Performance Analysis of GPU-Accelerated Applications using the Roofline Model

GTC 2019, San Jose

Charlene Yang
Application Performance Specialist
NERSC, LBNL
cjyang@lbl.gov

Samuel Williams
Senior Staff Scientist
CRD, LBNL
swwilliams@lbl.gov
You just bought a $10,000 throughput-optimized GPU!

Are you making good use of your investment?
You could just run benchmarks

- Imagine a mix of benchmarks or kernels…
- GFLOP/s alone may not be particularly insightful
- Moreover, speedup relative to a Xeon may seem random
Making good use of your GPU?

1. Are you operating it in the throughput-limited regime?
   - Not sensitive to Amdahl effects
   - Not sensitive to D2H/H2D transfers
   - Not sensitive to launch overheads
   - Not sensitive to latencies

2. If in the throughput-limited regime, are you making good use of the GPU’s **compute** and **bandwidth** capabilities?
The Roofline Model

- **Roofline Model** is a throughput-oriented performance model
- Premised on the interplay between FLOP/s, bandwidth, and reuse
- Tracks rates not times
- Independent of ISA and architecture (applies to CPUs, GPUs, Google TPUs, etc…)

https://crd.lbl.gov/departments/computer-science/PAR/research/roofline
One could hope to always attain peak performance (GFLOP/s) However, finite locality (reuse) and bandwidth limit performance.

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

\[
\text{Time} = \max \left\{ \frac{\text{#FLOPs}}{\text{Peak GFLOP/s}}, \frac{\text{#Bytes}}{\text{Peak GB/s}} \right\}
\]
One could hope to always attain peak performance (GFLOP/s) however, finite locality (reuse) and bandwidth limit performance.

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

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

Note, Arithmetic Intensity (AI) = FLOPs / Bytes (as presented to DRAM)
Arithmetic Intensity

- **Arithmetic Intensity** is the most important concept in Roofline.
- Measure of data locality (data reuse)
- Ratio of **Total FLOPs** performed to **Total Bytes** moved
- For the DRAM Roofline…
  - Total Bytes to/from DRAM and includes all cache and prefetcher effects
  - Can be very different from total loads/stores (bytes requested) due to cache reuse
(DRAM) Roofline

- Plot Roofline bound using Arithmetic Intensity as the x-axis
- **Log-log scale** makes it easy to doodle, extrapolate performance along Moore’s Law, etc...
- Kernels with AI less than machine balance are ultimately DRAM bound (we’ll refine this later…)

Transition @ AI ==
Peak Gflop/s / Peak GB/s ==
‘Machine Balance’
Example

- Consider 3 kernels (A, B, C)
  - calculate or measure the **Arithmetic Intensity** for each
  - Determine the Roofline intercept for each kernel
    - kernels A and B are bound by memory bandwidth
    - kernel C is bound by peak FLOP/s
Scaling to Future GPUs

- Imagine you run on a future GPU with twice the peak FLOPs…
  - kernel C’s performance could double
  - kernels A and B will be no faster
Scaling to Future GPUs

- What if that future GPU also doubled its memory bandwidth…
  - kernel A and B’s performance could also double
Why is Roofline Useful?

- Think back to our mix of loop nests where GFLOP/s alone wasn’t useful…
Why is Roofline Useful?

- We can sort kernels by AI …
Why is Roofline Useful?

- We can sort kernels by AI …
- … and compare performance relative to machine capabilities
Why is Roofline Useful?

- Kernels near the roofline are making good use of computational resources…
  - kernels can have low performance (GFLOP/s), but make **good** use of a machine
  - kernels can have high performance (GFLOP/s), but make **poor** use of a machine
Can Performance Be Below Roofline?

