Lawrence Berkeley National Laboratory

LBL Publications

Title

Performance analysis of gpu programming models using the roofline scaling trajectories

Permalink

<https://escholarship.org/uc/item/5nv5d9b2>

ISBN

9783030495558

Authors

Ibrahim, KZ
Williams, S
Oliker, L

Publication Date

2020

DOI

10.1007/978-3-030-49556-5_1

Peer reviewed

Performance Analysis of GPU Programming Models Using the Roofline Scaling Trajectories

Khaled Z. Ibrahim, Samuel Williams, and Leonid Oliker

Lawrence Berkeley National Laboratory,
One Cyclotron Road, Berkeley, CA 94720, USA
kzibrahim, swwilliams, loliker@lbl.gov

Abstract. Performance analysis is a daunting job, especially for the rapid-evolving accelerator technologies. The Roofline Scaling Trajectories technique aims at diagnosing various performance bottlenecks for GPU programming models through the visually intuitive Roofline plots. In this work, we introduce the use of the Roofline Scaling Trajectories to capture major performance bottlenecks on NVIDIA Volta GPU architectures, such as warp efficiency, occupancy, and locality. Using this analysis technique, we explain the performance characteristics of the NAS Parallel Benchmarks (NPB) written with two programming models, CUDA and OpenACC. We present the influence of the programming model on the performance and scaling characteristics. We also leverage the insights of the Roofline Scaling Trajectory analysis to tune some of the NAS Parallel Benchmarks, achieving up to $2\times$ speedup.

Keywords: Roofline Model, Performance Analysis, Parallel Scaling, GPU, OpenACC, CUDA.

1 Introduction

Accelerator technologies are nowadays prevalent in HPC computing. The top two machines in the top500 list [17] of June 2019 are based on NVIDIA Volta GPUs. The trend of using accelerator stems from the difficulty to improve the performance of general-purpose cores based on CMOS technologies. The introduction of GPU to general-purpose computing, although almost being more than a decade old, is still a complex endeavor for many application developers. Part of the difficulty is due to the architectural model, which impact how data should be laid out for optimal performance, the memory consistency model which affects handling data dependencies, the control-flow with its impact on lock-step execution, and the tradeoffs between doing recomputation or loading precomputed data.

Performance analysis for accelerator architectures is a daunting process. Performance tools, such as NVIDIA nvprof or Intel Vtune, could provide access to numerous hardware events capturing various events that could correlate to the

observed performance. They may always not provide a notion of performance optimality, needed to assess whether further optimizations are needed.

For GPUs, metrics such as warp efficiency and occupancy are known to influence the observed performance, but there no agreed-upon method to assess the dependency of the observed performance on these metrics, or how impactful are these metrics on performance.

To this end, this paper introduces the use of Roofline Scaling Trajectories [10] to analyze the performance of GPU architectures. Specifically, we show how the Roofline Scaling Trajectory method stresses various architectural features as we change the number of GPU SMs involved in the computations. As such, these trajectories reveal the efficiency of GPU warp execution, the level of occupancy while changing the GPU SM count, and the efficiency in handling the temporal locality of the shared last level cache. The scaling trajectories shows the impact of the above inefficiencies against the GPU performance limits that is a function of the arithmetic intensity of the algorithm. We perform the analysis for several NAS Parallel Benchmarks [2] ported to the GPU architecture using the CUDA programming model and the pragma-based OpenACC model.

We proposed the use of the Roofline Scaling Trajectories to reveal the root cause of various performance inefficiencies. We show how to visually associate the scaling trajectories to various sorts of warp inefficiencies, including those due to branch divergence and those related to latency divergence, and occupancy degradation due to the lack of thread block parallelism. Moreover, the Roofline Scaling Trajectories has been shown [10] to reveal cache thrashing effects and its impact on temporal locality as we scale applications. Identifying the performance bottleneck of applications, we leveraged these insights in tuning the performance of two of the NAS benchmarks, achieving up to a $2\times$ improvement.

