Hierarchical Roofline Analysis for GPUs: Accelerating Performance Optimization for the NERSC-9 Perlmutter System

Charlene Yang, Thorsten Kurth
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{*cjyang, tkurth*}*@lbl.gov*

Samuel Williams
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
swwilliams@lbl.gov

***Abstract*—The Roofline performance model provides an intuitive and insightful approach to identifying performance bottlenecks and guiding performance optimization. In preparation for the next-generation supercomputer Perlmutter at NERSC, this paper presents a methodology to construct a hierarchical Roofline on NVIDIA GPUs and extend it to support reduced precision and Tensor Cores. The hierarchical Roofline incorporates L1, L2, device memory and system memory bandwidths into one single figure, and it offers more profound insights into performance analysis than the traditional DRAM-only Roofline. We use our Roofline methodology to analyze three proxy applications: GPP from BerkeleyGW, HPGMG from AMReX, and conv2d from TensorFlow. In so doing, we demonstrate the ability of our methodology to readily understand various aspects of performance and performance bottlenecks on NVIDIA GPUs and motivate code optimizations.**

***Keywords*-Cray, NVIDIA GPU, Roofline, Tensor Core, Performance Analysis, Code Optimization**

I. Introduction

NERSC's next supercomputer Perlmutter will be an NVIDIA GPU-accelerated Cray supercomputer with AMD EPYC host CPUs and an Ethernet-compatible Slingshot network. Although NERSC users are generally familiar with performance optimization on Intel and AMD CPUs, there are a number of new facets of performance optimization on GPUs including thread predication, deep memory hierarchies, mixed precision computation, and Tensor Cores that need to be better understood.

Rather than forcing users to embrace a 'trial-and-error' approach to performance optimization or dig through numerous profiler metrics, the Roofline performance model [1] provides a visually-intuitive way for users to identify performance bottlenecks and motivate code optimization strategies. Roofline is a throughput-oriented performance model centered around the interplay between computational capabilities (e.g. peak GFLOP/s), memory bandwidth (e.g. STREAM GB/s), and data locality (i.e. reuse of data once it is loaded from memory). Data locality is commonly expressed as arithmetic intensity which is the ratio of floating-point operations performed to data movement (FLOPs:Byte). Performance (GFLOP/s) is bound by:

$$\text{GFLOP/s} \leq \min \begin{cases} \text{Peak GFLOP/s} \\ \text{Peak GB/s} \times \text{Arithmetic Intensity} \end{cases} \quad (1)$$

which produces the traditional Roofline formulation when plotted on a log-log plot.

Previously, the Roofline model was expanded to support the full memory hierarchy [2], [3] by adding additional bandwidth "ceilings". Similarly, additional ceilings beneath the Roofline can be added to represent performance bottlenecks arising from lack of vectorization or the failure to exploit fused multiply-add (FMA) instructions.

Orthogonal to the Roofline description of hardware is characterizing applications in terms of Roofline-related coordinates — Performance (GFLOP/s) and Arithmetic Intensity (FLOPs/Byte). One can employ a variety of methods to calculate these terms ranging from hand counting FLOPs and estimating bytes, to performance counters [4], [5], to software simulators [2] that trade performance for accuracy.

Over the last decade, Roofline analysis has been proven a great success especially with the hierarchical Roofline on Intel CPUs [2] and was a benefit to understanding performance on NERSC's previous KNL-based Cori Supercomputer [6], [7]. However, Roofline has yet to be fully developed on NVIDIA GPUs. This paper builds upon the previous work on CPU architectures as well as the HBM-only Roofline methodology we developed for NVIDIA GPUs [8] and expands the model into a hierarchical Roofline methodology that also captures the performance effects associated with reduced precision and Tensor Cores on NVIDIA's latest V100 GPUs.

Our expanded methodology includes:

- Empirical measurement of peak performance (GFLOP/s) and bandwidth (GB/s)
- Accurate measurement of the total number of FLOPs in the code
- Accurate measurement of data movement in the code, throughout the memory/cache hierarchy, i.e. Bytes_{L1}, Bytes_{L2}, $\text{Bytes}_{\text{HBM}}$, $\text{Bytes}_{\text{SystemMemory}}$

- Calculation of arithmetic intensities on various memory/cache levels, i.e. FLOPs:Byte$_{L1}$, FLOPs:Byte$_{L2}$, FLOPs:Byte$_{HBM}$, FLOPs:Byte$_{SystemMemory}$
- Quantifying the performance implications of FMA, FPADD, and FPMUL in the instruction mix
- Quantifying the performance implications of reduced precision (FP16 and FP32) and Tensor Cores, and
- Plotting application performance against architecture peaks

In this paper, we provide a detailed description of our Roofline methodology for NVIDIA GPUs. We then apply this methodology to three proxy applications, GPP from the Material Science code BerkeleyGW [9], HPGMG from the Adaptive Mesh Refinement framework AMReX [10], and conv2d from TensorFlow [11]. For each of these applications, we include multiple variants of the same code in order to highlight the ability of our methodology to capture the different nuances of performance analysis on NVIDIA GPUs. Throughout this process we provide a detailed analysis of the information our Roofline methodology extracts. Finally, we conclude the paper with some high-level insights, observations, and espouse several directions for future work.

II. Roofline Methodology on NVIDIA GPUs

In order to affect Roofline analysis of GPU-accelerated applications, one must perform three steps. First, one must characterize the underlying GPU's computational capabilities in terms of this Roofline model. In effect, this is measuring peak performance (GFLOP/s) and bandwidth (GB/s) as a function of data precision, operation, and memory/cache level. Second, one must characterize the execution of an application and extract the relevant Roofline-related parameters including data movement at each level of the memory hierarchy, floating-point operations performed (by precision), and kernel run times. Finally, one must synthesize these two data sets together and plot them in a single figure.

A. Architectural Characterization

The Empirical Roofline Toolkit (ERT) [12] was developed to characterize multicore, manycore, and GPU-accelerated systems. It was written in MPI+OpenMP+CUDA in order to replicate the most common programming environments on DOE (Department of Energy) supercomputers. ERT defines a kernel of varying L1 arithmetic intensity on a parameterized vector. By sweeping a range of arithmetic intensities and vector sizes, it can extract the peak performance of a target platform as well as the bandwidth at each level of the memory hierarchy. In this paper, we used the MPI+CUDA implementation of ERT to characterize a single Volta V100 GPU. Unfortunately, ERT, as written, consistently fails to identify the L1 cache on NVIDIA GPUs. To that end, throughout this paper, we use a theoretical L1 bandwidth coupled with empirical (ERT) bandwidths for L2 and HBM, for the Roofline ceilings.