- Analogous to asking whether one can always attain either…
  - Peak Bandwidth
  - Peak GFLOP/s

- **Sure, there can be other performance bottlenecks…**
  - Cache bandwidth / locality
  - Lack of FMA / tensor instructions
  - Thread divergence / predication
  - Too many non-FP instructions
  - …
Cache Effects…

- Hierarchical Roofline Model
- Construct superposition of Rooflines…
  - Measure AI and bandwidth for each level of memory/cache
  - Loop nests will have multiple AI’s and multiple performance bounds…
  - … but performance is ultimately the minimum of these bounds.
Cache Effects…

- Hierarchical Roofline Model
  - Construct superposition of Rooflines…
    - Measure AI and bandwidth for each level of memory/cache
    - Loop nests will have multiple AI’s and multiple performance bounds…
    - ... but performance is ultimately the minimum of these bounds.

- Extend to other memories…
  - L1 / Shared
  - System
Widely separated Arithmetic Intensities indicate high reuse in the cache.
Insights – Exploiting Caches

- Widely separated Arithmetic Intensities indicate high reuse in the cache
- Similar Arithmetic Intensities indicate effectively no cache reuse (== streaming)
- As one changes problem size, L2 and DRAM arithmetic intensities can behave very differently
Failure to Exploit CISC Instructions

- Death of Moore’s Law is motivating a return of Complex Instruction Set Computing (CISC)

- Modern CPUs and GPUs are increasingly reliant on special (fused) instructions that perform multiple operations.
  - FMA (Fused Multiply Add): \( z=a \times x + y \) …\( z, x, y \) are vectors or scalars
  - 4FMA (quad FMA): \( z=A \times x + z \) …\( A \) is a FP32 matrix; \( x, z \) are vectors
  - HMMA (Tensor Core): \( Z=AB+C \) …\( Z, A, B, C \) are FP16 matrices
  - …

- Performance is now a weighted average of Mul/Add, FMA, and HMMA operations.
Failure to Exploit CISC Instructions

- Total lack of FMA reduces Volta performance by 2x…
  - creates ADD.f64 ceiling

- In reality, applications are a mix of FMA.f64, ADD.f64, and MUL.f64…
  - Performance is a weighted average
    - Produces a partial FMA ceiling that bounds kernel performance
Failure to Exploit CISC Instructions

- On Volta, Tensor cores provide **125 TFLOPs** of FP16 performance (vs. 15 for FP32)
- However, kernels/apps will mix HMMA with FMA, MULs, ADDs, …
  - A few non-HMMA operations can quickly limit Tensor core performance
Using Roofline To Drive Optimization
Broadly speaking, there are three approaches to improving performance:

- Peak GFLOP/s
- No FMA
- Arithmetic Intensity (FLOP:Byte)
Broadly speaking, there are three approaches to improving performance:

- **Maximize SM performance** (e.g. minimize predication)
Driving Performance Optimization

- Broadly speaking, there are three approaches to improving performance:
  - Maximize SM performance (e.g. minimize predication)
  - Maximize memory bandwidth (e.g. avoid pathological memory access patterns)
Broadly speaking, there are three approaches to improving performance:

- Maximize SM performance (e.g. minimize predication)
- Maximize memory bandwidth (e.g. avoid pathological memory access patterns)
- **Minimize data movement** (i.e. exploit reuse)
Estimating Arithmetic Intensity
Consider a 7-point constant coefficient stencil…
- 7 FLOPs
- 8 memory references (7 reads, 1 store) per point
- \( AI = 0.11 \) FLOPs per byte (L1)

```c
#pragma omp parallel for
define
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-1][j][i]
                        + old[k][j+1][i]
                        + old[k+1][j][i];
        }
    }
}
```
**DRAM vs L1 Arithmetic Intensity**

- Consider a 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
  - $AI = 0.44$ FLOPs per byte

```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++)
{
    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];
}
```
Consider a 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 = 0.44 FLOPs per byte == memory bound

```plaintext
#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++){
      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];
    }
  }
}
```
Collecting Roofline Data with nvprof
General Roofline Data Collection

Most kernels are more complicated than the 7-point stencil…
Most kernels are more complicated than the 7-point stencil…

How do we measure the total number of FLOPs?
How do we measure the total number of bytes moved (read/write, L1/L2/HBM)?
How do we measure the runtime for each kernel?

How do we know the peak bandwidth (L1/L2/HBM) and the peak FLOP/s for the architecture?
General Roofline Data Collection

Most kernels are more complicated than the 7-point stencil…

How do we measure the total number of FLOPs?
How do we measure the total number of bytes moved (read/write, L1/L2/HBM)?
How do we measure the runtime for each kernel?

