Performance Analysis and Optimization

Samuel Williams

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

- Application-specific (manual) instrumentation…
  - Most robust
  - Minimal overhead (omp_get_time)
  - Insensitive to sampling effects
  - Application-specific knowledge can differ based on usage (e.g. different levels of MG)
  - High effort / large reward

- Auto-instrumentation (TAU, Advisor, Vtune)…
  - Minimal effort
  - Integrated visualization
  - Sampling effects can confuse performance analysis
  - Using the same function many different ways can confuse analysis
  - Can have high overhead (Advisor/Vtune)
The **Roofline Model** is a throughput-oriented performance model...

- Tracks rates not time
- Augmented with Little’s Law
  
  \( \text{concurrency} = \text{latency} \times \text{bandwidth} \)
- Independent of ISA and architecture
  
  (applies to CPUs, GPUs, Google TPUs\(^1\), etc…)

- Informs developers which routines are underperforming the processor’s capabilities == which routines to optimize

---

\(^1\) Jouppi et al, “In-Datacenter Performance Analysis of a Tensor Processing Unit”, ISCA, 2017.
Use by NESAP

- NESAP is the NERSC KNL application readiness project.
- NESAP used Roofline to drive optimization and analysis on KNL
  - Bound performance expectations (ERT)
  - Use Vtune to quantify DDR and MCDRAM data movement
  - Compare KNL data movement to Haswell (sea of private/coherent L2’s vs. unified L3)
  - Understand importance of vectorization

Roofline for NESAP Codes

**MFDn**

- Roofline Model
- wo/FMA
- 1 RHS
- 4 RHS
- 8 RHS

**EMGeo**

- Roofline Model
- wo/FMA
- Original
- SELL
- SB
- SELL+SB
- nRHS+SELL+SB

**PICSAR**

- Roofline Model
- wo/FMA
- Original
- w/Tiling
- w/Tiling+Vect

---

Arithmetic Intensity (FLOP/byte)

GFLOP/s
Intel Advisor

Intel Advisor is a performance analysis tool (evolved from vector advisor)

Background

- https://www.youtube.com/watch?v=h2QEM1HpFgg

Running Advisor on NERSC Systems

<table>
<thead>
<tr>
<th>Source</th>
<th>Lin.</th>
<th>Code</th>
<th>Description</th>
<th>Total Time</th>
<th>% Loop/Function Time</th>
<th>% Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>stencil_v2.c:29</td>
<td>25</td>
<td>#pragma omp parallel for</td>
<td>for(k=1;k&lt;dim+1;k++)</td>
<td>9.890s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>26</td>
<td></td>
<td>for(j=1;j&lt;dim+1;j++)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td></td>
<td>#pragma novector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>for(i=1;i&lt;dim+1;i++)</td>
<td>102.403s</td>
<td>157.994s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>int ijk = i<em>iStride + j</em>jStride + k*kStride;</td>
<td></td>
<td>53.651s</td>
<td></td>
<td>FMA</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>new[ijk] = -6.0*old[ijk]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>+ old[ijk*kStride]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Selected (Total Time): 102.403s
likwid

- Performance counter infrastructure for x86
- No sampling (simply reads counters)
- <1% overhead (SDE/VTune/Advisor see >10x)
- MPI Scalable (can run across many processes)
- Useful for high-level characterization (not connected to source)

Now runs on NERSC machines:
- Verified on Haswell
- Needs Vtune and likwid modules
- #SBATCH –perf=vtune
- Likwid-perfctr
- Used to characterize AMReX ECP apps...
Threading vs. Processes

- Threads provide no inherent compute advantage over processes
- Threads incur additional overhead (omp parallel, single, barrier, ...) \(\Rightarrow\) slower in the perfectly parallel world
- Threads provide easy access to shared memory...
  - Some codes are easier to parallelize with threads than MPI
  - Easier to avoid data duplication (memory requirements) with threads than processes
  - Threads can access shared data in cache rather than copying data between processes
  - Using threads simplifies topology-aware MPI process mapping (MPICH_RANK_REORDER is often insufficient)... topology-aware is still important on Aries/Dragonfly and IB/FatTrees
  - Using threads provides on-ramp to GPUs and other emerging architectures
Threading Experiments/Analysis

- Fix process concurrency, increase threading …
  - Ideally, function runtime should scales as $O(1/\text{omp\_num\_threads})$
  - Identify functions that plateau (saturation)
  - Identify functions that are flat (sequential bottlenecks)
  - Identify functions that increase (duplicated work… e.g. use of Fortran’s sum(a(:)))

- Fix hardware concurrency (cores), increase threads while reducing processes…. (32x1, 16x2, 8x4, 4x8, 2x16, 1x32)…
  - Distinguish true flat MPI (no –fopenmp) from hybrid w/1 thread (-fopenmp + OMP_NUM_THREADS=1) == threading overhead
  - Identify functions that are flat, better with threads, or better with processes
  - Especially useful for communication routines
KNL-specific issues

- GNU and Intel runtimes treat OMP_PLACES/PROC_BIND differently...
  - use same compiler for all threaded routines / know which settings to use
  - (I use KMP_AFFINITY with Intel)

- If you can fit in 16GB, run in quadflat...
  - avoids cache aliasing issues where some nodes are slower than others
  - Can manifest as abnormally high MPI_Wait times and degraded scalability
  - Use `srun` ... `numactl` `–m1` `./a.out` ... (or use `–p1` or no `numactl` but allocate key data in HBM)

- use large contiguous blocks with 2M pages for data and MPI buffers...
  - Better performance for complex (less streaming) memory access patterns
  - minimizes NIC TLB pressure == higher MPI bandwidth

- Examine vectorization reports...
  - KNL is very slow if not vectorized
Questions