The Current and Future of Roofline

Charlene Yang
Application Performance Specialist
NERSC, LBNL
# The Roofline Chronical

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Developed foundations</strong> for the Roofline Model</td>
<td><strong>Developed performance counter Rooflines for CPUs and GPUs</strong></td>
<td><strong>Collaboration with CRD, Intel and NVIDIA on hierarchical Roofline</strong></td>
<td><strong>FPGAs, CGRAs, AI processors, ...</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Applied to kernels using canonical flops and bytes</strong></td>
<td><strong>Roofline for Simulations and Machine Learning</strong></td>
<td><strong>Incorporated VPU%, divides, integer operations</strong></td>
<td><strong>Asymmetric memory hierarchies</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Created the ERT prototype for CPUs and GPUs</strong></td>
<td><strong>Roofline model incorporated into Intel Advisor</strong></td>
<td><strong>Effects of extreme heterogeneity</strong></td>
<td><strong>Horizontal data movement</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Quantified CUDA UVM effects</strong></td>
<td><strong>Installed at NERSC, LANL, etc</strong></td>
<td><strong>Rooflines that serialize data transfers (vs. assume overlap)</strong></td>
<td><strong>Integration with compilers/runtimes</strong></td>
<td></td>
</tr>
</tbody>
</table>

## Prototype

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Created the ERT prototype for CPUs and GPUs</strong></td>
</tr>
<tr>
<td><strong>Quantified CUDA UVM effects</strong></td>
</tr>
</tbody>
</table>

## Production

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Roofline model incorporated into Intel Advisor</strong></td>
</tr>
<tr>
<td><strong>Installed at NERSC, LANL, etc</strong></td>
</tr>
</tbody>
</table>

## Future

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Roofline for GPUs (multiple vendors)</strong></td>
</tr>
<tr>
<td><strong>Roofline for FPGAs/CGRAs</strong></td>
</tr>
<tr>
<td><strong>Integer/instruction/non-FP Rooflines</strong></td>
</tr>
<tr>
<td><strong>CISC/DL instructions</strong></td>
</tr>
</tbody>
</table>
The Roofline People

Researchers...
- Sam Williams (Roofline Lead, LBL/CRD)
- Doug Doefler (LBL/NERSC)
- Khaled Ibrahim (LBL/CRD)
- Nan Ding (LBL/CRD)
- Yunsong Wang (LBL/NERSC)
- Jack Deslippe (LBL/NERSC)
- Lenny Oliker (RAPIDS deputy, LBL/CRD)
- Terry Ligocki (LBL/CRD)
- Brian Van Straalen (LBL/CRD)
- Aleksandar Ilic (INESC, Portugal)
- Diogo Marques (INESC, Portugal)

Vendors/Industry...
- Zakhar Matveev (Intel)
- Max Katz, Magnus Strengert (NVIDIA)
- Constantios Evangelinos (IBM)
- Protonu Basu (Facebook; formerly LBL/CRD)
- Linda Lo (Facebook; formerly U. Utah)
- David Patterson (Google, formerly UC Berkeley)

Thank You!
What is Roofline?
Modern architectures are complicated!  A holistic view is important!

Intel Haswell CPU

NVIDIA Volta GPU

Performance Modelling

- Many components contribute to the kernel run time
- An interplay of application characteristics and machine characteristics

#FP operations FLOP/s
Cache data movement Cache GB/s
DRAM data movement DRAM GB/s
PCIe data movement PCIe bandwidth
MPI Message Size Network Bandwidth
MPI Send:Wait ratio Network Gap
#MPI Wait’s Network Latency
IO File systems

Roofline Model

Focus on one or two dominant components!
Roofline Performance Model

- Sustainable performance is bound by
  \[ \text{GFLOP/s} = \min \left\{ \frac{\text{Peak GFLOP/s}}{\text{AI} \times \text{Peak GB/s}} \right\} \]

- Arithmetic Intensity (AI) = \[ \text{FLOPs / Bytes} \]