How do we know the peak bandwidth (L1/L2/HBM) and the peak FLOP/s for the architecture?

nvprof

ERT
Step 1. Collect Roofline Ceilings

- **Empirical Roofline Toolkit (ERT)**
  - Different than the architecture specs, **MORE REALISTIC**
  - Reflects **actual** execution environment (power constraints, *etc*)
  - Sweeps through a range of configurations, and **statistically stable**
    - Data elements per thread
    - FLOPs per data element
    - Threadblocks/threads
    - Trails per dataset
    - *etc*


![Graph showing Total Bandwidth vs Working Set Size (bytes)](image-url)
ERT Configuration

**Kernel.c**
- actual compute
- customizable

**Driver.c**
- setup
- call kernels
- loop over parameters

**config script**
- set up ranges of parameters

**job script**
- submit the job and run it

ERT Configuration

ERT_FLOPS = 1, 2, 4, 8, 16, 32, 64, 128, 256
ERT_GPU_BLOCKS = 80, 160, 320, 640, 1280, 2560
ERT_GPU_THREADS = 64, 128, 256, 512, 1024
ERT_MEMORY_MAX = 1073741824
ERT_WORKING_SET_MIN = 128
ERT_TRIALS_MIN = 1

Driver.c (uses some Macros from config.txt)
initialize MPI, CUDA
loop over dataset sizes ≤ ERT_MEMORY_MAX
loop over trial sizes ≥ ERT_TRIALS_MIN
cudaMemcpy
start timer
call kernel
end timer

Kernel.c
loop over ntrails
distribute dataset on threads and each computes
ERT_FLOPS=1: a = b + c
ERT_FLOPS=2: a = a x b + c

job script
• submit the job and run it

config script
• set up ranges of parameters

Driver.c
• setup
• call kernels
• loop over parameters

Kernel.c
• actual compute
• customizable
ERT Output

roofline.json

```
"gbytes": {
  "data": [
    "L1", 2996.82
  ],
  "DRAM", 828.83
},

"gflops": {
  "data": [
    "GFLOPs", 7068.90
  ]
},
```

roofline.ps

Empirical Roofline Graph (Results.cori.nersc.gov.03/Run.001)

7068.9 GFLOPs/sec (Maximum)

L1 - 2996.8 GB/s
DRAM - 828.8 GB/s

39
ERT Output

roofline.json

```
"gbytes": {
  "data": [
    ["L1", 2996.82],
    ["DRAM", 828.83],
  ],
},

"gflops": {
  "data": [
    ["GFLOPs", 7068.90],
  ],
},
```

roofline.ps

NVIDIA V100 -- Voltar at UOregon

Empirical Roofline Graph (Results.cori.nersc.gov.03/Run.001)

- L1: 2996.8 GB/s
- DRAM: 828.8 GB/s

7068.9 GFLOPs/sec (Maximum)
ERT Output

roofline.json

```
"gbytes": {
  "data": [
    ["L1", 2996.82],
    ["DRAM", 828.83]
  ],
},

"gflops": {
  "data": [
    ["GFLOPs", 7068.90]
  ],
},
```

roofline.ps

NVIDIA V100 -- Voltar at UOregon

Missing L1 due to L2 saturation, before L1 saturation; Use specs instead

Empirical Roofline Graph (Results.cori.nersc.gov.03/Run.001)
Theoretical FP64 compute ceilings on V100:

- FMA: \(80 \text{ SMs} \times 32 \text{ FP64 cores} \times 1.53 \text{ GHz} \times 2 = 7.83 \text{ TFLOP/s}\)
- no FMA: \(80 \text{ SMs} \times 32 \text{ FP64 cores} \times 1.53 \text{ GHz} = 3.92 \text{ TFLOP/s}\)

Theoretical memory bandwidths on V100:

- HBM: 900 GB/s
- L2: \(~4.1 \text{ TB/s}\)
- L1: \(~14 \text{ TB/s}\)