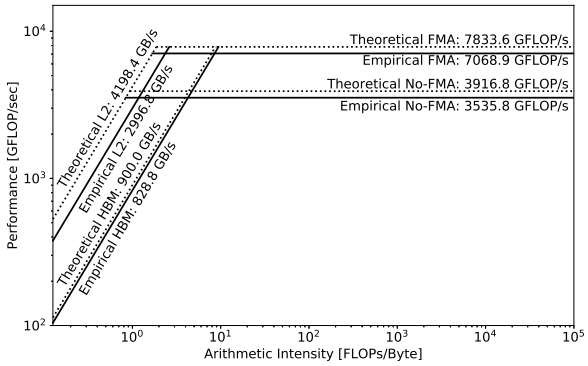

Figure 1. NVIDIA V100 Hierarchical Roofline Ceilings. Observe V100 advertised performance is very close to empirical performance.

ERT, as written, is solely a double-precision benchmark. As such, in this paper, we use a simple linear extrapolation for single- and half-precision performance. Moreover, whereas ERT's kernels are optimized for fused-multiply-add (FMA) instruction-set architectures, we estimate the penalty of not exploiting FMA by defining a "no FMA" ceiling that is half the FMA performance. NVIDIA GPUs implement 16-bit (FP16) Tensor Core matrix-matrix multiplications. Throughout this paper, we use the theoretical peak Tensor Core performance. This may seem optimistic, but we will show it does not skew our analysis.

Ultimately, we collect 10 performance numbers for our target GPU: L1, L2, and HBM bandwidth, FP16/FP32/FP64 FMA and FP16/FP32/FP64 "no FMA" performance, and Tensor Core peak performance. For brevity, Figure 1 plots measured ERT and theoretical performance for FP64 FMA, no-FMA, L2, and HBM on an NVIDIA Volta V100 GPU. Clearly, theoretical performance generally overestimates attainable performance by about 10%.

B. Application Characterization

In this paper, we leverage the proof of concept methodology developed by Yang et al. [8] and extend it to support both hierarchical (L1, L2, HBM, System Memory) Roofline analysis as well as FP32 and FP16 precision (including Tensor Core). To that end, we use `nvprof` to collect a set of metrics for each kernel in an application. We then synthesize those metrics together in order to plot each kernel on a Roofline using its Arithmetic Intensity (x) and GFLOP/s (y) coordinates.

In order to calculate a kernel's arithmetic intensity (AI) and GFLOP/s performance, we must collect three raw quantities — kernel run time, FLOPs executed (for FP64, FP32, and FP16), and bytes read and written by each level of the memory hierarchy (L1, L2, HBM and System Memory).

$$AI_{<precision>,<level>} = \frac{nvprof\ FLOPs_{<precision>}}{nvprof\ Bytes_{<level>}} \quad (2)$$

$$FLOP/s_{<precision>} = \frac{nvprof\ FLOPs_{<precision>}}{nvprof\ Run\ Time} \quad (3)$$

where <*level*> can be L1, L2, HBM (Device Memory) or System Memory, and <*precision*> can be FP64, FP32, FP16, or Tensor Core.

Kernel Run Time: To collect application run time, we use the following commands to obtain either the timing of a particular invocation of a kernel or the average timing of a kernel over multiple invocations.

nvprof --print-gpu-trace ./application

nvprof --print-gpu-summary ./application

Kernel FLOPs: `nvprof` provides a rich set of metrics to measure the total number of FLOPs executed in a kernel. These metrics only account for non-predicated threads so operations that are masked out, are not included. For complex operations such as divides, logarithms and exponentials, each operation is implemented with multiple instructions and hence is counted as multiple FLOPs. To collect the FLOP counts, we use:

nvprof --kernels < kernel_name >
--metrics < metric_name > ./application

where <*metric_name*> can be `flop_count_dp`, `flop_count_sp`, or `flop_count_hp` for FP64, FP32, and FP16 respectively.

The aforementioned floating-point metrics can account for the majority of FLOPs in a large range of applications. However, FLOPs executed inside the NVIDIA V100 Tensor Cores are not captured by these counters. The Tensor Cores are designed to accelerated matrix-FMA operations, that is, operations of the form $D{=}A{\cdot}B{+}C$ where A, B are real valued 4×4 matrices in half precision (FP16) and C, D are real valued 4×4 matrices in 16-bit (FP16) or 32-bit (FP32) precision. In the latter case, the accumulator in the Tensor Core operation is performed using FP32 arithmetic.

As of early 2019, `nvprof` does not offer an accurate `flop_count_` metric for Tensor Cores, like for the normal SM cores, but rather a "utilization" metric `tensor_precision_fu_utilization`. This metric spans an integer range from 0 (not used) to 10 (fully utilized). In order to estimate the Tensor Core FLOP count, we assume that a utilization value of 10 corresponds to 125 TFLOP/s and then multiply this number with the run time of the kernel to estimate the total number of FLOPs. It is expected that NVIDIA's next-generation Nsight profiling tool will have enhanced capabilities to measure Tensor Core FLOPs and we will investigate that in our future work.

Bytes: The data moved between each two levels in the memory/cache hierarchy must be collected in order to construct the hierarchical Roofline. We use `nvprof` to collect the total number of read and write transactions and multiply the total by the size of each transaction in bytes.

$$Bytes = (read\ transactions + write\ transactions) \times transaction\ size \quad (4)$$

The invocation of `nvprof` on command line is the same as when collecting FLOPs, but the metrics are more complicated (see Table I). Note, in this paper, all applications fit in the GPU's HBM memory. As such, system transactions are virtually zero as there is no data movement over PCIe/NVLINK. Thus, they will not be presented in Section IV.

Level	Metrics	Transaction Size

L1 Cache	`gld_transactions`
	`gst_transactions`
	`atomic_transactions`
	`local_load_transactions`
	`local_store_transactions`
	`shared_load_transactions`
	`shared_store_transactions`
		32B

L2 Cache	`l2_read_transactions` `l2_write_transactions`	32B
HBM Memory	`dram_read_transactions` `dram_write_transactions`	32B
PCIe/NVLINK	`system_read_transactions` `system_write_transactions`	32B

Table I
NVPROF METRICS FOR MEASURING DATA TRAFFIC IN THE MEMORY/CACHE HIERARCHY[1]

C. Roofline Visualization

With both the empirical ceilings collected via ERT and the application kernel AI (2) and GFLOP/s (3) coordinates determined through the collection of `nvprof` metrics, we plot the resultant Roofline model using Python Matplotlib [13]. Some of our Matplotlib scripts are available on GitHub [14] and users are free to tweak them based on their specific needs. The most basic example `plot_roofline.py` takes an input file that specifies the memory ceilings, compute ceilings, AI's for each kernel and GFLOP/s performance for each kernel.