- How did this come about?
  \[ \rightarrow \text{A CPU DRAM example} \]

Transition @ AI ==
Peak GFLOP/s / Peak GB/s ==
‘Machine Balance’
One could hope to always attain peak performance (FLOP/s)
However, finite locality (reuse) and bandwidth limit performance.
Assume:
- Idealized processor/caches
- Cold start (data in DRAM)

Time = max \[
\frac{\text{#FP ops}}{\text{Peak GFLOP/s}} \quad \frac{\text{#Bytes}}{\text{Peak GB/s}}
\]
One could hope to always attain peak performance (FLOP/s).
However, finite locality (reuse) and bandwidth limit performance.

Assume:
- Idealized processor/caches
- Cold start (data in DRAM)

\[
\frac{\text{Time}}{\text{#FP ops}} = \max \begin{cases} 
\frac{1}{\text{Peak GFLOP/s}} \\
\frac{\text{Bytes}}{\text{#FP ops}} / \text{Peak GB/s}
\end{cases}
\]
One could hope to always attain peak performance (FLOP/s) However, finite locality (reuse) and bandwidth limit performance. Assume:
• Idealized processor/caches
• Cold start (data in DRAM)

\[
\frac{\text{#FP ops}}{\text{Time}} = \min \left\{ \text{Peak GFLOP/s} \right\}
\]

\[
(\text{#FP ops} / \text{#Bytes}) \times \text{Peak GB/s}
\]
One could hope to always attain peak performance (FLOP/s) However, finite locality (reuse) and bandwidth limit performance. Assume:

- Idealized processor/caches
- Cold start (data in DRAM)

$$\text{Peak GFLOP/s} = \min \left\{ \text{AI} \times \text{Peak GB/s} \right\}$$

Arithmetic Intensity ($\text{AI}$) = Flops / Bytes (as presented to DRAM)
Thus we obtain the model as

\[
GFLOP/s = \min \left\{ \text{Peak GFLOP/s} \right. \\
\left. \text{AI} \times \text{Peak GB/s} \right\}
\]

where Arithmetic Intensity (AI) is \( \text{FLOPs} / \text{Bytes} \)

• Machine Balance (FLOPs/Byte) = 8.9 (V100, DP, HBM) or 5.1 (KNL, DP, HBM)
Roofline Performance Model

- A throughput-oriented model
  - tracks rates not times, i.e. GFLOP/s, GB/s, not seconds

- An abstraction over
  - architectures, ISA (CPU, GPU, Haswell, KNL, Pascal, Volta)
  - programming models, programming languages
  - numerical algorithms, problem sizes

- In log-log scale to easily extrapolate performance along Moore’s Law
What can Roofline do?
Roofline is Useful for…

• **Identifying** performance bottlenecks & **motivating** software optimizations

• **Understanding** performance differences between architectures, programming models, implementations, etc

• **Determining** when we’re done optimizing code
  – Assess performance relative to machine capabilities
  – Motivate need for algorithmic changes

• **Predicting** performance on future machines / architectures
  – Set realistic performance expectations
  – Drive for HW/SW Co-Design
Activities on Roofline

Current, Future
The Roofline Tree

Brings People Together
- NESAP
- CRD
- Intel
- NVIDIA
- all HPCers

Performance Optimization
Scaling Trajectories
Energy Roofline
Mixed Precision
Vendor Integration
ERT
Outreach
Performance Portability
Instruction Roofline

Roofline Performance Model
1. Performance Optimization

NESAP, Hierarchical Roofline, Roofline drives optimization
Roofline Drives Optimization

The Roofline Model

- helps you identify the bottlenecks
- guides you through optimization
- tells you when to stop

An example:

- NESAP for Cori - BerkeleyGW

(NERSC Exascale Scientific Application Program)
Roofline Example: BerkeleyGW

Optimization Path for Kernel-C (Sigma):

1. Add OpenMP
2. Initial Vectorization
   - loop reordering
   - conditional removal
3. Cache-Blocking
4. Improved Vectorization
   - divides
5. Hyper-threading
1.1 Roofline Variations
Roofline Performance Model