You may never achieve 7.8 TFLOP/s

You may be closer to the ceiling than you think you are
Step 2. Collect Application Performance
Step 2. Collect Application Performance

Where to put these dots?
Step 2. Collect Application Performance

Require three raw measurements:

- Runtime
- FLOPs
- Bytes (on each cache level)

to calculate AI and GFLOP/s:

Arithmetic Intensity = \frac{\text{nvprof FLOPs}}{\text{nvprof Data Movement}}

Performance = \frac{\text{nvprof FLOPs}}{\text{Runtime}}
Collect Application Performance

- **Runtime:**
  - Time per invocation of a kernel
    
    ```bash
    nvprof --print-gpu-trace ./application
    ```
  - Average time over multiple invocations
    
    ```bash
    nvprof --print-gpu-summary ./application
    ```
  - Same kernel with different input parameters are grouped separately

- **FLOPs:**
  - Predication aware and complex-operation aware (such as divides)
    
    ```bash
    nvprof --kernels 'kernel_name' --metrics 'flop_count_xx' ./application
    ```
  - e.g. `flop_count_{dp/dp_add/dp_mul/dp_fma, sp*, hp*}`
Collect Application Performance

- Bytes for different cache levels in order to construct hierarchical Roofline:
  - **Bytes = (read transactions + write transactions) x transaction size**
  - `nvprof --kernels 'kernel_name' --metrics 'metric_name'
    ./application`

<table>
<thead>
<tr>
<th>Level</th>
<th>Metrics</th>
<th>Transaction Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Level Cache*</td>
<td><code>gld_transactions, gst_transactions, atomic_transactions, local_load_transactions, local_store_transactions, shared_load_transactions, shared_store_transactions</code></td>
<td>32B</td>
</tr>
<tr>
<td>Second Level Cache</td>
<td><code>l2_read_transactions, l2_write_transactions</code></td>
<td>32B</td>
</tr>
<tr>
<td>Device Memory</td>
<td><code>dram_read_transactions, dram_write_transactions</code></td>
<td>32B</td>
</tr>
<tr>
<td>System Memory</td>
<td><code>system_read_transactions, system_write_transactions</code></td>
<td>32B</td>
</tr>
</tbody>
</table>

- Note: surface and texture transactions are ignored here for simplicity (HPC applications)
Example Output

[cjyang@voltar source]$ nvprof --kernels "1:7:smooth_kernel:1" --metrics flop_count_dp --metrics gld_transactions --metrics gst_transactions --metrics l2_read_transactions --metrics l2_write_transactions --metrics dram_read_transactions --metrics dram_write_transactions --metrics sysmem_read_bytes --metrics sysmem_write_bytes ./hpgmg-fv-fp 5 8

- Export to CSV: --csv -o nvprof.out

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;Tesla V100-PCIE-16GB (0)&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel: void smooth_kernel&lt;int=6, int=32, int=4, int=8&gt;(level_type, int, int, double, double, int, double*, double*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>flop_count_dp</td>
<td>Floating Point Operations(Double Precision)</td>
<td>30277632</td>
<td>30277632</td>
<td>30277632</td>
</tr>
<tr>
<td>1</td>
<td>gld_transactions</td>
<td>Global Load Transactions</td>
<td>4280320</td>
<td>4280320</td>
<td>4280320</td>
</tr>
<tr>
<td>1</td>
<td>gst_transactions</td>
<td>Global Store Transactions</td>
<td>73728</td>
<td>73728</td>
<td>73728</td>
</tr>
<tr>
<td>1</td>
<td>l2_read_transactions</td>
<td>L2 Read Transactions</td>
<td>890596</td>
<td>890596</td>
<td>890596</td>
</tr>
<tr>
<td>1</td>
<td>l2_write_transactions</td>
<td>L2 Write Transactions</td>
<td>85927</td>
<td>85927</td>
<td>85927</td>
</tr>
<tr>
<td>1</td>
<td>dram_read_transactions</td>
<td>Device Memory Read Transactions</td>
<td>702911</td>
<td>702911</td>
<td>702911</td>
</tr>
<tr>
<td>1</td>
<td>dram_write_transactions</td>
<td>Device Memory Write Transactions</td>
<td>151487</td>
<td>151487</td>
<td>151487</td>
</tr>
<tr>
<td>1</td>
<td>sysmem_read_bytes</td>
<td>System Memory Read Bytes</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>sysmem_write_bytes</td>
<td>System Memory Write Bytes</td>
<td>160</td>
<td>160</td>
<td>160</td>
</tr>
</tbody>
</table>
Step 3. Plot Roofline with Python