III. Experimental Setup

In this section, we describe our test machine, software configuration, and the applications we use to evaluate our hierarchical GPU Roofline methodology.

A. Hardware and Software Configuration

Results presented in this paper were all obtained on the Cori supercomputer at the National Energy Research Scientific Center (NERSC), Lawrence Berkeley National

[1]Add surface and texture related metrics if surface and texture memory are in use, for example, in Graphics applications.

Laboratory (LBNL). To prepare for the arrival of the next-generation supercomputer Perlmutter, NERSC has installed a GPU-accelerated partition on Cori, comprised of nodes with Intel Skylake CPUs and NVIDIA V100 GPUs (4:1 GPU:CPU ratio). This partition will enable the NESAP (NERSC Exascale Science Applications Program) teams in their prototyping, debugging, porting, and development activities in preparation for migrating to Perlmutter.

CUDA 10 is installed on Cori GPU partition and `nvprof`, frequently used in this paper, is part of the CUDA Toolkit. Additionally, ERT is from the BitBucket repository [12] with example configuration scripts deployed for the Cori GPU partition. For the conv2d benchmark from TensorFlow, we used TensorFlow v1.12.0 linked against CUDA 9.0 and cuDNN 7.3.1.

B. Benchmarks

In this paper, we evaluate our Roofline methodology for NVIDIA GPUs using three benchmarks: GPP from BerkeleyGW, HPGMG from AMRex, and conv2d from TensorFlow. These benchmarks were selected because they exhibit a range of computational characteristics including a range of memory access patterns, data types, data locality, and thread divergence properties.

GPP: The General Plasmon Pole (GPP) kernel [9] is a proxy application based on the BerkeleyGW Material Science code [15]. It calculates electron self-energy using the common General Plasmon Pole approximation [16]. GPP is written in C++ and accelerated with CUDA. The computation in the kernel is tensor-contraction like, wherein a few pre-calculated complex double-precision arrays are multiplied and summed over one dimension and collapsed into a small matrix. The problem we chose in this paper is comprised of 512 electrons and 32768 plane wave basis elements, and is a medium sized problem for real-world materials science. It requires around 1.5GB of memory and fits well into the HBM memory on a V100 GPU.

The pseudo code of the GPP kernel can be described as

```
do band = 1, nbands #threadblocks Idx.x
 do igp = 1, ngpown #threadblocks Idx.y
  do ig = 1, ncouls #threads Idx.x
   do iw = 1, nw    #unrolled
     load wtilde_array(ig,igp)
     load aqsntemp(ig,band)
     load eps(ig,igp)
     compute wdiff, delw, sch_array
   update achtemp(iw)
```

Not only does GPP offer abundant parallelism that can be mapped to threads, warps, and thread blocks, but it is also heavily parameterized in order to capture the full spectrum of realistic problem configurations. One of these parameters, nw, enables arbitrary increases in arithmetic intensity by increasing reuse of the arrays accessed within the iw loop. Similarly, we can modify the ig loop to affect strided memory accesses. Ultimately, not only does this single, well-understood kernel act as a stand-in for a range of potential application kernels, but it also enables us to test the limits of our Roofline methodology for NVIDIA GPUs.

HPGMG GSRB Smoother: HPGMG [10], [17], [18] is a geometric multigrid benchmark designed to proxy the multigrid solves found in block structured AMR (Adaptive Mesh Refinement) applications that use the AMReX framework [19]. HPGMG solves the 4^{th} order, variable-coefficient Laplacian on a unit-cube with Dirichlet boundary conditions using a multigrid F-cycle. As such, it is only moderately compute intensive (FP64 FLOPs:Byte$>$1), but is highly demanding of SIMT and cache locality.

HPGMG was originally implemented in C with MPI and OpenMP parallelization and has shown scalability to 8.5 million cores. It was subsequently extended to support GPUs on the finer (larger) mesh levels [10].

The Gauss-Seidel, Red-Black (GSRB) smoother generally dominates HPGMG's run time. However, the performance of the smoother varies substantially among the various multigrid levels (mesh sizes). GSRB smoothers perform two stencil kernel invocations per smooth (red and black). Cells are marked as either red or black in a 3D checkerboard pattern. Cells matching the sweep color are updated, while the others are simply copied to the result array. Thus, for a 128^3 box, one must perform $128^3/2$ =1M stencils per kernel invocation.

In order to balance the quality of a compiler against hardware's ability to efficiently execute strided memory access patterns or predication, HPGMG includes three different implementations of its GSRB smoother (`GSRB_FP`, `GSRB_BRANCH`, and `GSRB_STRIDE2`). All three implementations perform the same computation and touch the same data over the course of a threadblock's execution. As such, they represent an ideal testbed for using Roofline and performance tools to understand the subtle interactions between compiler and hardware.

The `GSRB_FP` implementation realizes red-black updates via multiplication by a precomputed auxiliary array of 1.0's and 0.0's. Such an implementation is trivially vectorized, but requires twice the computation and an additional load from cache.

The `GSRB_BRANCH` version uses a branch to perform stencils only on cells whose color matches the sweep while copying data for the others. Such an implementation can be realized through either predication (masking) or loop fissioning into two stride-2 updates. Regardless, the number of floating-point operations is minimized, but execution on SIMD or SIMT hardware may preclude this.

Finally, in the `GSRB_STRIDE2` implementation, each CUDA thread is responsible for two adjacent cells within a plane. The thread updates the cell whose color matches the sweep color and copies the other. This implementation min-

imizes computation, avoids predication of computation, but requires the compiler/hardware to execute efficient stride-2 L1 cache accesses (HBM access will always be coalesced into unit-stride transactions).

For this paper, we run HPGMG with eight 128^3 boxes (`./hpgmg-fv 7 8`). This results in levels 5-8 running on GPU and levels 1-4 on CPU. As this paper is focused on Roofline on GPUs, we only examine levels 5-8.

TensorFlow conv2d: TensorFlow [11], [20] is a deep learning framework that allows users to express complicated neural network graphs in a reasonable amount of lines of code. Besides flexibility, it also provides performance portability across a variety of computing architectures by using optimized libraries such as cuDNN. 2D convolution layers are the most compute intensive kernels in most modern deep neural networks, so in this paper we chose to analyze `tf.nn.conv2d` [21] for our Roofline validation.

