Using the Roofline Model and Intel Advisor

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

Tuomas Koskela
TKoskela@lbl.gov
NERSC
Lawrence Berkeley National Lab
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 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.
- This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

- Special Thanks to:
  - Zakhar Matveev, Intel Corporation
  - Roman Belenov, Intel Corporation
Introduction
Performance Models and Tools

- Identify performance bottlenecks
- Motivate software optimizations
- **Determine when we’re done optimizing**
  - Assess performance relative to machine capabilities
  - Motivate need for algorithmic changes
- **Predict performance on future machines / architectures**
  - Sets realistic expectations on performance for future procurements
  - Used for HW/SW Co-Design to ensure future architectures are well-suited for the computational needs of today’s applications.
Performance Models / Simulators

- Historically, many performance models and simulators tracked latencies to predict performance (i.e. counting cycles)

- The last two decades saw a number of latency-hiding techniques...
  - Out-of-order execution (hardware discovers parallelism to hide latency)
  - HW stream prefetching (hardware speculatively loads data)
  - Massive thread parallelism (independent threads satisfy the latency-bandwidth product)

- Effectively latency hiding has resulted in a shift from a latency-limited computing regime to a throughput-limited computing regime
The **Roofline Model** is a throughput-oriented performance model...

- Tracks rates not time
- Augmented with Little’s Law (concurrency = latency*bandwidth)
- Independent of ISA and architecture (applies to CPUs, GPUs, Google TPUs, etc…)

Three main components:

- Machine Characterization (realistic performance potential of the system)
- Monitoring (characterize application’s execution)
- Application Models (how well could my kernel perform with perfect compilers, procs, …)

https://crd.lbl.gov/departments/computer-science/PAR/research/roofline

---

Ideally, we could always attain peak Flop/s.

However, finite locality (reuse) limits performance.

Plot the performance bound using Arithmetic Intensity (AI) as the x-axis...

- Perf Bound = min (peak Flop/s, peak GB/s * AI)
- AI = Flops / Bytes presented to DRAM
- Log-log makes it easy to doodle, extrapolate performance, etc...
- Kernels with AI less than machine balance are ultimately memory bound.
Roofline Examples

- Typical machine balance is 5-10 flops per byte…
  - 40-80 flops per double to exploit compute capability
  - Artifact of technology and money
  - Unlikely to improve

- Consider STREAM Triad…
  ```c
  #pragma omp parallel for
  for(i=0;i<N;i++){
    z[i] = X[i] + alpha*Y[i];
  }
  ```
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - AI = 0.166 flops per byte == Memory bound
§ Conversely, 7-point constant coefficient stencil...

- 7 flops
- 8 memory references (7 reads, 1 store) per point
- Cache can filter all but 1 read and 1 write per point
- AI = 0.43 flops per byte == memory bound, but 3x the flop rate

```c
#pragma omp parallel for
for(k=1;k<dim+1;k++){
  for(j=1;j<dim+1;j++){
    for(i=1;i<dim+1;i++){
      int ijk = i + j*jstride + k*kstride;
      new[ijk] = -6.0*old[ijk] +
                old[ijk-1] +
                old[ijk+1] +
                old[ijk-jstride] +
                old[ijk+jstride] +
                old[ijk-kstride] +
                old[ijk+kstride];
    }
  }
}
```
Real processors have multiple levels of memory

- Registers
- L1, L2, L3 cache
- MCDRAM/HBM (KNL/GPU device memory)
- DDR (main memory)
- NVRAM (non-volatile memory)

We may measure a bandwidth and define an AI for each level

- A given application / kernel / loop nest will thus have multiple AI’s
- A kernel could be DDR-limited…
Hierarchical Roofline

- Real processors have multiple levels of memory
  - Registers
  - L1, L2, L3 cache
  - MCDRAM/HBM (KNL/GPU device memory)
  - DDR (main memory)
  - NVRAM (non-volatile memory)