- Calculate Arithmetic Intensity and GFLOP/s performance
  - x coordinate: Arithmetic Intensity
  - y coordinate: GFLOP/s performance

\[
\text{Performance} = \frac{\text{nvprof FLOPs}}{\text{Runtime}} \text{ (GFLOP/s)}, \quad \text{Arithmetic Intensity} = \frac{\text{nvprof FLOPs}}{\text{nvprof Data Movement}} \text{ (FLOPs/Byte)}
\]

- Plot Roofline with Python Matplotlib
  - Example scripts:
  - [https://github.com/cyanguwa/nersc-roofline/tree/master/Plotting](https://github.com/cyanguwa/nersc-roofline/tree/master/Plotting)
  - Tweak as needed for more complex Rooflines
Plot Roofline with Python

- Quick example: `plot_roofline.py data.txt`
- Accepts space-delimited list for values
- Use quotes to separate names/labels

```
data.txt

# all data is space delimited
memroofs 14336.0 2996.8 828.758
mem Roof_names 'L1' 'L2' 'HBM'
comproofs 7068.86 3535.79
comp Roof_names 'FMA' 'No-FMA'

# omit the following if only plotting roofs
# AI: arithmetic intensity; GFLOPs: performance
AI 0.87 2.25 2.58
GFLOPs 2085.756683
labels 'Kernel'
```
Recap: Methodology to Construct Roofline

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

2. Collect application performance
   - nvprof: --metrics, --events, --print-gpu-trace
   - FLOPs, bytes (DRAM, L2, …), runtime

3. Plot Roofline with Python Matplotlib
   - arithmetic intensity, GFLOP/s performance, ceilings
   - example scripts: https://github.com/cyanguwa/nersc-roofline
Recap: Methodology to Construct Roofline

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

2. Collect application performance
   - nvprof: **--metrics, --events, --print-gpu-trace**
   - FLOPs, bytes (DRAM, L2, …), runtime

3. Plot Roofline with Python Matplotlib
   - arithmetic intensity, GFLOP/s performance, ceilings
   - example scripts: https://github.com/cyanguwa/nersc-roofline
Recap: Methodology to Construct Roofline

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

2. Collect application performance
   - nvprof: --metrics, --events, --print-gpu-trace
   - **FLOPs**, **bytes** (DRAM, L2, …), **runtime**

3. Plot Roofline with Python Matplotlib
   - arithmetic intensity, GFLOP/s performance, ceilings
   - example scripts: https://github.com/cyanguwa/nersc-roofline
Recap: Methodology to Construct Roofline

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

2. Collect application performance
   - nvprof: --metrics, --events, --print-gpu-trace
   - **FLOPs**, **bytes** (DRAM, L2, …), **runtime**

3. Plot Roofline with Python Matplotlib
   - **arithmetic intensity**, **GFLOP/s** performance, **ceilings**
   - example scripts: https://github.com/cyanguwa/nersc-roofline
Roofline Analysis with Use Cases
Code Example 1: GPP

- GPP (General Plasmon Pole) kernel from BerkeleyGW (Material Science)
- https://github.com/cyanguwa/BerkeleyGW-GPP
- Medium problem size: 512 2 32768 20

- Tensor-contraction, abundant parallelism, large reductions
- Low FMA counts, divides, complex double data type, HBM data 1.5GB

Pseudo Code

```c
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
```

56
Code Example 1: GPP

- Three experiments:

<table>
<thead>
<tr>
<th>Vary ( n_w ) from 1 to 6</th>
<th>To study impact of varying Arithmetic Intensity on performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile w/wo FMA</td>
<td>To study impact of instruction mix on performance on performance</td>
</tr>
<tr>
<td>Stride ( i_g ) loop</td>
<td>To study impact of suboptimal memory coalescing on performance</td>
</tr>
</tbody>
</table>