The rest of this paper is organized as follows. § 2 presents the motivation of this study. We summarize the performance influencing factors for GPU programming in § 3. The programming model used for offloading computation to GPU accelerator and experimental setup are presented in §4 and § 6, respectively. We introduce our novel performance analysis technique based on scaling trajectories in § 5. We show the effectiveness of the proposed analysis technique in studying various NPB in § 7, and our tuning efforts based on the introduced analysis in § 8. We finally present related work in § 9 and conclude in § 10.

2 Motivation

The performance of two GPU ports of NPB 3.3, detailed in Section 6.1, is shown in Figure 1. The first is based on OpenACC directives [15], and the other is based on the CUDA programming model [6]. We run the problem sizes, or classes, that could fit in the GPU memory for each implementation. We present strong scaling behavior while changing the number of SMs involved in the computation for multiple problem sizes. To conduct these experiments, we leveraged a new feature of the Volta architecture to enable Multi-Process Service (MPS) [14]. The hardware support for MPS allows dedicating a subset of the compute resources

to a particular process or an application, thus achieving performance isolation (or QoS) and address space isolation.

We could easily recognize various suboptimal performance trends. For the OpenACC port of the NPB [19], we observe that LU and FT do not significantly improve with the increase in SM count, while MG improves and then drop in performance at high SM count. The performance does not improve significantly with the increase of problem size from Class A to B or C. For the CUDA variant of NPB [9], LU has a significant improvement with SM count. Increasing the problem size from Class A to B further improves the performance, but the trend is reversed when changing from Class B to C.

Unfortunately, the strong scaling curves show that there is room for improvement, but they do not shed light on the causes of the observed behavior or where the tuning effort of these GPU applications should be steered. Several profiling

Fig. 1: Strong scaling of various NPB applications with the number of GPU SM using two programming models, CUDA and OpenACC. We observe various suboptimal scaling trends without a clear cause. The largest Class that could fit on a single GPU depends on the programming model. tools tries to address this challenge by showing hotspot analysis, hardware metrics for various architectural activities, communication, or access pattern. These

tools include nvprof [13], TAU [16], etc. This paper extends the Roofline Scaling Trajectory technique [10] to demystify some of the observed performance trends.

3 Performance on GPU Architecture

The performance on a GPU is typically correlated to the efficiency of leveraging various architectural features. The NVIDIA GPU architectures have multiple levels of parallelism [14]: threads within a warp, warps sharing an SM, and multiple SMs within a GPU. Threads within a warp use the SIMT execution model. The execution of a warp of threads is typically more efficient when all threads choose the same execution path and create coalesced memory access. Earlier generations of GPUs use a single program counter for a warp of threads, as such the impact of a branch divergence is severe on performance. Volta GPUs provide a program counter and stack per thread, which alleviates the need for frequent re-convergence, but the execution remains more efficient when branch divergence is minimized. A CUDA thread block is a software abstraction that uses multiple warps and provides low-overhead synchronization primitives and communication through the shared memory. All warps of a thread block are scheduled and executed on the same SM. The hardware scheduler could schedule multiple thread warps (and possibly blocks) within an SM, assuming sufficient hardware resources are available, including registers, shared memory, etc. All warps co-scheduled within an SM use the same set of function units, 64 FP32 for Volta. The number of SMs in Volta is 80. Volta can schedule up to 64 warps per SM. Volta provides the multiprocess service (MPS) facility to allow multiple kernels to run concurrently on the GPU and to control the number of SM assigned to a kernel. We leveraged the MPS support in this study to control the number of SMs used concurrently.

The efficiency of performing computation on NVIDIA GPU requires careful consideration for the following dimensions.

- Warp efficiency within a group of threads, or SIMT efficiency, i.e., all threads follow the same path of execution or have the same latency to execute an instruction.
- SM Occupancy, i.e., the ability to schedule as many warps per SM as possible to hide the long latency of accessing the memory system. This also involves keeping GPU busy most of the time.
- Data locality, i.e., effective memory request coalescing and temporal L2 cache access.