- This is a single Roofline
- What about the memory hierarchy, different execution configurations, and instruction mixes?
  - Hierarchical Roofline
  - Multiple compute ceilings

\[ \text{Transition @ AI} \Rightarrow \frac{\text{Peak GFLOP/s}}{\text{Peak GB/s}} \Rightarrow \text{‘Machine Balance’} \]
Hierarchical Roofline

- Superposition of multiple Rooflines
  - Incorporates full memory hierarchy
  - Arithmetic Intensity = \( \frac{\text{FLOPs}}{\text{Bytes}} \)
    \( L_1/L_2/\text{HBM}/\text{SysMem} \)

- Each kernel will have multiple AI’s but one observed GFLOP/s performance

- Hierarchical Roofline tells you about cache locality
Cache-Aware Roofline Model (CARM)

Hierarchical Roofline

Cache-Aware Roofline

Hierarchical Roofline

Peak Flop/s

Attainable Flop/s

Arithmetic Intensity (Flop:Byte)

Capacity misses reduce DRAM AI and performance

Multiple AI’s:
- flop:DRAM ~ 0.20
- flop:L1 ~ 0.11

Single AI based on flop:L1 bytes

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

7-point Stencil
<table>
<thead>
<tr>
<th>Hierarchical</th>
<th>vs</th>
<th>Cache-Aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Captures cache effects</td>
<td></td>
<td>Captures cache effects</td>
</tr>
<tr>
<td>AI is Flop:Bytes after being filtered by lower cache levels</td>
<td></td>
<td>AI is Flop:Bytes as presented to the L1 cache (plus non-temporal stores)</td>
</tr>
<tr>
<td>Multiple Arithmetic Intensities (one per level of memory)</td>
<td></td>
<td>Single Arithmetic Intensity</td>
</tr>
<tr>
<td>AI dependent on problem size (capacity misses reduce AI)</td>
<td></td>
<td>AI independent of problem size</td>
</tr>
<tr>
<td>Memory/Cache/Locality effects are observed as decreased AI</td>
<td></td>
<td>Memory/Cache/Locality effects are observed as decreased performance</td>
</tr>
<tr>
<td>Requires performance counters or cache simulator to correctly measure AI</td>
<td></td>
<td>Requires static analysis or binary instrumentation to measure AI</td>
</tr>
</tbody>
</table>
Multiple Compute Ceilings

- **Impact of execution configuration**
  - Concurrency affects your peak
    - OpenMP thread concurrency
    - SM occupancy
    - load balance
    - threadblock/thread configuration

- Performance is bound by the **actual concurrency** ceiling
Multiple Compute Ceilings

- Impact of **instruction mix**
- Applications are usually a mix of FMA.f64, ADD.f64, MUL.f64...
- Performance is a **weighted** average ... bound by a **partial FMA ceiling**
1.2 Roofline Drives Optimization
Broadly speaking, three approaches to improving performance:
General Optimization Strategy

- Broadly speaking, three approaches to improving performance:
  - Maximize compute performance
    - multithreading
    - vectorization
    - increase SM occupancy
    - utilize FMA instructions
    - minimize thread divergence

![Diagram showing Attainable FLOP/s vs Arithmetic Intensity (FLOP:Byte)]

- Peak FLOP/s
- No FMA
- HBM GBs
- NUMA or O-H GBs
- No vectorization
General Optimization Strategy

- Broadly speaking, three approaches to improving performance:
  - Maximize compute performance
  - Maximize memory bandwidth
    - utilize higher-level caches
    - NUMA-aware allocation
    - avoid H-D transfers
    - avoid uncoalesced memory access
Broadly speaking, three approaches to improving performance:

- Maximize compute performance
- Maximize memory bandwidth
- Improve AI
  - minimize data movement
  - exploit cache reuse
1.3 Example Applications
Example 1: GPP, KNL, Cache Blocking