- Note that \texttt{nvprof} has already taken care of
  - Appropriate counting of FLOPs for complex instructions
    - div, exp, log and sin/cos should be counted as multiple FLOPs rather than 1
  - Appropriate counting of FLOPs for predicated-out threads
    - FLOPs are only counted on non-predicated threads


Code Example 1: GPP

- Highly parameterizable
  1. Varying \( nw \) from 1 to 6 to increase arithmetic intensity
     - FLOPs increases, but data movement stays (at least for HBM)

Pseudo Code

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

2. Compiling with and without FMA
   - -fmad=true/false
Code Example 1: GPP

- Highly parameterizable
  3. Striding \(ig\) loop to analyze impact of suboptimal memory coalescing
    - Split \(ig\) loop to two loops and place the ‘blocking’ loop outside

Pseudo Code

```
do band = 1, nbands      # blockIdx.x
  do igp = 1, ngpown       # blockIdx.y
    do igs = 0, stride - 1
      do ig = 1, ncouls/stride # threadIdx.x
        do iw = 1, nw          # unrolled
          compute; reductions
```

Stride 2
Code Example 1: GPP

- **Experiments 1:** study the impact of varying AI on performance

- HBM Roofline, i.e. bytes are HBM bytes
  - AI increases as \( nw \) grows
  - GPP moves from a bandwidth bound region to a compute bound region

- Roofline captures the change in AI


- **Experiments 1 & 2:** study the impact of instruction mix on performance

- HBM Roofline, i.e. bytes are HBM bytes
  - No-FMA performance converges to the no-FMA ceiling, but FMA performance is still far from the FMA ceiling
  - Not reaching FMA ceiling due to lack of FMA instructions

- Roofline captures effects of instruction mix


Code Example 1: GPP

- **Experiments 1 & 2:** study the impact of instruction mix on performance

- At $nw=6$, GPP has $\alpha = \frac{\text{FMA FP64 instr.}}{\text{FMA FP64 instr.} + \text{non-FMA FP64 instr.}} = 60\%$ of FMA instructions

- Expected performance is $\beta = \frac{\alpha \times 2 + (1 - \alpha)}{2} = 80\%$ of compute peak.

But at $nw=6$, GPP is only achieving 66%.

- Other FP/non-FP instructions may be taking up the instruction issue/execution pipeline

- Partial Roofline can show you the headroom
Experiments 1 & 2: What else is going on?

Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
- GPP is HBM bound at low nw’s and compute bound at high nw’s
- FLOPs $\propto nw$
- HBM bytes: constant
- L2 bytes: increasing at $\alpha > 1$
- L1 bytes: constant
- Spike in L2 curve at nw=2, 3

Hierarchical Roofline captures more details about cache locality
**Code Example 1: GPP**

- **Experiment 3:** study the effects of suboptimal memory coalescing
  - \(nw=6\)

- Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
  - L1/L2 bytes doubles from stride 1 to 2, but stays almost constant afterwards
  - at \(nw=6\), GPP moves from compute bound to bandwidth bound
  - Eventually all dots converge to HBM

- Roofline captures effects of memory coalescing
Code Example 2: HPGMG

- HPGMG (High-performance Geometric Multigrid) from Adaptive Mesh Refinement codes
  - https://bitbucket.org/nsakhranykh/hpgmg-cuda

- Stencil code, F-cycles and V-cycles, GSRB smoother kernel (Gauss-Seidel Red-Black)

Code Example 2: HPGMG

- Hybrid GPU and CPU code
  - Example: `hpgmg-fv 7 8`
  - $128^3$ box x 8, Level 5-8 run on GPU, Level 1-4 on CPU

- Three versions of GSRB kernel
  - GSRB_FP, GSRB_BRANCH, GSRB_STRIDE2
for(int k=klo; k<(klo+kdim); k++){
    const int ijk = i + j*jStride + k*kStride;
    const double *__restrict__ RedBlack =
        level.RedBlack_FP + ghosts*(1+jStride)
        +((k^color000)&1)*kStride;
    const double Ax = apply_op_ijk();
    const double lambda = Dinv_ijk();
    const int ij = i + j*jStride;
    xo[ijk] = X(ijk) + RedBlack[ij]*lambda*(rhs[ijk]-Ax);
}
**Code Example 2: HPGMG**