In a *forward* pass, `tf.nn.conv2d` performs 2D convolution of an input tensor and a convolution kernel, and produces another tensor as the output. Assume an input tensor A of shape $N \times H \times W \times C$, where N is the number of samples in the batch, H and W are the height and width, and C is the number of channels. With a convolution kernel K of shape K_H, K_W, C, C', the resultant tensor B is of shape $N \times H' \times W' \times C'$, where $H' = H - K_H + 1$, $W' = W - K_W + 1$, and the individual elements are given by

$$B_{nhwc} = \sum_{m=0}^{C-1} \sum_{k_h=0}^{K_H-1} \sum_{k_w=0}^{K_W-1} A_{n\, h+k_h\, w+w_h\, m}\, K_{k_h\, k_w\, m\, c} \quad (5)$$

In the more computationally expensive *backward* pass, the derivative of Equation 5 is computed with respect to K. The TensorFlow routine further allows for generalizations such as non-unit stride or dilation. Values at the image boundary are treated separately, e.g. by replication or zero-padding [22]. In this study, we will focus on the performance characteristics of a typical convolution kernel found in one of the most commonly used networks for image analysis, ResNet50 [23]. We analyze this kernel for forward and backward passes as well as the effects of parameters such as batch, input, kernel and stride sizes on their performance.

Most TensorFlow routines support specifying different precisions via a `dtype` argument. For deep learning applications, the most relevant are FP16 and FP32, and we will focus on these two precisions in this paper. The convolution operations such as those in Equation 5 are essentially matrix-matrix multiplications, and on NVIDIA's Volta GPUs, TensorFlow can leverage the specialized instructions such as HMMA's on the Tensor Cores for more performance. Due to this GEMM-like operation, the conv2d kernel will have a much higher arithmetic intensity than our previous two examples, GPP and HPGMG.

The pseudo code of the conv2d kernel is as follows.

```
#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)

#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.Gradient\
                  DescentOptimizer(0.5)
        exec_op = opt.compute\
                  _gradients(output_result)
elif pass=="calibrate":
    with tf.device(gpu_dev):
        exec_op = input_image
...

with tf.Session(config=...) as sess:
    ...

    #warm-up
    for i in range(n_warm):
        result = sess.run(exec_op)

    #measurement
    pyc.driver.start_profiler()
    for i in range(n_iter):
        result = sess.run(exec_op)
    pyc.driver.stop_profiler()
```

The main measurement part is towards the end of this code block (the last four lines). The `pyc.driver` commands are from PyCUDA [24] and they allow for precise instrumentation on this region. To facilitate this, `nvprof` will also need to be launched with `--profile-from-start off`, to disable profiling from the start of the application.

Measuring the performance of a TensorFlow kernel is not straightforward. First, every kernel can translate into a series of subkernel calls. These subkernels include the kernel that does the essential computation but also kernels that perform housekeeping work before and after the main kernel, such as data layout transformation, data type transformation and index calculation. Second, the autotuning mechanism in TensorFlow and cuDNN library adds to the complexity of the measurement of a kernel's performance. TensorFlow selects the best performing sequence of subkernels based on input parameters and some heuristic data, and the cuDNN

library, frequently called by TensorFlow, also performs certain algorithm selection.

In this paper, we include the housekeeping subkernels in our performance measurement of conv2d, as they are necessary to the core computation subkernel, but we exclude the autotuning subkernels, because we want to focus on the best implementation of the conv2d kernel for given parameters. To achieve this, we split the loop in `tf.Session` into two parts: a warm-up part with trip count n_warm=5, and a measurement part with trip count n_iter=20. Depending on the value of *pass*, `exec_op` will either compute the convolution (forward pass), or the convolution along with its derivatives (backward pass). The extra `calibrate` option allows for exclusion of subkernels that are associated with random input tensor generation, and we would like to exclude that in our measurement as well.

In Section IV, we use the sum of the following three measurements as the FLOP count of conv2d, when calculating Arithmetic Intensity and the GFLOP/s performance, `flop_count_sp`, `flop_count_hp`, and FLOPs derived from `tensor_precision_fu_utilization` (see Section II-B). Here we include the `flop_count_sp` metric (metric for FP32 operations) even in the FP16 kernels, because even if the input and output tensors are set to be in FP16 precision, TensorFlow could still decide to deploy subkernels that are in FP32 precision, based on its autotuning mechanism. An example of this is the backward pass in Figure 10. However, all the other kernels in Figure 8, 9 and 10, execute as expected, i.e. FP16 kernels are run on the Tensor Core and FP32 kernels are run on the common cores.

In this paper, if not otherwise stated, we use input tensor size 112×112×64 and kernel size 3×3×64×C', where C' is the number of output filters as defined in Equation 5, C'=64, and the stride size is 2.

IV. Results

In this section we discuss our observations and insights from applying our Roofline methodology for NVIDIA GPUs to our three benchmarks.

A. GPP Performance Analysis

Figure 2 presents a hierarchical Roofline model for GPP running on a V100 GPU as a function of the parameter nw. Recall, nw increases arithmetic intensity at all levels of the memory hierarchy as it creates a tighter inner loop that reuses data loaded from the cache. We observe several effects.

First, both L1 and HBM arithmetic intensity (x coordinate of each dot) increase linearly with nw. Interestingly L2 intensity shows a much more complex pattern. Linear increases in arithmetic intensity for kernels that are linearly increasing the number of FLOPs (numerator of

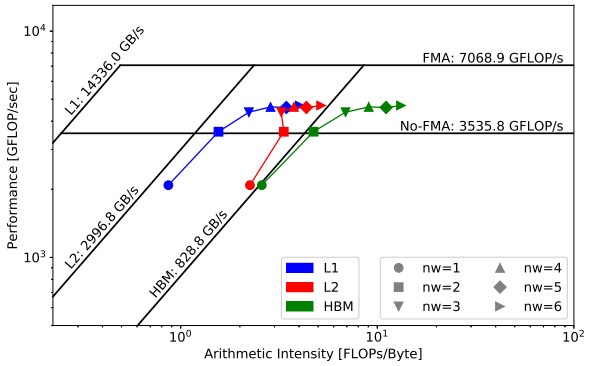

Figure 2. GPP hierarchical Roofline on the NVIDIA V100 GPU as a function of nw (stride=1).

arithmetic intensity) implies roughly constant data movement (denominator of arithmetic intensity). As such, we can infer very good locality in the register file (constant L1 intensity) as well as good locality in the L2 (constant HBM intensity). However, the only slight improvement in L2 intensity implies substantial increases in L2 data movement or substantial losses in L1 locality.

Second, HBM intensity is consistently much larger than L1 intensity implying there is substantially higher locality in the caches than in the register file. Moreover, L2 intensity is initially ($nw = 1$) very close to HBM intensity, but it approaches L1 intensity as nw approaches 6. This is indicative of a transition from a regime where there is high L1 locality and virtually no L2 locality ($nw = 1$) to a regime where there is virtually no L1 locality but high L2 locality ($nw = 6$).

