Roofline Performance Analysis with nvprof

Charlene Yang
Application Performance Group, NERSC
Email: cjyang@lbl.gov
Outline

- Use **ERT** to obtain empirical Roofline ceilings
  - compute: FMA, no-FMA
  - bandwidth: system memory, device memory, L2, L1
- Use **nvprof** to obtain application performance
  - FLOPs: active non-predicated threads, divides-aware
  - bytes: read + write; system memory, device memory, L2, L1
  - runtime: --print-gpu-summary, --print-gpu-trace
- Plot Roofline with **Python** and Matplotlib

- **Examples and analysis**
  - GPP from BerkeleyGW: varying AI, FMA, strided memory access
  - HPGMG from Multi-Grid applications: thread divergence
Questions

- Confirm metrics for unified cache, L2, DRAM, system memory
  - Relations with other metrics? texture, constant, etc
  - What exactly do gld/gst\_transactions measure?
  - \text{gst\_transaction} = 0 for GPP?
  - Transaction size 32B for all cache levels? not 128B on L1?
  - L2 data movement < DRAM?
  - Cache bypassing?
  - Formula between metrics and events? NDA?
- Missing L1 in ERT?

- Roofline feature in \text{nvprof}?
- Khaled -- GTC-P, overhead of \text{nvprof}?
Measure Roofline Ceilings
Roofline Ceilings

- **Empirical Roofline Toolkit (ERT)**
- [https://bitbucket.org/berkeleylab/cs-roofline-toolkit/](https://bitbucket.org/berkeleylab/cs-roofline-toolkit/)
- Characterizes machines with **highly tuned** but **real** ‘micro-kernels’
- Sweeps through a variety of configurations:
  - 1 data element per thread -> multiple
  - 1 FLOP operation per data element -> multiple
  - number of threadblocks/threads
  - number of trails, dataset sizes, etc
- Four components
  - Driver.c, Kernel.c, configuration script, and job script
**ERT Configuration**

### Job script

submit the job and run it

```
./ert config.txt
```

**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
```

**Kernel.h**

```
ERT_FLOPS=1: a = b + c
ERT_FLOPS=2: a = a x b + c
```

**config.txt**

```
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
...```

ERT Caveats

• Read-modify-write Polynomial on a vector
  
  - ERT_FLOPS=1: a = b + c;  ERT_FLOPS=2: a = a x b + c;  ..........

• Uses 1:1 Read:Write ratio
  
  - ERT_FLOPS=1: a = b + c
  
  - May underestimate aggregate cache bandwidth on architectures with 2:1 ratio

• May require an unroll-and-jam or large OOO window to hit peak
  
  - #pragma unroll 8

• Labels the largest/slowest bandwidth ‘DRAM’ and the smallest/fastest ‘L1’
  
  - May label L2 as ‘L1’ on architectures with write-through
Peak Bandwidths

- NVIDIA V100, Voltar at Oregon
- ERT_FLOPS=1, GPU_BLOCKS=640, GPU_THREADS=256
- Bandwidth: HBM 828GB/s, L2 3TB/s → These are the peak bandwidths!
- GFLOP/s: 200GFLOP/s → Still in a bandwidth-bound regime
Missing L1 Bandwidth

- Unified cache size is 128KB (L1 data + shared memory) per SM; L2 cache size is 6MB
- Similar size: aggregated L1 size vs L2
- Filling up L1 and L2 at the same time
Peak GFLOP/s

- NVIDIA V100, Voltar at Oregon
- **ERT_FLOPS = 1024**, GPU_BLOCKS = 640, GPU_THREADS = 256
- Bandwidth: HBM 100GB/s → ERT is now in a *compute-bound* regime
- GFLOP/s: **7TFLOP/s** → This is the peak GFLOP/s!
Empirical vs. Theoretical Ceilings

- Empirical ceilings from ERT:
  - compute peak: 7 TFLOP/s, HBM: 828 GB/s, L2: 3 TB/s