242 GFlop/s, Bound by MCDRAM Bandwidth

Most Flops in the main loop (O)

Read/Write 2MB of data per inner loop iteration ➤ No reuse of data in L1/L2, shown by overlapping points at MCDRAM bandwidth

BW Bound ➤ Increase MCDRAM AI by adding cache locality
Example 1: GPP, KNL, Cache Blocking

Cache blocking implemented to achieve L2 data reuse

3x Increase in MCDRAM AI

Performance increased from 242 to 287 GFlop/s (+18%)

Why not 3x Flops increase?
➤ Not BW bound any more, divide, shuffle and unpack instructions involved

Example 1: GPP, V100, Hierarchical

Three experiments to study the effects of
• cache reuse (varying \(nw\) from 1 to 6)
• instruction mix (FMA vs. Mul/Add)
• memory coalescing

```
  do band = 1, nbands  #blockIdx.x
    do igp = 1, ngpown  #blockIdx.y
      do ig = 1, ncouls  #threadIdx.x
        do iw = 1, nw    #unrolled
          compute; reductions
        enddo
      enddo
    enddo
  enddo
```

Example 2: XGC1, KNL

(Left) Hotspots for unoptimized XGC1 on 1024 Cori KNL nodes in Quad-Flat mode;
(Right) Speedup in XGC1 Electron Push routine after back porting the optimizations made in ToyPush kernel
Example 2: ToyPush from XGC1

- **Force Kernel:**
  - close to vector add peak
  - not much optimization done

- **Interpolate Kernel:**
  - L1 blocking, indirect memory access
  - memory alignment, more efficient vectorization
  - 10x speedup, closer to vector FMA peak

- **Search Kernel:**
  - multiple exits, simd private, enable vectorization
  - 3x speedup, closer to L2 bandwidth roof

- Code is available at
  - [https://github.com/tkoskela/toypush](https://github.com/tkoskela/toypush)
Example 3: `conv2d` from TensorFlow

- Kernel `tf.nn.conv2d`

```
B_{n,h,w,c} = \sum_{m=0}^{C-1} \sum_{k_h=0}^{K_H-1} \sum_{k_w=0}^{K_W-1} A_{n,h+k_h,w+w_h,m} K_{k_h,k_w,m,c}
```

https://www.tensorflow.org
**Example 3: conv2d from TensorFlow**

exec_op:
- **forward pass** -- conv in 2D
- **backward pass** -- conv + derivative
- **calibrate** -- tensor generation

```python
#choose operation depending on pass
if pass=="forward":
    with tf.device(gpu_dev):
        exec_op = output_result
elif pass=="backward":
    with tf.device(gpu_dev):
        opt = tf.train.GradientDescentOptimizer(0.5)
        exec_op = opt.compute_gradients(output_result)
elif pass=="calibrate":
    with tf.device(gpu_dev):
        exec_op = input_image
```

```python
#generate random input tensor
input_image = tf.random_uniform(shape=input_size, minval=0., maxval=1., dtype=dtype)
#create network
output_result = conv2d(input_image, 'NHWC', kernel_size, stride_size, dtype)
```
Each TensorFlow kernel translates to a series of subkernels
- padding, shuffling, data conversion, etc

TensorFlow based on heuristics decides what subkernels to call

cuDNN also has some algorithm selection mechanism

We **INCLUDE** the housekeeping subkernels in our measurements, but **EXCLUDE** the autotuning subkernels
Example 3: TF / Forward Pass

**#Batch Size**
- Constant performance (no!)
- FP16 performance anti-correlated with batch size
- Performance << TC peak
- Transformation kernels
- Low L2 locality

**#Filters**
- Intensity \(\propto\) #Filters
- Low L2 data locality
- Some use of TC’s (>FP16 FMA)… partial TC ceiling

**#Kernel Size**
- Intensity \(\propto\) kernel size
- Low L2 data locality
- Autotuner switched FP32 algorithm to FFT at 9x9
- Some use of TC’s (>FP16 FMA)… partial TC ceiling
Example 3: TF / Backward Pass