Third, at low nw, performance is clearly bound by HBM (green curve tracks the HBM ceiling). However, as nw increases, performance quickly saturates. Our hierarchical Roofline analysis demonstrates GPP is clearly not bound by either L1 or L2 bandwidth (red and blue curves are far from their respective ceilings). This indicates other effects have manifested that limit performance.

Unlike linear algebra routines (e.g. matrix-multiplications), GPP includes a mix of floating-point adds, multiplies, and fused multiply-adds (FMA). As 100% of the dynamic instructions are not FMA, peak performance will never be attainable.

In fact, we can create an effective ceiling by using `nvprof` to collect the number of FMA and non-FMA floating-point instructions. Using Equation 6, it is observed that at $nw = 6$, only 60% of the floating-point instructions GPP executes are FMA. Moreover, Equation 7 bounds GPP performance at 80% of the V100's (double-precision) FMA peak performance. However, Figure 3 shows that the observed performance from GPP is roughly 66% of the full FMA peak. This clearly indicates that other aspects of GPU

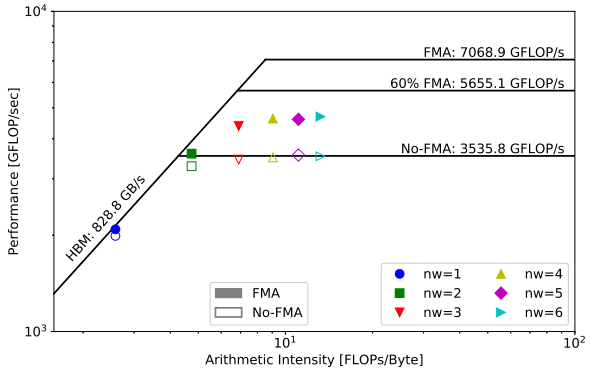

Figure 3. GPP Roofline on the NVIDIA V100 GPU as a function of the FMA instruction mix. Note, 60% of GPP's floating-point instructions are FMA.

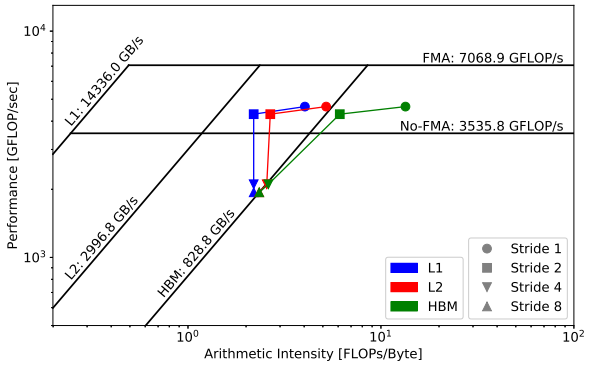

Figure 4. GPP hierarchical Roofline on the NVIDIA V100 GPU as a function of stride and $nw = 6$

execution are ultimately limiting performance.

$$\alpha = \frac{\textit{FMA FP64 instr.}}{\textit{FMA FP64 instr. + non-FMA FP64 instr.}} = 60\% \quad (6)$$

$$\beta = \frac{\alpha \times 2 + (1 - \alpha)}{2} = 80\% \quad (7)$$

As mentioned, GPP is highly parameterizable. To that end, Figure 4 shows a Roofline model for the strided implementation of GPP. Here threads within a warp update access every n^{th} element (threads stride by $32 * n$ words instead of the nominal Stride-32). Unlike our previous work [8] which focused solely on HBM Rooflines for GPUs, the GPP hierarchical Roofline shows that the L1 and L2 cache behave quite differently from HBM. Whereas HBM intensity decreases linearly with increasing stride up to Stride-4 (4 double complex words = 64 Bytes), L1 and L2 intensity stops decreasing beyond Stride-2 (32B). One might conclude the cache line size in the L2 (or at least its behavior) is larger than the L1 line size or the L1 transaction size.

B. HPGMG Performance Analysis

We evaluate HPGMG's three implementations of its GSRB smoother using the hierarchical Roofline model. As all variants are memory-intensive (arithmetic intensity is always less than machine balance), unlike GPP, the FMA fraction of the instruction mix will play no role in our analysis. Moreover, in lieu of an algorithmic parameter to affect changes in arithmetic intensity, in all of our analysis, we examine performance on each level in the multigrid hierarchy HPGMG that runs on the GPU. Naively, one might think the same stencil should have the same intensity. However, the deep ghost zone can reduce arithmetic intensity for the smaller boxes (lower levels) while cache effects can manifest and reduce intensity for the larger boxes (upper levels).

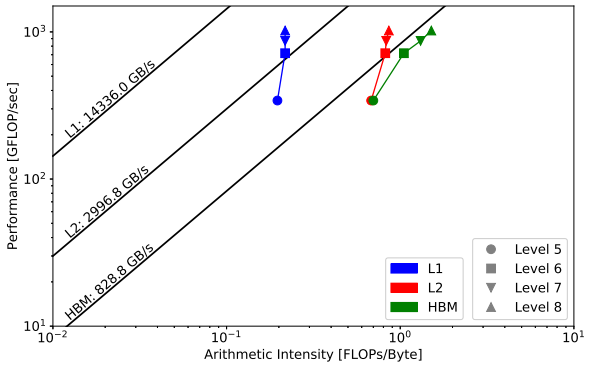

Figure 5. HPGMG GSRB hierarchical Roofline on the NVIDIA V100 GPU for the `GSRB_FP` implementation.

Figure 5 presents the hierarchical Roofline for the `GSRB_FP` variant of HPGMG's smoother as a function of level (Level 5 with 8 16^3 boxes to Level 8 with 8 128^3 boxes). Roofline clearly provides several immediate observations. First, performance is highly correlated with HBM bandwidth (green line tracks the HBM ceiling). Second, HBM intensity increases with level. This should come as no surprise as intensity should scale as $O(dim^3/(dim+4)^3)$ as there is a ghost zone (2 elements deep) on the high and low faces of each box. This substantially reduces intensity for small boxes ($dim = 16$). Third, L1 intensity is roughly constant with box size. Once again, this should come as no surprise as internally, the CUDA implementation uses a fixed thread block dimension to tile each box. The constant thread block dimension exerts a constant pressure on the L1 cache. Fourth, there is substantial reuse in the L1 cache (L1(blue) and L2(red) intensities are widely separated) while there is virtually no reuse in the L2 cache (L2(red) and HBM(blue) intensities are very close). This implies that virtually all reuse in HPGMG is captured by the L1 cache

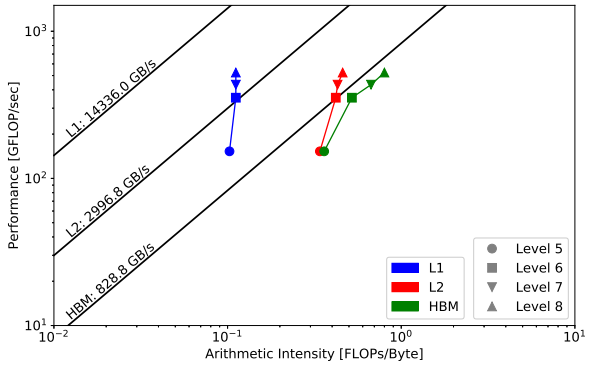

Figure 6. HPGMG GSRB hierarchical Roofline on the NVIDIA V100 GPU for the `GSRB_BRANCH` implementation.

