/Locality effects are *observed as decreased AI*
- Requires *performance counters or cache simulator* to correctly measure AI

Cache-Aware Roofline

- Captures cache effects
- AI is Flop:Bytes *as presented to the L1 cache (plus non-temporal stores)*
- Single Arithmetic Intensity
- AI *independent* of problem size
- Memory/Cache/Locality effects are *observed as decreased performance*
- Requires static analysis or *binary instrumentation* to measure AI
Example: STREAM

- L1 AI...
  - 2 flops
  - 2 x 8B load (old)
  - 1 x 8B store (new)
  - = 0.08 flops per byte

- No cache reuse...
  - Iteration i doesn’t touch any data associated with iteration i+delta for any delta.

- … leads to a DRAM AI equal to the L1 AI

```c
#pragma omp parallel for
for(i=0;i<N;i++){
    Z[i] = X[i] + alpha*Y[i];
}
```
Example: STREAM

Hierarchical Roofline

Cache-Aware Roofline

Performance is bound to the minimum of the two intercepts...

Multiple AI's...
1) Flop:DRAM bytes
2) Flop:L1 bytes (same)

Observed performance is correlated with DRAM bandwidth

Single AI based on flop:L1 bytes
Example: 7-point Stencil (Small Problem)

- **L1 AI...**
  - 7 flops
  - 7 x 8B load (old)
  - 1 x 8B store (new)
  - \( = 0.11 \) flops per byte
  - some compilers may do register shuffles to reduce the number of loads.

- **Moderate cache reuse...**
  - \( \text{old}[k][j][i+1] \) is reused on next iteration of \( i \).
  - \( \text{old}[k][j+1][i] \) is reused on next iteration of \( j \).
  - \( \text{old}[k+1][j][i] \) is reused on next iterations of \( k \).

- **... leads to DRAM AI larger than the L1 AI**
Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

Multiple AI’s:
1) flop:DRAM ~ 0.44
2) flop:L1 ~ 0.11

Performance bound is the minimum of the two

Cache-Aware Roofline

Arithmetic Intensity (Flop:Byte)
Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

Cache-Aware Roofline

- Arithmetic Intensity (Flop:Byte)
- Attainable Flop/s
- DRAM GB/s
- L1 GB/s

Performance bound is the minimum of the two

Multiple AI's....
1) flop:DRAM $\approx 0.44$
2) flop:L1 $\approx 0.11$

Observed performance is between L1 and DRAM lines (== some cache locality)

Single AI based on flop:L1 bytes
Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

Cache-Aware Roofline

<table>
<thead>
<tr>
<th>DRAM GB/s</th>
<th>Peak Flop/s</th>
<th>Attainable Flop/s</th>
<th>Arithmetic Intensity (Flop:Byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.11</td>
<td></td>
<td></td>
<td>0.11</td>
</tr>
<tr>
<td>0.20</td>
<td></td>
<td></td>
<td>0.20</td>
</tr>
</tbody>
</table>

Multiple AI’s:
1) flop:DRAM ~ 0.20
2) flop:L1 ~ 0.11

Capacity misses reduce DRAM AI and performance.

Observed performance is closer to DRAM line (less cache locality).

Single AI based on flop:L1 bytes.
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

- Attainable Flop/s vs. Arithmetic Intensity (Flop:Byte)
- L1 GB/s
- Actual observed performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Cache-Aware Roofline

- Attainable Flop/s vs. Arithmetic Intensity (Flop:Byte)
- L1 GB/s
- Observed performance is closer to DRAM line (== less cache locality)
- Single AI based on flop:L1 bytes
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