**#Batch Size**
- Autotuner chose different (better) algorithm for FP32 with batch size = 64 (boost)

**#Filters**
- Close to FP16 TC peak
- Close to FP32 FMA peak

**#Kernel Size**
- Good FP32 performance trend (almost peak)
- Autotuner chose to run 9x9 FP16 in FP32!!
Summary

• Useful for **characterization** as well as **optimization** of HPC applications

• Roofline has a wide **applicability**
  - different architectures (KNL, V100, ...)
  - different algorithms (Simulation, Machine Learning, ...)
2. Vendor Integration

Intel VTune, LIKWID, Intel Advisor, NVIDIA nvprof
Example #1: STREAM Triad

- 2 FLOPs per iteration
- Transfer 24 bytes per iteration
  - read $X[i]$, $Y[i]$, and write $Z[i]$
- $AI = 0.083$ FLOPs per byte
- Memory bound

for($i=0;i<N;i++$){
  $Z[i] = X[i] + alpha*Y[i]$;
}

Arithmetic Intensity (Flop:Byte)
Example #2: 7-pt stencil
- 7 FLOPs; 8 memory references (7 reads, 1 store) per pt
- Cache can filter all but 1 read and 1 write per pt
- $AI = 0.44$ FLOPs per byte
- Memory bound, but 5x the GFLOP/s rate

```c
for(k=1;k<dim+1;k++){
  for(j=1;j<dim+1;j++){
    for(i=1;i<dim+1;i++){
      new[k][j][i] = -6.0*old[k][j][i]
      + old[k][j][i-1]
      + old[k][j][i+1]
      + old[k][j-1][i]
      + old[k][j+1][i]
      + old[k-1][j][i]
      + old[k+1][j][i];
    }
  }
}
```

```
Pen and Paper
```

```
\begin{align*}
\text{Attainable FLOP/s} & \leq AI \times \text{DRAM GB/s} \\
\text{Peak FLOP/s} & \text{GFLOP/s} \leq AI \times \text{DRAM GB/s}
\end{align*}
```

```
TRIAD
```

```
7-point Stencil
```

```
0.083 0.44 5.1
```

```
Arithmetic Intensity (Flop:Byte)
```

```
Attainable FLOP/s
```

```
Peak FLOP/s
```

```
DRAM GB/s
```

```
GFLOP/s \leq AI \times \text{DRAM GB/s}
```

```
7-point Stencil
```

```
0.083 0.44 5.1
```

```
Arithmetic Intensity (Flop:Byte)
```

```
Attainable FLOP/s
```

```
Peak FLOP/s
```

```
DRAM GB/s
```

```
GFLOP/s \leq AI \times \text{DRAM GB/s}
```

```
7-point Stencil
```

```
0.083 0.44 5.1
```

```
Arithmetic Intensity (Flop:Byte)
```

```
Attainable FLOP/s
```

```
Peak FLOP/s
```

```
DRAM GB/s
```

```
GFLOP/s \leq AI \times \text{DRAM GB/s}
```

```
7-point Stencil
```
Pen and Paper

- Not scalable for real-life applications
- Millions of lines of code; mix of different languages
  - Complicated modern architecture
    - memory hierarchy, caching effects
    - ISA
- Different problem sizes

We need tools!
We Need Tools!

- Roofline ceilings
  - vendor specifications
  - empirical measurements
    - ERT
    - https://bitbucket.org/berkeleylab/cs-roofline-toolkit
We Need Tools!

Where to put these dots?
We Need Tools!