- We may measure a bandwidth and define an AI for each level
  - A given application / kernel / loop nest will thus have multiple AI’s
  - A kernel could be DDR-limited…
  - or MCDRAM-limited depending on relative bandwidths and AI’s
Data, Instruction, Thread-Level Parallelism…

- We have assumed one can attain peak flops with high locality.
- In reality, this is premised on sufficient...
  - Use special instructions (e.g. fused multiply-add)
  - Vectorization (16 flops per instruction)
  - unrolling, out-of-order execution (hide FPU latency)
  - OpenMP across multiple cores
- Without these, …
  - Peak performance is not attainable
  - Some kernels can transition from memory-bound to compute-bound
  - n.b. in reality, DRAM bandwidth is often tied to DLP and TLP (single core can't saturate BW w/scalar code)
Roofline using ERT, VTune, and SDE
Basic Roofline Modeling

**Machine Characterization**

*Potential of my target system*

- How does my system respond to a lack of FMA, DLP, ILP, TLP?
- How does my system respond to reduced AI (i.e. memory/cache bandwidth)?
- How does my system respond to NUMA, strided, or random memory access patterns?
- ...

**Application Instrumentation**

*Properties of my app’s execution*

- What is my app’s real AI?
- How does AI vary with memory level?
- How well does my app vectorize?
- Does my app use FMA?
- ...

How Fast is My Target System?

- **Challenges:**
  - Too many systems; new ones each year
  - Voluminous documentation on each
  - Real performance often less than "Marketing Numbers"
  - Compilers can “give up” on big loops

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

- [https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/](https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/)
Application Instrumentation Can Be Hard…

- Flop counters can be broken/missing in production HW (Haswell)
- Counting Loads and Stores is a poor proxy for data movement as they don’t capture reuse
- Counting L1 misses is a poor proxy for data movement as they don’t account for HW prefetching.
- DRAM counters are accurate, but are privileged and thus nominally inaccessible in user mode
- OS/kernel changes must be approved by vendor (e.g. Cray) and the center (e.g. NERSC)
Application Instrumentation

- NERSC/CRD (==NESAP/SUPER) collaboration...
  - Characterize applications running on NERSC production systems
  - Use Intel SDE (binary instrumentation) to create software Flop counters (could use Byfl as well)
  - Use Intel VTune performance tool (NERSC/Cray approved) to access uncore counters
  - Produced accurate measurement of Flop’s and DRAM data movement on HSW and KNL

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/
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)
  - 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
  - nRHS+SELL+SB
Need a integrated solution…

- Having to compose VTune, SDE, and graphing tools worked correctly and benefitted NESAP, but …
- …placed a very high burden on users…
  - forced to learn/run multiple tools
  - forced to instrument each routine in their application
  - forced to manually parse/compose/graph the output
- …still lacked integration with compiler/debugger/disassembly

- CRD/NERSC wanted a more integrated solution…
Break / Questions
Roofline vs. “Cache-Aware” Roofline
There are two Major Roofline Formulations:

- **Original / DRAM / Hierarchical Roofline…**
  - Defines multiple bandwidth ceilings and multiple AI’s per kernel
  - Performance bound is the minimum of the intercepts and flops

- **“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 original Roofline, some use cache-aware == Users need to understand the differences
  - Intel Advisor uses the Cache-Aware Roofline Model (alpha/experimental DRAM Roofline being evaluated)
  - CRD/NERSC prefer the hierarchical Roofline as it provides greater insights into the behavior of the memory hierarchy
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 directly observed
- Requires performance counters to measure AI

“Cache-Aware” Roofline

- Captures cache effects
- AI is Flop:Bytes as presented to the L1 cache
- Single Arithmetic Intensity
- AI independent of problem size
- Memory/Cache/Locality effects are indirectly observed
- 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
**Example: STREAM**

**Roofline**

- **Peak Flop/s**
- **Attainable Flop/s**
- **Arithmetic Intensity (Flop:Byte)**

**“Cache-Aware” Roofline**

- **Peak Flop/s**
- **Attainable Flop/s**

Multiple AI’s:
1. based on flop:DRAM bytes
2. Based on flop:L1 bytes (same)

Actual Performance is the minimum of the two intercepts.