These objectives could be conflicting. For instance, GPU occupancy would require pipelining small kernels, but one must provide enough parallelism to saturate the SMs. Efficient use of the cache hierarchy may also conflict with providing enough independent thread warps.

Ideally, a performance tool or technique would identify the performance sub-optimality and link it to one of the above performance dimensions.

4 GPU Programming Models in This Study

The benchmarks examined in this paper are written in two popular approaches for programming accelerators: directive-based offloading, and vendor-specific programming model (specifically CUDA for NVIDIA GPUs). The directive-based approach is a more productive approach for porting code to GPUs, but it may not leverage all vendor-specific architectural features. It allows for an incremental approach for porting codes to GPUs. The use of the CUDA model provides full control of vendor-specific hardware features, such as shared memory across threads, texture cache, etc. As such, it could provide a performance advantage while sacrificing some portability and productivity.

The OpenACC code parallelization of the NPB relies on maintaining the loop structure of the code, where the outer level of the loop nest is assigned to gangs, the second level to workers, and the third to vectors. For loop nest levels beyond three, some of the inner levels are unrolled manually. The objective of this programming style for accelerators is to strike a balance between the achieved performance and the coding effort to port the code. The OpenACC directive-based approach generally preserves the code structure and data layout.

The CUDA version uses an explicit mapping of each level of the loop into a dimension of the thread block or grid of threads. Because some of the inner loops have small trip counts, the developer used one of the thread dimensions to serve multiple levels of the loop nest. The developers also leveraged the GPU shared memory for efficient communication between threads sharing an SM.

5 GPU Roofline Scaling Trajectories

In this paper, we leverage the Roofline Scaling Trajectory technique for analyzing the performance and scalability of GPU-accelerated parallel applications. We aim to leverage this analysis technique to identify various kinds of performance bottlenecks that an application may experience on a GPU architecture. The Roofline Scaling Trajectory visualizes the scaling behavior and identifies the effects of cache and memory access locality, warp efficiency, and SM and GPU occupancy on application performance.

Figure 2 shows an example of Roofline Scaling Trajectories curves. Nominal machine-specific Rooflines are constructed for both the lowest and the highest level of concurrency (2 SMs and 80 SMs on Volta). The trajectory is a trend line of application performance and arithmetic intensity at each level of concurrency. One may apply this analysis for the full application or individual kernels. We use the empirical measurement technique laid out in earlier GPU Roofline studies [4, 12] to characterize the machine characteristics. Our focus is mainly on the DRAM Roofline model, where one must measure `dram_read_transactions` and `dram_write_transactions` metrics through the NVIDIA `nvprof` profiling tool. We use the FLOPs reported by the application for all concurrency level to ensure that throughput reflects the application performance. We calculate the applica-

tion arithmetic intensity as follows:

$$AI \leftarrow \frac{\text{canonical flop count}}{(\text{dram_read_trans} + \text{dram_write_trans}) \times 32} \quad (1)$$

The model could be easily extended to other levels of the hierarchy as presented in [4]. We rely on the application canonical FLOP count, estimated by the application developer rather than on relying on the profiling tool measurements. This allows consistent performance comparison because the number of FLOPs could change with the run configuration due to the use of data replication and reduction operations within a thread block, especially in the CUDA version. This method allows for a fair comparison across programming models.

Fig. 2: The Roofline Scaling Trajectory on GPU architectures. Each point represents the throughput at a certain SM concurrency level. We use the Roofline Scaling Trajectories to diagnose various performance scaling bottlenecks, including warp execution inefficiency, loss of occupancy while scaling, excessive data movement to the cache hierarchy, etc.