Require three raw measurements:
- Runtime
- FLOPs
- Bytes (on each cache level)

In order to calculate AI and GFLOP/s:

\[
\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Data Movement}} \quad (\text{FLOPs/Byte})
\]

\[
\text{Performance} = \frac{\text{FLOPs}}{\text{Runtime}} \quad (\text{GFLOP/s})
\]
Methodology to Construct Roofline

1. Collect Roofline ceilings
   - **compute** (FMA/no FMA) and **bandwidth** (DRAM, L2, …)
   - ERT: https://bitbucket.org/berkeleylab/cs-roofline-toolkit

2. Collect application performance
   - **FLOPs**, **bytes** (DRAM, L2, …), **runtime**
   - SDE, VTune, LIKWID, Advisor, nvprof, …

3. Plot Roofline with Python Matplotlib (or other tools of your preference)
   - **arithmetic intensity**, **GFLOP/s** performance, **ceilings**
   - example scripts: https://github.com/cyanguwa/nersc-roofline
2.1 Intel CPUs and NVIDIA GPUs
Data Collection on Intel CPUs

The not-so-automated way 1:

- **Intel SDE** for FLOPs (emulation)
- **Intel VTune** for DRAM bytes (HW counters)
- Runtime

- DRAM Roofline only

- Used by NESAP for Cori
  - NERSC Exascale Science Application Program
Data Collection on Intel CPUs

The not-so-automated way 2:

- **LIKWID** for FLOPs and bytes
  - Both are based on HW counters
- Runtime
- Hierarchical Roofline

- Limited by quality of HW counters
- High-level characterization, no callstack

https://github.com/RRZE-HPC/likwid
Data Collection on Intel CPUs

The fully automated way:

- Intel Advisor, Roofline feature
- Instrument applications automatically
  - one dot per loop nest/function
- FLOPs, bytes and runtime
- Hierarchical Roofline
- Integrates with other Advisor capabilities
- Benchmarks target system
Data Collection on Intel CPUs

New features in Intel Advisor 2019
(picture courtesy of Z. Matveev)

Data Collection on NVIDIA GPUs

- Still manual at this stage, but...

- Runtime:
  - Internal timers or `nvprof --print-gpu-trace`

- FLOPs:
  - DP/SP/HP counters and metrics, `nvprof --metrics `flop_count_dp/sp/hp’` or `tensor_precision_fu_utilization’`

- Bytes for different cache levels:
  - Bytes = (read transactions + write transactions) x transaction size
  - `nvprof --metrics `metric_name’` e.g. gld/gst_transactions

- Hierarchical Roofline
Brings People Together

- NESAP
- CRD
- Intel
- NVIDIA
- all HPCers

Roofline Performance Model
3. Performance Portability

Definition, Metric, Roofline, KNL, V100
Introduction

- No consensus on the definition or metric for performance portability
- But Pennycook et al…

\[
\Phi(a, p, H) = \begin{cases} 
\frac{|H|}{\sum_{i \in H} e_i(a, p)} & \text{if } i \text{ is supported, } \forall i \in H \\
0 & \text{otherwise}
\end{cases}
\]

- Application’s Efficiency ?
  - application performance on platform \(i\) / peak performance of platform \(i\)
  - application performance on platform \(i\) / application’s best performance on all platforms of interest \(H\)

Introduction

• No consensus on the definition or metric for performance portability

• But Pennycook et al…

\[ \Phi(a, p, H) = \begin{cases} \frac{|H|}{\sum_{i \in H} e_i(a, p)} & \text{if } i \text{ is supported, } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \]

• Architectural Efficiency [Williams et al]

\[ e_i(a, p) = \frac{P_i(a, p)}{\min(F_i, B_i \times I_i(a, p))} \]

Actual Application Performance

Max Attainable Performance defined by Roofline

---


Introduction

- No consensus on the definition or metric for performance portability
- But Pennycook et al…

\[
\Phi(a, p, H) = \begin{cases} 
\frac{|H|}{\sum_{i \in H} e_i(a, p)} & \text{if } i \text{ is supported, } \forall i \in H \\
0 & \text{otherwise}
\end{cases}
\]

- Architectural Efficiency [Williams et al]

\[
e_i(a, p) = \frac{P_i(a, p)}{\min(F_i, B_i \times I_i(a, p))}
\]