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...**
  - old[ijk] is reused on subsequent iterations of i,j,k
  - old[ijk-1] is reused on subsequent iterations of i.
  - old[ijk-jStride] is reused on subsequent iterations of j.
  - old[ijk-kStride] is reused on subsequent iterations of k.

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

**Roofline**

- Peak Flop/s
- Attainable Flop/s
- Arithmetic Intensity (Flop:Byte)

**“Cache-Aware” Roofline**

- Observed performance is between L1 and DRAM lines (≈ some cache locality)
- Single AI based on flop:L1 bytes

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

Actual Performance is the minimum of the two
Example: 7-point Stencil (Large Problem)

Roofline

“Cache-Aware” Roofline

- Capacity misses reduce DRAM AI and performance
- Multiple AI’s:
  1) flop:DRAM ~ 0.20
  2) flop:L1 ~ 0.11

- Observed performance is closer to DRAM line (less cache locality)
- Single AI based on flop:L1 bytes
Break / Questions
Intel Advisor: Introduction and General Usage

*DRAM Roofline and OS/X Advisor GUI: These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.
Intel Advisor

- Integrated Performance Analysis Tool
  - Performance information including timings, flops, and trip counts
  - Vectorization Tips
  - Memory footprint analysis
  - Uses the Cache-Aware Roofline Model
  - All connected back to source code

- CRD/NERSC began a collaboration with Intel
  - Ensure Advisor runs on Cori in user-mode
  - Push for Hierarchical Roofline
  - Make it functional/scalable to many MPI processes across multiple nodes
  - Validate results on NESAP, SciDAC, and ECP codes
Intel Advisor (Useful Links)

Background

- [https://www.youtube.com/watch?v=h2QEM1HpFgg](https://www.youtube.com/watch?v=h2QEM1HpFgg)

Running Advisor on NERSC Systems

Using Intel Advisor at NERSC

- **Compile...**
  
  use `-g` when compiling

- **Submit Job...**
  
  `% salloc -perf=vtune`  
  `--perf` only needed for DRAM Roofline
  
  `-or-`
  
  `#SBATCH -perf=vtune`  
  `--perf` only needed for DRAM Roofline

- **Benchmark...**
  
  `% module load advisor`
  
  `% export ADVIXE_EXPERIMENTAL=roofline_ex`  
  `--only needed for DRAM Roofline`
  
  `% srun [args] advixe-cl -collect survey -no-stack-stitching -project-dir $DIR -- ./a.out [args]`
  

- **Use Advisor GUI...**
  
  `% module load advisor`
  
  `% export ADVIXE_EXPERIMENTAL=roofline_ex`  
  `--only needed for DRAM Roofline`
  
  `% advixe-gui $DIR`
Welcome to Intel Advisor 2017
Vectorization Optimization and Thread Prototyping

Current project: advi.stencil.aug2.16

- Show My Result
- Configure Project...

Recent Projects:
- advi.dram.stencil.aug2.16
- New Project...
- Open Project...
- Open Result
Program metrics

Elapsed Time: 50.50s
Vector Instruction Set: AVX
Number of CPU Threads: 16
Total GFLOP Count: 753.95
Total GFLOPS: 14.93
Total Arithmetic Intensity: 0.12

Loop metrics

Total CPU time: 806.22s
Time in 5 vectorized loops: 641.62s
Time in scalar code: 164.60s

Vectorization Gain/Efficiency
Vectorized Loops Gain/Efficiency: 3.81x (95%)
Program Approximate Gain: 3.23x