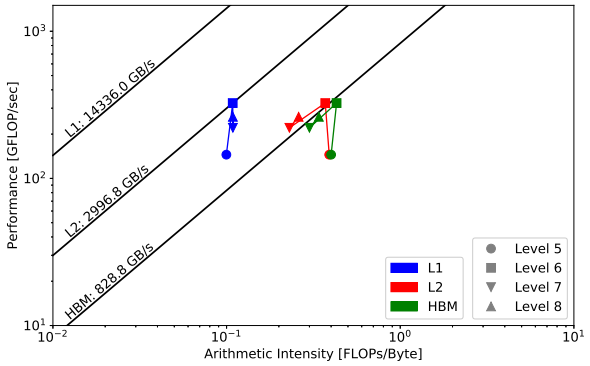

Figure 7. HPGMG GSRB hierarchical Roofline on the NVIDIA V100 GPU for the `GSRB_STRIDE2` implementation. Observe the unexpected loss in L2 and HBM arithmetic intensity for the larger levels.

or in register reuse and there is very little inter-thread block bandwidth filtering (something expected for tiled stencil computations). Finally, the very astute will notice that the empirical HBM arithmetic intensity is twice the theoretical HPGMG intensity. This is an artifact of the `GSRB_FP` implementation redundantly performing the stencil on every point and quashing the results by multiplying by the array of 1's and 0's — something `nvprof` dutifully observes.

Figure 6 presents the hierarchical Roofline for the `GSRB_BRANCH` implementation. Recall, this implementation differs from the GSRB_FP implementation in that it uses optimized modulo-2 arithmetic and a branch to avoid redundant computation and an extra (L1) load. As such, it performs half as many (non-predicated) floating-point operations, and thus has half the performance and half the arithmetic intensity on each level of multigrid and each level of the memory hierarchy as the `GSRB_FP` variant shown in Figure 5. All analysis and insights derived from `GSRB_FP` apply to `GSRB_BRANCH`.

On paper, HPGMG's `GSRB_STRIDE2` implementation seems like the ideal implementation. It performs no redundant work, all computation remains converged/non-predicated, and the stride-2 memory access pattern presented to the L1 should be filtered into a unit-stride pattern presented to the L2/HBM. As such, it can be quite puzzling as to why the `GSRB_STRIDE2` implementation underperforms the `GSRB_BRANCH` and `GSRB_FP` variants. Our `nvprof`-based hierarchical Roofline model helps elucidate the causes.

Figure 7 shows the hierarchical Roofline for the `GSRB_STRIDE2` variant. It should be immediately obvious that it looks quite different from the `GSRB_BRANCH` version shown in Figure 6 with performance on the largest boxes (those that dominate HPGMG's solve time) substantially lower. First, the trend in L1 intensity (x-coordinate) is very similar to `GSRB_BRANCH`. This indicates that as expected, each thread block, accesses memory in a similar manner to the `GSRB_BRANCH` variant. However, when looking at L2 and HBM intensity, we observe very different behaviors. As one proceeds from level 5 (16^3) to level 6 (32^3), one observes increases in performance (ultimately bound by HBM bandwidth), but only slight increases in arithmetic intensity. Conversely, from level 6 to level 7 (64^3) and level 8 (128^3), we see substantial reductions in L2 and HBM intensity. The former implies that unlike the `GSRB_BRANCH` variant, the `GSRB_STRIDE2` variant is failing to capture locality in the the L1 cache and flooding L2 with additional data movement. The fact that HBM intensity is correlated with L2 intensity implies that the L2 is also failing to capture any locality and transactions received by the L2 are passing through and becoming increased HBM data movement. Increasing data movement when HBM-bound results in performance sliding down along the HBM Roofline.

C. TensorFlow conv2d Performance Analysis

In this section we investigate the effects of different input parameters on the arithmetic intensity and performance of the TensorFlow conv2d kernel. More precisely, we start with the baseline parameters described in Section III-B and then vary one parameter at a time. The parameters we examine are *batch size*, *number of output filters* and the *kernel (or filter) size*. The two data types, FP32 and FP16, are the precisions of the input and output data, but not necessarily those of the FLOPs or bytes measured, i.e. TensorFlow may decide to execute in FP32 on FP16 inputs.

Batch Size: Figure 8 depicts the impact of the batch size on performance (the batch size is the parameter N in Equation 5). For FP32 kernels, i.e. when both input and output tensors are in FP32 type, the arithmetic intensity and GFLOP/s performance of conv2d don't change much, simply because the underlying algorithm is the same. There is an exception though, with the backward pass where TensorFlow has decided to call a different `wgrad` subkernel from

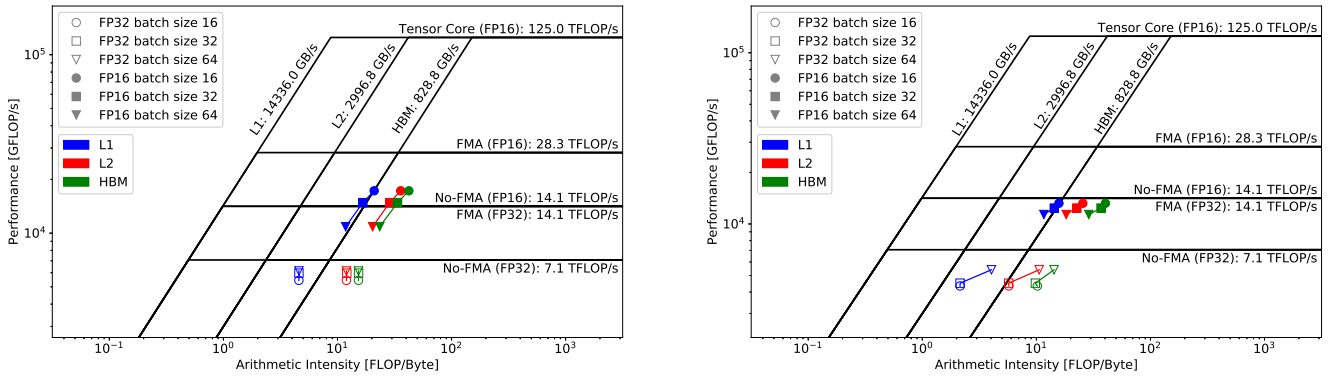

Figure 8. Effects of batch size on performance for forward (left panel) and backward pass (right panel) on second convolution layer from the ResNet50 [23] network. Open symbols represent kernels with FP32 input and output, and filled symbols represent FP16.