Peak FLOP/s  
Arithmetic Intensity  
Peak Bandwidth
Bottleneck Changes

- Bottleneck shifts at $nw = 2$ on KNL vs. V100 (no-FMA performance)
- Easier to achieve no-FMA ceiling on V100 than KNL, due to higher ratio of instruction issue bandwidth vs. instruction execution bandwidth
No FMA: performance portability consistently > 80%

FMA: benefit is far less than 2x at high $nw$; architectural efficiency suffers (so does performance portability)

Could regain some architectural efficiency if non-floating-point vector operations were considered

| Architectural Efficiency | $nw = 1$ | $nw = 2$ | $nw = 3$ | $nw = 4$ | $nw = 5$ | $nw = 6$
|--------------------------|----------|----------|----------|----------|----------|----------
| FMA                      |          |          |          |          |          |          
| KNL                      | 84.98%   | 77.50%   | 66.77%   | 55.28%   | 46.56%   | 39.65%   
| V100                     | 97.36%   | 91.50%   | 76.70%   | 65.44%   | 65.07%   | 66.38%   
| Performance Portability  | 90.76%   | 83.92%   | 71.39%   | 59.93%   | 54.28%   | 49.65%   
| No-FMA                   |          |          |          |          |          |          
| KNL                      | 82.06%   | 72.95%   | 73.74%   | 78.72%   | 81.28%   | 82.81%   
| V100                     | 92.88%   | 92.88%   | 97.43%   | 98.91%   | 1        | 99.73%   
| Performance Portability  | 87.14%   | 81.72%   | 83.95%   | 87.67%   | 89.93%   | 90.49%   

Bottleneck Changes
Summary

• Roofline is very powerful in capturing changes in machine and application characteristics such as
  – compute/bandwidth bound, problem size
  – instruction issue bandwidth, strided memory access

• It is important to
  – measure bandwidth/compute ceilings empirically
  – account for non-multiply/add instructions appropriately
  – select relevant ceilings in performance analysis and performance portability analysis
4. Energy Roofline
Performance, Power Consumption, Energy Efficiency
Energy Roofline - GEMM

Cache-aware Roofline Models

- Power Consumption based on CARM
  - Relates Watts with FLOPs/bytes
  - Defines power envelope for different types of FP and memory operations

<table>
<thead>
<tr>
<th>VERSION</th>
<th>OPTIMIZATION STRATEGY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Basic implementation: Row-major matrices</td>
</tr>
<tr>
<td>2</td>
<td>Improved memory access by transposing B matrix</td>
</tr>
<tr>
<td>3, 4, 5</td>
<td>Blocking for caches: L3 (pt. 3), L2 (pt. 4) and L1 (pt. 5)</td>
</tr>
<tr>
<td>6</td>
<td>Highly optimized Intel MKL implementation</td>
</tr>
</tbody>
</table>

Use Cases

Application Characterization

Online Monitoring


5. Scaling Trajectories

What’s causing bad scaling from Roofline point of view?
Roofline Scaling Trajectories

- We often plot performance as a function of thread concurrency
  - Carries no insight or analysis
  - Provides no actionable info
Roofline Scaling Trajectories

- We often plot performance as a function of thread concurrency
  - Carries no insight or analysis
  - Provides no actionable information.

- Use Roofline to analyze thread (or process) scalability
  - 2D scatter plot of performance as a function of intensity and concurrency
  - Identify loss in performance due to increased cache pressure (data movement)

Khaled Ibrahim, Samuel Williams, Leonid Oliker, "Roofline Scaling Trajectories: A Method for Parallel Application and Architectural Performance Analysis", HPBench, July 2018
6. Mixed Precision

FP64, FP32, FP16, CPU, GPU

NERSC
Benefits of reduced/mixed precision:
- From FP64 to FP32
  - 2x due to bandwidth savings or compute unit availability
  - similar for network communication
- More support on modern architectures
  - ~15x FP16 over FP64 for some ops

NESAP collaboration with CRD (Costin Iancu) and NVIDIA (Chris Newburn)
7. Instruction Roofline