Top time-consuming loops

<table>
<thead>
<tr>
<th>Loop</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Trip Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in bench_stencil_ver2$omp$parallel_for@102 at stencil_v2.c:108]</td>
<td>160.035s</td>
<td>160.035s</td>
<td>31; 3; 2; 3</td>
</tr>
<tr>
<td>[loop in bench_stencil_ver3$omp$parallel_for@146 at stencil_v2.c:152]</td>
<td>159.953s</td>
<td>159.953s</td>
<td>32; 2</td>
</tr>
<tr>
<td>[loop in bench_stencil_ver4$omp$parallel_for@193 at stencil_v2.c:201]</td>
<td>159.595s</td>
<td>159.595s</td>
<td>130</td>
</tr>
<tr>
<td>[loop in bench_stencil_ver1$omp$parallel_for@62 at stencil_v2.c:63]</td>
<td>159.307s</td>
<td>159.307s</td>
<td>31; 3; 2; 3</td>
</tr>
</tbody>
</table>
Loop in `bench_stencil_ver0$omp$parallel_for... at stencil_v2.c:29`

Average Trip Counts: 512

Instruction Mix:
- Memory: 5
- Compute: 9
- Mixed: 4
- Other: 4
- Number of Vector Registers: 9

GFLOPS: 9.00873

Code Optimizations
- Compiler: Intel(R) C Intel(R) 64
- Compiler for applications running on Intel(R) 64,
  Version: 17.0.2.174 Build 20170213
Performance: 116.2 GFLOPS
Arithmetic Intensity: 0.16 FLOP/Byte
Break / Questions
Intel Advisor:
Stencil Roofline Demo*

*DRAM Roofline and OS/X Advisor GUI: These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.
7-point, Constant-Coefficient Stencil

- Apply to a $512^3$ domain on a single NUMA node (single HSW socket)
- Create 5 code variants to highlight effects (as seen in advisor)

ver0. Baseline: thread over outer loop (k), but prevent vectorization

```c
#pragma novector // prevent simd
int ijk = i*iStride + j*jStride + k*kStride; // variable iStride to confuse the compiler
```

ver1. Enable vectorization

```c
int ijk = i + j*jStride + k*kStride; // unit-stride inner loop
```

ver2. Eliminate capacity misses

```c
2D tiling of j-k iteration space // working set had been $O(6MB)$ per thread
```

ver3. Improve vectorization

`Provide aligned pointers and strides`

ver4. Force vectorization / cache bypass

```c
__assume(jstride%8 == 0); // stride by variable is still aligned
#pragma omp simd, vector nontemportal // force simd; force cache bypass
```
<table>
<thead>
<tr>
<th>Line</th>
<th>Source</th>
<th>Total Time</th>
<th>%</th>
<th>Loop/Function Time</th>
<th>%</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>#pragma omp parallel for</td>
<td>9.890s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>for(k=1;k&lt;dim1;k++){</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>for(j=1;j&lt;dim1;j++){</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>#pragma novector</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>for(i=1;i&lt;dim1;i++){</td>
<td>102.403s</td>
<td>100%</td>
<td>157.994s</td>
<td>158%</td>
<td>FMA</td>
</tr>
<tr>
<td>30</td>
<td>int ijk = i<em>iStride + j</em>jStride + k*kStride;</td>
<td>53.651s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>new[ijk] = -6.0*old[ijk]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>+ old[ijk+1Stride]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cache-Aware Roofline

Performance (GFLOPS)

L1 Bandwidth: 5516.07 GB/sec
L2 Bandwidth: 1801.27 GB/sec
L3 Bandwidth: 502.16 GB/sec
DRAM Bandwidth: 128.85 GB/sec

Self Elapsed Time: 10.144 s  Total Time: 159.953 s

File: stencil_v2.c:152  benchStencilVer3$omp$parallel_for@146 stencil_v2.c:152
Performance: 16.27 GFLOPS
L1 Arithmetic Intensity: 0.12 FLOP/Byte
Self Elapsed Time: 10.144 s  Total Time: 159.953 s