**GSRB_FP**

- Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes

- Highly bandwidth bound, inherent to stencil codes
- From Level 5 to Level 8:
  - AI slightly increases due to better Surface: Volume ratio
  - More HBM bound as more data is read in

- Roofline captures computational characteristics of the algorithm
Code Example 2: HPGMG

**GSRB_FP**

```c
for(int k=klo; k<(klo+kdim); k++){
    const int ijk = i + j*jStride + k*kStride;
    const double *__restrict__ RedBlack =
        level.RedBlack_FP + ghosts*(1+jStride)
        +((k^color000)&1)*kStride;
    const double Ax = apply_op_ijk();
    const double lambda = Dinv_ijk();
    const int ij = i + j*jStride;
    xo[ijk] = X(ijk) + RedBlack[ij]*lambda*(rhs[ijk]-Ax);
}
```

**GSRB_BRANCH**

```c
for(int k=klo; k<klo+kdim; k++){
    const int ijk = i + j*jStride + k*kStride;
    if(((i^j^k^color000^1)&1)) {
        const double Ax = apply_op_ijk();
        const double lambda = Dinv_ijk();
        xo[ijk] = X(ijk) + lambda*(rhs[ijk]-Ax);
    } else {
        xo[ijk] = X(ijk);
    }
}
```

- **GSRB_BRANCH** has half the FLOPs as **GSRB_FP** but the same HBM/L1/L2 bytes.
**Code Example 2: HPGMG**

**GSRB_FP vs. GSRB_BRANCH**

- FLOPs halves, bytes doesn’t change, thus AI halves and GFLOP/s halves
- Runtime is comparable even though GFLOP/s has halved
- Same number of threads occupied, only with half predicated in GSRB_BRANCH
Code Example 2: HPGMG

GSRB_STRIDE2

```cpp
for(int k=klo; k<klo+kdim; k++){
    i = ilo +!((ilo^j^k^color000)&1) + threadIdx.x*2;
    if(i < ilo+idim){
        const int ijk = i + i*stride + k*kStride;
        xo[ijk] = X(ijk);
    }
    i = ilo +((ilo^j^k^color000)&1) + threadIdx.x*2;
    if(i < ilo+idim){
        const int ijk = i + j*jStride + k*kStride;
        const double Ax = apply_op_ijk();
        const double lambda = Dinv_ijk();
        xo[ijk] = X(ijk) + lambda*(rhs[ijk]-Ax);
    }
}
```

- GSRB_STRIDE2 should have the same FLOPs as GSRB_BRANCH, but same bytes?
- More writes than GSRB_BRANCH?
Code Example 2: HPGMG

**GSRB_BRANCH vs. GSRB_STRIDE2**

- Extra writes in GSRB_STRIDE2 cause more capacity misses in L2, leading to AI drop on L2 and DRAM, starting from Level 7 (data size $\approx$ L2 cache size).
- Runtime almost doubled and GFLOP/s halved.

Roofline captures all of this!
Conclusions

- Roofline can gracefully capture various aspects of application performance and architecture characteristics such as arithmetic intensity, instruction mix, memory coalescing and thread predication.

- The proposed methodology is effective in collecting machine characteristics and application data on NVIDIA GPUs to construct hierarchical Roofline.

- The Roofline model provides insights that profilers alone cannot:
  - identify the most immediate bottleneck
  - prioritize optimization efforts
  - tell you when you can stop

A systematic and intuitive way of code optimization
Reference

- S. Williams, A. Waterman and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009
- Example scripts for plotting Roofline: https://github.com/cyanguwa/nersc-roofline
- General Plasmon Pole kernel: https://github.com/cyanguwa/BerkeleyGW-GPP
- HPGMG-CUDA kernel: https://bitbucket.org/nsakharynh/hpgmg-cuda
Acknowledgement

- 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!