Cache-Aware Roofline

Actual observed performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes
Roofline with Intel® Advisor

slides from Zakhar Matveev (intel)
Intel Advisor

- Includes Roofline Automation…
  - Automatically instruments applications (one dot per loop nest/function)
  - Computes FLOPS and AI for each function (**CARM**)
  - AVX-512 support that incorporates masks
  - **Integrated Cache Simulator**\(^1\) (hierarchical roofline / multiple AI’s)
  - Automatically benchmarks target system (calculates ceilings)
  - Full integration with existing Advisor capabilities

http://www.nersc.gov/users/training/events/roofline-training-1182017-1192017

\(^1\)Experimental Feature, the look and feel and exact behavior is subject for change
Intel® Advisor: Components

Step 1. Compiler diagnostics + Performance Data + SIMD efficiency information
Guidance: detect problem and recommend how to fix it

Step 2. “Precise” Trip Counts & FLOPs. Roofline analysis.
Characterize your application.

Step 3. Loop Carried Dependency Analysis

Step 4. Memory Access Patterns Analysis
- Roofline for INT OP/S
- Integrated Roofline (exp)
- Interactive(!) HTML export
- MAC OS viewer
- Function call counts
- Python API

What's new in “2019” release
### Intel® Advisor: 2-pass Approach

<table>
<thead>
<tr>
<th>Roofline:</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Axis (AI): #FLOPs / #Bytes</td>
<td></td>
</tr>
<tr>
<td>Y-Axis (FLOP/s): #FLOP(mask-aware)/time</td>
<td></td>
</tr>
</tbody>
</table>

**Step 1: Survey** ([collect survey](#))
- Records run times
- User-mode sampling; non-intrusive
- **No need for root access**

**Step 2: FLOPs** ([collect tripcounts](#)-flops)
- Record #FLOPs, #Bytes, AVX512 masks
- Precise, instrumentation-based count of the number of instructions
- **No need for root access**

---

1. With Integrated Roofline (Cache Simulator) enabled.
Each Ceiling provides peak CPU/Memory throughput of your PLATFORM (benchmarked)

Each Dot represents loop or function in YOUR APPLICATION (profiled)

Automatic and integrated – first class citizen in Intel® Advisor
NEW: Integrated Roofline

CARM (L1+NTS) CPU perspective

Highly optimized

All hotspots but 1 are not CPU-bound

Full waveform Inversion. Seismic Workload

DRAM (ORM)

Not Memory bound

Some Locality

Data: Courtesy Philippe Thierry
NEW: Integer, Float, Int+Float Rooflines
NEW: Memory Traffic in Survey Grid
Integrated Roofline Model

Old Approach…

source advixe-vars.sh
advixe-cl -collect survey --project-dir ./your_project -- <your-executable-with-parameters>
advixe-cl -collect tripcounts -enable-cache-simulation -flop --project-dir ./your_project -- <your-executable-with-parameters>

New Approach (but not compatible with MPI)…

source advixe-vars.sh
advixe-cl -collect roofline -enable-cache-simulation --project-dir ./your_project -- <your-executable-with-parameters>

(optional) copy data to your UI desktop system
advixe-gui ./your_project

Advisor on NERSC’s Cori

- [Link to Advisor](http://www.nersc.gov/users/software/performance-and-debugging-tools/advisor/)

```bash
module load advisor/2018.integrated_roofline
cc -g -dynamic -openmp -O2 -o mycode.exe mycode.c
```

- Best to run advisor only on rank 0... `srun` calls a script like...

```bash
#!/bin/bash
if [[ $SLURM_PROCID == 0 ]]; then
 advixe-cl -collect=survey --project-dir knl-result -data-limit=0 -- ./a.out
else
 sleep 30
 ./a.out
fi
```
Exporting Roofline Figures

- Advisor can directly export a HTML Roofline figure …

- Alternately, you can output directly from the command line (no GUI needed)…

  advixe-cl -report roofline --project-dir ./your_project > roofline.html
Questions?
Summary

In this talk, we discussed several approaches to constructing Rooflines on CPUs…

- Machine Characterization
- Using LIKWID to access performance counters
- Using SDE to get more accurate FLOP counts
- Using Advisor to provide a single tool that integrates cache simulation and accurate FLOP counts.
Backup