Ideally, the performance should improve linearly with the increase of computational resources without degrading the arithmetic intensity. On a Roofline plot, this translates into a vertical change of throughput proportional to the increase of computational resources while changing concurrency. In practice, an application may experience a suboptimal change in throughput, or a change of the arithmetic intensity while scaling, e.g., a scaling curve pending to the left.

The dominant bottleneck typically changes when strong scaling the application. For instance, at low concurrency, it is typically difficult to saturate the bandwidth to shared levels of the memory hierarchy. In such case, the warp efficiency becomes the main limiting factor for an application to reach the Roofline. Although lower occupancy could result in a similar effect at low concurrency, for a kernel with non-trivial size, this is unlikely. Additionally, the lack of occupancy is easily distinguishable, as will be discussed later. As we strong-scale the application run, observing a loss of arithmetic intensity implies some cache

thrashing at the shared cache levels. The potential of throughput improvement is typically impacted by the loss of arithmetic intensity for memory-bound applications. An occupancy reduction while scaling further lowers the observed performance gains.

Identifying the performance bottleneck is typically the first step to steer the optimization effort to the right problem. For instance, a warp efficiency issue occurs due to divergent branches or inefficient data indexing that results in a non-coalesced memory access across a thread warp. It would require particular code refactoring techniques, while loss of occupancy would require different remedies. The occupancy issue involves dealing with two conflicting constraints. First, improving parallelisms within a kernel requires a coarse-grained kernel to improve the GPU occupancy. Second, reducing stalls to launch a kernel requires pipelining kernel invocations by assigning a smaller task for each kernel. Balancing the two conflicting requirements requires some tuning for each target architecture. The code should ideally be structured to handle different task granularities.

6 Experimental Setup

6.1 Benchmark Suite

The NAS Parallel Benchmarks (NPB) [2] represent a broad set of computational patterns. The suite uses FT for spectral methods, CG for sparse linear algebra, LU for solving a regular-sparse lower and upper triangular system, and MG for multigrid PDE solver using a hierarchy of meshes. In addition, the suite contains two mini-apps, SP and BT, which carry key computational fluid dynamics (CFD) calculations on a structured grid. They involve the solution of independent systems of block tridiagonal equations with a 5×5 block size. The inner block dimensions are not friendly to warp sizes and different approaches are taken by the CUDA [9] and OpenACC [19] implementations. The OpenACC implementation assigns the full block to a single thread, while the CUDA version split these blocks between threads within a block. The first improves the efficiency of warp execution, while the second enhances the level of parallelism. Unfortunately, the CUDA authors only ported the SP, BT, and LU benchmarks.

6.2 System Setup

We conducted our experiments on the OLCF Summit supercomputer. Each summit node has two clusters of an IBM Power9 CPU and three NVIDIA Volta GPUs. Each cluster is connected through high-speed NVLink. Each node has a half-terabyte of coherent memory, and nodes are connected using dual-rail Mellanox EDR InfiniBand interconnect. Our experiments focused on the performance of a single Volta GPU. Each Volta GPU has 80 SMs each with a 256KB register file and 96KB of unified cache/shared memory. The SMs shared 6MB of L2 cache and 16 GB of HBM2 memory, and each GPU has a theoretical peak of 7.8 TFLOP/s in double precision.

7 Scaling Trajectory Analysis for Computational Kernels

Before presenting the Roofline Scaling Trajectory, we show the warp efficiency of various kernels for the studied set of benchmarks. The warp efficiency is defined as the ratio of the average active threads per warp to the maximum number of threads per warp [13]. This metric captures thread execution divergence and typically does not change with the workload distribution across SMs. Latency divergence for application with irregular memory access [5] has the same impact on performance.

Fig 3: Warp efficiency for various kernels of the NPB benchmarks. Generally, we observe higher warp efficiency for the OpenACC version compared to the CUDA version, specifically for the BT and LU benchmarks. Warp efficiency remains constant while changing the SM count and could change with the problem size.