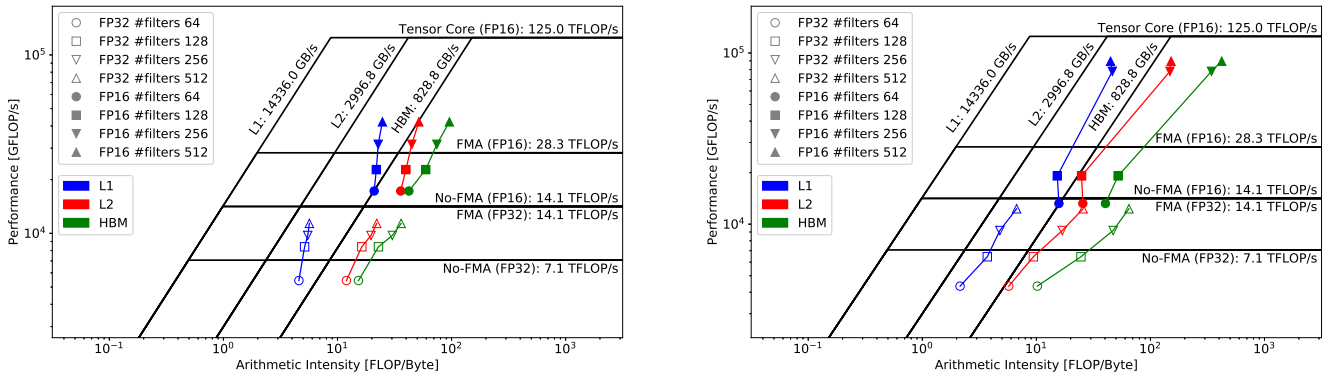

Figure 9. Effects of the number of output filters on performance for forward (left panel) and backward pass (right panel) on second convolution layer from the ResNet50 [23] network. Open symbols represent kernels with FP32 input and output, and filled symbols represent FP16.

cuDNN for batch size 64, which has raised the performance a little bit compared to the other two batch sizes.

The FP16 kernels should follow the same trend, i.e. same intensity and performance for all batch sizes. However, in the raw data, we observe a significant increase in the data movement for larger batch sizes, and because we include housekeeping subkernels such as padding and shuffling in our measurement, the performance of this fairly small kernel is severely affected by these essentially bandwidth-bound subkernels. In the next experiment (Number of Output Filters), where kernels are larger, this effect may be better amortized. However, for TensorFlow applications in general, this could be one of the reasons why the peak performance of 125 TFLOP/s is very hard to reach.

The cuDNN library is called very frequently in TensorFlow and it is observed in our raw data that cuDNN utilizes the shared memory on Volta a lot. On the hierarchical Roofline charts, this is presented by the gaps between the L1 symbols and their respective L2 symbols, i.e. these conv2d kernels have good cache locality in level-one cache (including shared memory).

Number of Output Filters: Figure 9 shows the hierarchical Roofline for conv2d when the number of output filters increases, i.e. C' in Equation 5. The batch size for this set of results is fixed at 16. In both FP16 and FP32 cases, and both forward and backward passes, the arithmetic intensity and the GFLOP/s performance increase with the number of filters because the kernel becomes more compute intensive, i.e. more computation is done for the same amount of data movement. At the highest, the FP16 kernel in the backward pass is reaching 80% of the peak (at about 100 TFLOP/s), with 512 filters. In the meantime, the FP32 kernel gets even closer to the FMA(FP32) peak, at about 13 TFLOP/s. All of this shows great promise of achieving Volta's compute capability with certain input parameters.

Kernel Size: Figure 10 shows the hierarchical Roofline of conv2d as a function of the kernel size, i.e. $K_H \times K_W$, from 3×3, to 7×7, to 9×9. The batch size here is fixed at 16, and the number of output filters is 64. The stride size is 2.

The increase in kernel size should have the same effect as the number of filters, i.e. increased computation for the

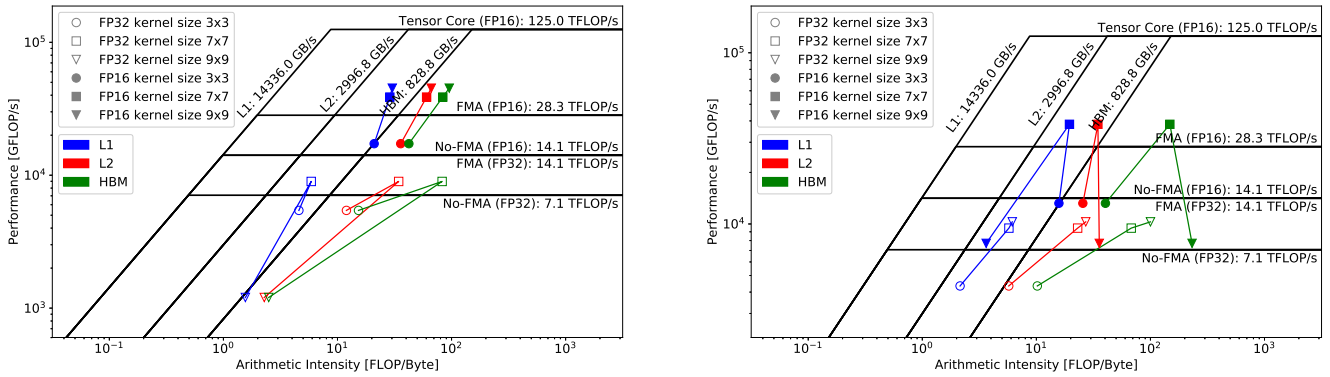

Figure 10. Effects of kernel size on performance for forward (left panel) and backward pass (right panel) on second convolution layer from the ResNet50 [23] network. Open symbols represent kernels with FP32 input and output, and filled symbols represent FP16.

same amount of data movement, hence increased arithmetic intensity and performance. However, there are two exceptions. One is the sudden drop in both arithmetic intensity and performance at kernel size 9×9 in the forward pass for the FP32 input and output. In this case, the TensorFlow framework decides to run a different set of subkernels – instead of the `wgrad` subkernels for the 3×3 and 7×7 kernel sizes, it calls FFT subkernels underneath. This could be due to mistakes in the autotuning decision-making process, where 9×9 seems large enough but it is still not at the level where FFT could be run optimally. This phenomena is not observed in the backward pass case, which suggests there is certain sensitivity in TensorFlow's autotuning mechanism to the heuristic data possibly from the warm-up stage.