- Theoretical compute ceilings on V100:
  - $80 \times 32 \times 2 \times 1.53 \text{ GHz} = 7.83 \text{ TFLOP/s}$

- Theoretical memory bandwidths on V100:
  - HBM: 900 GB/s
  - L2: 4.1 TB/s
  - L1: ~14 TB/s

Application Performance

- Three raw measurements: Runtime, FLOPs, Bytes (on a memory/cache level)

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

- Runtime:
  - time per invocation of a kernel
    \( \text{nvprof} \ --\text{print-gpu-trace} \ ./\text{application} \)
  - average time over multiple invocations
    \( \text{nvprof} \ --\text{print-gpu-summary} \ ./\text{application} \)
  - same kernel with different input parameters are grouped separately
**Application Performance**

- **FLOPs:**
  - predication aware, and divides aware, dp/dp_add/dp_mul/dp_fma, sp*
  - `nvprof --kernels 'kernel_name' --metrics 'flop_count_xx' ./application`

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

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Metrics</th>
<th>Transaction Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td><code>gld_transactions, gst_transactions</code></td>
<td>32B</td>
</tr>
<tr>
<td>L2</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>
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

```bash
./backup-bin/hpgmg-fv-fp 5 8
```

- All metrics at once or one at a time: do they take the same amount of time??

- Output in CSV; Python/Excel for multiple output files

---

**Invocations** | **Metric Name** | **Metric Description** | **Min** | **Max** | **Avg**
---|---|---|---|---|---
Kernel: volt smooth_kernel<int=5, int=32, int=4, int=8>(level_type, int, int, double, double, int, double*, double*)
1 | flop_count_dp | Floating Point Operations(Double Precision) | 30277632 | 30277632 | 30277632
1 | gld_transactions | Global Load Transactions | 4280320 | 4280320 | 4280320
1 | gst_transactions | Global Store Transactions | 73728 | 73728 | 73728
1 | l2_read_transactions | L2 Read Transactions | 896596 | 896596 | 896596
1 | l2_write_transactions | L2 Write Transactions | 85927 | 85927 | 85927
1 | dram_read_transactions | Device Memory Read Transactions | 702911 | 702911 | 702911
1 | dram_write_transactions | Device Memory Write Transactions | 151487 | 151487 | 151487
1 | sysmem_read_bytes | System Memory Read Bytes | 0 | 0 | 0
1 | sysmem_write_bytes | System Memory Write Bytes | 160 | 160 | 160
Plot Roofline
Plot Roofline

- Runtime, FLOPs, Bytes → Arithmetic Intensity, application performance (GFLOP/s)

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

\[
\text{Performance} = \frac{\text{nvprof FLOPs}}{\text{Runtime}}
\]

- Python scripts using Matplotlib
- \url{https://github.com/cyanguwa/nersc-roofline/tree/master/Plotting}
- Simple example: \texttt{plot_roofline.py data.txt}
- Tweaking needed for more sophisticated plotting, see examples
Plot Roofline

- Simple example: `plot_roofline.py data.txt`
- Roofline plot = Compute/Bandwidth ceilings + Two Coordinates per data point
- Accepts space-delimited list for values
- Use quotes to separate names/labels

```plaintext
data.txt

# all data is space delimited
memroofs 828.758
mem_roof_names '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 2.584785579
GFLOPs 2085.756683
labels 'FMA, nw=1'
```
Code Analysis
Code Example 1

- GPP (General Plasmon Pole) kernel from BerkeleyGW (Material Science)
- [https://github.com/cyanguwa/BerkeleyGW-GPP](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

```cpp
  do band = 1, nbands  #threadblocks
    do igp = 1, ngpown
      do iq = 1, ncouls  #threads
        do iw = 1, nw  #unrolled
          compute; reductions
```
Highly parameterizable:

- Varying \texttt{nw} from 1 to 6 to increase arithmetic intensity
  - increasing FLOPs, same HBM data movement
- Striding \texttt{ig} loop to analyze impact of strided memory access
  - Split \texttt{ig} loop to two loops and place the 'blocking' loop outside
- Compile with/without FMA

```c
do band = 1, nbands #threadblocks
  do igp = 1, ngpown
    do igs = 0, stride - 1 #threads
      do ig = 1, ncouls/stride
        do iw = 1, nw #unrolled
          compute; reductions
```

Stride 2
Analysis for GPP

- Effects of varying AI, and FMA/no-FMA
- Appropriate counting of FLOPs for divides
- FLOPs on masked-out threads
- HBM Roofline (i.e. bytes are HBM bytes)
- AI increases as \( nw \) grows
- Bandwidth bound \( \rightarrow \) compute bound
- No-FMA converges to its ceiling
- But FMA doesn’t \((\text{-fmad=true/false})\)

nvprof has taken care of these!
Analysis for GPP

- Hierarchical Roofline

- GPP is more HBM bound than L2/L1 bound at low \(nw\)’s

- L1/L2 performance far from L1/L2 roof

- FLOPs \(\propto nw\)

- HBM bytes: constant

- L2 bytes: increasing at \(\alpha > 1\)

- L1 bytes: constant

- Steep jump in L2 curve at \(nw=2, 3\)
Analysis for GPP

- HBM Roofline (i.e. bytes are HBM bytes)
- Stride size doubles $\rightarrow$ AI halves
- compute bound $\rightarrow$ bandwidth bound
- Cache line 32B; Each complex data 16B
- AI should bottom out at Stride = 2
- But instead Stride = 4
- Prefetching may be in effect
Analysis for GPP

- Hierarchical Roofline

- At fixed \( nw (nw=6) \), striding leads to suboptimal memory coalescing
  - \( L1 \) bytes doubles from stride 1 to stride 2; stays constant after that
  - stride 2 = 16B x 2 = 1 transaction
  - \( L2/DRAM \) AI drops as well

- At Stride = 8, \( L1/L2/DRAM \) performance dots converge to HBM bandwidth
Code Example 2

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

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

Code Example 2

- 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

- Versions: GSRB_FP, GSRB_BRANCH, GSRB_STRIDE2
Code Example 2

- **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); }
}
```

- GS RB/Branch should have half the FLOPs as GS RB_FP, but same HBM/L1/L2 bytes

- Sweep

| 1 0 1 0 1 0 1 0 |

- 8 elements
- 8 threads

| 1 1 1 1 1 |

- 8 elements
- 8 threads
Code Example 2

GSRB_Stride2

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 + j*jStride + 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, and the same bytes
Analysis for HPGMG

**GSRB_FP**
- HBM AI increases as Level 5 → 8
- Due to better surface: volume ratio
- Also more HBM bound
- L1 AI stays constant (roughly)
- FLOPs x 8 when Level +1
- L1 bytes x 8 when Level +1
Analysis for HPGMG

**GSRB_BRANCH**

- Half the FLOPs as GSRB_FP; Same bytes
- Thread predication/divergence
Analysis for HPGMG

**GSRB_STRIDE2**

- L1 AI stays the same;
- Extra stores at Level 7 $\rightarrow$ capacity misses $\rightarrow$ L2/DRAM AI drops
- Striding/Memory coalescing
• Methodology to profile applications on GPUs with Hierarchical Roofline
  – Use ERT to obtain empirical compute/bandwidth peaks
  – Use nvprof to collect FLOPs and Bytes on various memory levels
  – Handy Python scripts at https://github.com/cyanguwa/nersc-roofline

• Hierarchical Roofline is very helpful in understanding performance bounds (compute/bandwidth), analyzing the effects of memory coalescing and thread divergence, and guiding performance optimization efforts.

• Still questions to answer…
Thank You!