Lin. Source Total Time Loop/Function Time % Traits
151 for(j=Lo;j<Lo+16;j++)
  0.016s
152 for(i=0;i<Stride;i++)
  0.708s  159.953s

Selected (Total Time): 0.708s
Cache-Aware Roofline

Performance (GFLOPS)

Self Elapsed Time: 10.004 s  Total Time: 159.595 s

L1 Bandwidth: 5516.07 GB/sec
L2 Bandwidth: 1801.27 GB/sec
L3 Bandwidth: 502.16 GB/sec
DRAM Bandwidth: 128.85 GB/sec

DP Vector FMA Peak: 843.06 GFLOPS
DP Vector Add Peak: 210.95 GFLOPS

File: stencil_v2.c:201 bench_stencil_ver4$omp$parallel_for@193 stencil_v2.c:201
Performance: 23.08 GFLOPS
L1 Arithmetic Intensity: 0.12 FLOP/Byte
Self Elapsed Time: 10.004 s
Total Time: 159.595 s
DRAM Roofline*

Performance (GFLOPS)
1688.18

Self Elapsed Time: 0.000 s  Total Elapsed Time: 10.000 s

Source: stencil_v2.c:26  bench_stencil_ver0$omp$parallel_for@25

<table>
<thead>
<tr>
<th>Line</th>
<th>Source</th>
<th>Total Time</th>
<th>%</th>
<th>Loop/Function Time</th>
<th>%</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>159417.000ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[loop in bench_stencil_ver0$omp$parallel_for@25 at stencil_v2.c:26]
DRAM Roofline*

Performance (GFLOPS)
1688.18

Self Elapsed Time: 0.000 s
Total Elapsed Time: 0.000 s

[loop in stencil_v2.c:63 bench_stencil_ver1$omp$parallel_for@62 at stencil_v2.c:63]
Total Performance: 10.18 GFLOPS
Total L1 Arithmetic Intensity: 0.17 FLOP/Byte
Self Elapsed Time: 0.000 s
Total Elapsed Time: 9.972 s

File: stencil_v2.c:63

<table>
<thead>
<tr>
<th>Line</th>
<th>Source</th>
<th>Total Time</th>
<th>Loop/Function Time</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>while (ElapsedTime &lt; TIME)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>#pragma omp parallel for</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>for (k = 1; k &lt; dim + 1; k++)</td>
<td>159453.000ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>for (j = 1; j &lt; dim + 1; j++)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Selected (Total Time): 0ms
DRAM Roofline*
DRAM Roofline*
DRAM Roofline

Performance (GFLOPS)

1688.18

L1 Bandwidth: 1.1e+4 GB/sec
L2 Bandwidth: 3542.01 GB/sec
L3 Bandwidth: 1093.46 GB/sec

DRAM Bandwidth: 128.58 GB/sec

Self Elapsed Time: 0.000 s
Total Elapsed Time: 9.956 s

Total Performance: 18.98 GFLOPS
Total L1 Arithmetic Intensity: 0.41 FLOP/Byte
Self Elapsed Time: 0.000 s
Total Elapsed Time: 9.956 s

File: stencil_v2.c:193 bench_stencil_ver4$omp$parallel_for@193
Lin. Source
191 StartTime = omp_get_wtime();
192 while(ElapsedTime < TIME){
193 #pragma omp parallel for schedule(static,1)
194 for(tile=0;tile<Tiles*tiles;tile++)

Selected (Total Time): 8.004 ms
Wrap up / Questions
Roofline/Advisor Tutorial at SC’17

- Sunday November 12th
- 8:30am-12pm (half-day tutorial)
- multi-/manycore focus
Intel Advisor (Useful Links)

Background

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

Running Advisor on NERSC Systems

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 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.
- This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

- Special Thanks to:
  - Zakhar Matveev, Intel Corporation
  - Roman Belenov, Intel Corporation