The other exception is with the FP16 kernel in the backward pass at kernel size 9×9. Even though the input and output tensors are specified to be FP16, the kernel is run in FP32 precision, i.e. not on Tensor Cores. The input data is first converted to FP32, then the FP32 subkernels are executed, and finally the output is converted back to FP16. These unnecessary conversions lead to the overall kernel performance being even worse than the natual FP32 kernel (with FP32 intput and output, open triangles in Figure 10, right panel). It also suggests that the robustness of the autotuning mechanism in TensorFlow could be improved.

V. Summary, Conclusions, and Future Work

In this paper, we extend the `nvprof`-based HBM Roofline methodology we developed for NVIDIA GPUs [8] to capture the full NVIDIA GPU memory hierarchy, the effects of FPADD/FPMUL in the instruction mix, the effects of reduced precision FP16 and FP32 Rooflines, and the benefits of using FP16 Tensor Cores (HMMA instructions). To demonstrate the value of this hierarchical GPU Roofline methodology, we used it to analyze three benchmarks: the moderately compute intensive GPP Material Science proxy application, the cache-intensive HPGMG AMR-multigrid proxy application, and the reduced precision and a Tensor Core-accelerated 2D convolution kernel from TensorFlow.

We observe that the hierarchical Roofline can capture insights into compute, cache, or memory performance bottlenecks as well as properties of locality within each level of the cache hierarchy with performance being highly correlated with Roofline in the memory-intensive GPP and HPGMG benchmarks. However, there were several cases in both HPGMG and TensorFlow where empirical performance and arithmetic intensity diverged from Roofline or theoretical expectations. Similarly, ultimate performance of GPP for high nw was only roughly correlated with the "partial" FMA performance ceiling derived from the FMA fraction of the instruction mix. We found that although Roofline provides observations of performance metrics (e.g. decreased arithmetic intensity), it does not inform users as to exactly what went wrong in their application's execution or the code changes required to fix it. Nevertheless, it does provide some key first steps in potentially identifying areas of interest and may motivate further experiments.

In the future, we will extend our Roofline methodology and usage along three axes — more ceilings, instruction-based Rooflines. As for the first axis, we see several potential extensions to Roofline. In lieu of simply scaling performance or relying on marketing numbers, we will extend ERT to support reduced precision (FP16 and FP32) and Tensor Cores (HMMA instructions). Additionally, whereas the FMA mix in the instruction set was insufficient in determining the performance asymptote in GPP, we will extend our `nvprof`-based Roofline methodology to incorporate occupancy in order to determine if we are expressing sufficiently thread-level parallelism to hide the GPU's high latencies. Moreover, echoing our efforts to understand the impact of FPADD and FPMUL on FP64 performance, we will develop the requisite methodology to capture and visualize the performance bottlenecks arising from mixed precision or Tensor Core accelerated applications.

Although the traditional, operation-oriented (GFLOP/s) Roofline model can readily assess performance and memory bottlenecks, it is poorly suited for assessing either an application's exploitation of complex instruction set computing (CISC) or SIMD instruction set architectures (ISAs), or the degree to which an application utilizes a processor's functional units — something that can manifest in mixed precision or partially vectorized code. To that end, we will create an alternate Roofline methodology focused on floating-point instructions per cycle (IPC_{FP}) and floating-point instructions per byte. Recasting Roofline as such does not express performance (as it is agnostic of SIMD and FMA), but allows users to understand when functional units can be fully utilized executing a mix of reduced precision, Tensor Core, SIMD, or scalar instructions.

Finally, we will continue to apply our GPU Roofline methodology to more applications from a wider set of domains as well as extending our methodology to other accelerated architectures.

ACKNOWLEDGEMENTS

This material is based on 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.

REFERENCES

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, 2009.

[2] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin, R. Belenov, P. Thierry, Z. Zhao, R. Gayatri, H. Shan, L. Oliker, J. Deslippe, R. Green, and S. Williams, "A Novel Multi-level Integrated Roofline Model Approach for Performance Characterization," in *ISC*, 2018.

[3] S. Williams, "Auto-tuning Performance on Multicore Computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, December 2008.

[4] NERSC LIKWID Documentation. [Online]. Available: https://www.nersc.gov/users/software/performance-and-debugging-tools/likwid/

[5] NERSC SDE Documentation. [Online]. Available: https://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

[6] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. H. He, T. Kurth, T. Koskela, M. Lobet, T. M. Malas, L. Oliker, A. Ovsyannikov, A. Sarje, J. Vay, H. Vincenti, S. Williams, P. Carrier, N. Wichmann, M. Wagner, P. R. C. Kent, C. Kerr, and J. M. Dennis, "Evaluating and Optimizing the NERSC Workload on Knights Landing," in *PMBS Workshop at SC*, 2016, pp. 43–53.

[7] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. M. Malas, J.-L. Vay, and H. Vincenti, "Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor," in *IXPUG Workshop at ISC*, 2016, pp. 339–353.

[8] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. Deslippe, and S. Williams, "An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability," in *P3HPC Workshop at SC*, 2018.

[9] General Plasmon Pole (GPP) Kernel. [Online]. Available: https://github.com/cyanguwa/nersc-roofline

[10] HPGMG CUDA Code. [Online]. Available: https://bitbucket.org/nsakharnykh/hpgmg-cuda

[11] TensorFlow. [Online]. Available: https://tensorflow.org

[12] Empirical Roofline Toolkit (ERT). [Online]. Available: https://bitbucket.org/berkeleylab/cs-roofline-toolkit

[13] Python Matplotlib. [Online]. Available: https://matplotlib.org

[14] Example Scripts for Plotting Roofline. [Online]. Available: https://github.com/cyanguwa/nersc-roofline

[15] BerkeleyGW. [Online]. Available: https://berkeleygw.org

[16] J. Soininen, J. Rehr, and E. L. Shirley, "Electron Self-Energy Calculation using a General Multi-Pole Approximation," *Journal of Physics: Condensed Matter*, vol. 15, no. 17, p. 2573, 2003.

[17] HPGMG Website. [Online]. Available: https://hpgmg.org/

[18] HPGMG-FV Documentation. [Online]. Available: http://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg

[19] AMReX Documentation. [Online]. Available: https://amrex-codes.github.io/amrex/

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." [Online]. Available: https://www.tensorflow.org/

[21] `tf.nn.conv2d` Kernel. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/nn/conv2d

[22] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *arXiv e-prints*, p. arXiv:1802.09941, Feb 2018.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv e-prints*, p. arXiv:1512.03385, Dec 2015.

[24] PyCUDA Website. [Online]. Available: https://mathema.tician.de/software/pycuda