FLOP, INTOP, IPC
FP instructions can be the minority in many HPC codes

Emerging domains have ~no FP
- Graphs
- Hash tables
- Bloom filters
- Searches

FLOPs is agnostic of precision, scalar/vectors/tensors, …

Instruction Roofline
Instruction Roofline

FLOPs-based Roofline

- FMA doesn’t change Arithmetic Intensity (FMA == FMUL+FADD)
- Vectors/tensors don’t change Arithmetic Intensity
- Vector integer operations don’t change Arithmetic Intensity
- Reducing precision (64b, 32b, 16b) increases Arithmetic Intensity

Tells us about performance

VUOPs-based Roofline

- FMA cuts Arithmetic Intensity in half (half the number of VUOPS)
- vectors/tensors reduce Arithmetic Intensity (SIMD cuts VUOPS by 8x)
- Vector integer operations increases Arithmetic Intensity
- Changing precision doesn’t change Arithmetic Intensity

Tells us about VPU/pipeline utilization and bottlenecks
8. Empirical Roofline Toolkit (ERT)

Machine Characterization, Peak FLOP/s, Bandwidths

NERSC
Empirical vs. Theoretical Ceilings

Theoretical compute ceiling on KNL:

$$64 \text{ cores} \times 8 \text{ DP/vector} \times 2 \text{ FLOPs/FMA} \times 2 \text{ vectors} \times 1.2 \text{ GHz} = 2.46 \text{ TFLOP/s}$$

Theoretical compute ceiling on V100:

$$80 \text{ SMs} \times 32 \text{ FP64 cores/SM} \times 2 \text{ FLOPs/FMA} \times 1.53 \text{ GHz} = 7.83 \text{ TFLOP/s}$$

![Graph showing theoretical vs. empirical ceilings for KNL and V100](image)
Machine Characterization

- ERT can’t detect all the ceilings yet - IN DEVELOPMENT!
  - Haswell/KNL: L1, L2, L3/HBM, DDR
  - V100: L2, HBM, DDR

- Our goal is to incorporate
  - the full memory hierarchy
  - instruction mix (e.g. FMA/no-FMA)
  - data type (e.g. FP64, FP32, FP16)
  - compute units
    (e.g. CPU/CUDA core/Tensor core)

- Ceilings can be omitted if irrelevant

9. Outreach

Tutorials, Talks, Publications, Collaboration
Tutorials

- S. Williams, J. Deslippe, C. Yang, P. Basu, Performance Tuning of Scientific Codes with the Roofline Model, Exascale Computing Project 2nd Annual Meeting, Feb 5-9, 2018, Knoxville
- S. Williams, J. Deslippe, C. Yang, Performance Tuning of Scientific Codes with the Roofline Model, Exascale Computing Project (ECP) Annual Meeting, Jan 14-18, 2019, Houston
Talks

- C. Yang, S. Williams, Performance Analysis of GPU-Accelerated Applications using the Roofline Model, GTC’2019, Mar 17-21 2019, San Jose
- C. Yang, Roofline Performance Analysis with nvprof, NERSC/NVIDIA F2F, Feb 7 2019, Berkeley
- C. Yang, S. Williams, Performance tools and performance counters, Intel PathForward Hack-a-Thon, Apr 10-12 2018, Santa Clara
- C. Yang, Performance Analysis of GPU-Accelerated Applications using the Roofline Model, Cray COE Webinar, Apr 11 2019, Berkeley
- C. Yang, Using Intel Tools at NERSC, Intel KNL Training, May 22-23 2019, Berkeley
All Publications...

LBNL CRD Roofline Research:

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

Collaborate with us!
Closing
Roofline Helps

- Drive performance optimization
  - Application characterization, application readiness

- Create a dialogue between Applied Maths and CS
  - Communication-avoiding algorithms, high-order methods, new algorithms

- Facilitate cross-architecture comparisons for procurements
  - CPUs, GPUs, other accelerators
• 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.
Thank You