Figure 3 shows the warp efficiency for various kernels of each NPB application. Generally, the OpenACC port has a higher warp efficiency compared with the CUDA port, especially for the BT and LU. Changing the dataset size (i.e., the Class) could result in improvement of the warp efficiency, such as in the case of CUDA LU, or reduction in efficiency as in the case of OpenACC BT, or may have irregular change as with the CUDA SP. We will show that warp efficiency, as well as the indexing scheme of the arrays, influences the performance at low SM concurrency.

Figure 4 shows the Roofline line scaling behavior of the NPB applications. First, we observe that all applications are memory-bound based on their arithmetic intensity. As such, we simplified the Roofline architectural limits by removing the fused multiply-add ceiling for the compute-bound region as it is unattainable. The CUDA LU and OpenACC MG show the most noticeable change in AI, during strong or weak scaling. For BT, we notice a better efficiency at low concurrency for the OpenACC implementation compared with the CUDA variant due to the better warp efficiency, but the CUDA variant has a better occupancy when scaling. Except for CG and LU, OpenACC implementations have a high starting efficiency, due to the better warp efficiency. The OpenACC CG warps have a low efficiency at low concurrency that is due to the latency divergence [5], which reduces the warp efficiency. Importantly, this sort of warp inefficiency is typically not captured by the nvprof metric, but is captured by the Roofline Scaling Trajectory. The CUDA BT low warp efficiency manifests at low SM count in the Roofline scaling plot. For applications where both OpenACC and CUDA implementation exist, the arithmetic intensity is higher for the CUDA implementation, i.e., less data movement to the L2 cache is involved, which is critical for achieving a higher performance for memory-bound applications.

For CUDA LU and SP, we observe the loss of arithmetic intensity as we change the problem class (size). For LU, increasing the problem size improves the GPU occupancy, as such change from Class A to B results in performance improvement. For Class C, the reduction in arithmetic intensity, due to excessive data movement, reduced the overall gains. We observe a similar but less profound trend for SP. For OpenACC MG, we observe a performance exceeding the memory bound at low concurrency due to the efficient reuse of the L2 data. This results in arithmetic intensity improvement with weak-scaling at low concurrency. As one strong scales, the arithmetic intensity is reduced, especially at high concurrency.

For the OpenACC LU, we notice both low efficiency at small scale and suboptimal occupancy improvement as one strong scales. There is a slight improvement with the change of SM count. This application has a GPU occupancy problem that manifests at low SM count, as will be discussed later. For the CUDA LU, we notice the improvement at low concurrency while increasing the problem size, which correlates with the warp-efficiency improvement.

The change in occupancy affects the strong scaling behavior of applications. An application needs enough parallelism to saturate all available SMs to achieve

Fig. 4: Roofline Scaling Trajectories for Volta GPUs. CUDA BT and OpenACC LU application are limited by warp efficiency, while CUDA LU, OpenACC SP, BT, FT are limited by GPU occupancy. The arithmetic intensity of CUDA LU and OpenACC MG is noticeably affected by strong and weak scaling.

optimal performance. As discussed in Section 5, the occupancy term affects the potential improvements while increasing the SM count. In Figure 5, we consider the detailed SM occupancy behavior for CUDA BT and CUDA LU. For BT, we observe a constant occupancy during both weak and strong scaling. The scaling trajectory shows a high correlation with the measured SM occupancy behavior.

The occupancy of CUDA LU degrades with strong scaling. Increasing the problem size improves the occupancy, and the impact is not uniform across kernels. The `jacld.blts` kernel is the most affected by the loss of occupancy. The Roofline Scaling Trajectories for individual kernels captures such behavior precisely as shown in Figure 6, both for loss of occupancy for the `jacld.blts` kernel, and the maintenance of good occupancy for the `rhs_kernel_*`. We notice that the `rhs_kernel_x` has a small change in arithmetic intensity during weak scaling compared with the `rhs_kernel_y, z`. Inspecting the code, we found a unit strided access in the x -direction and strided jumps for the y and z direction. Applying data transposes could typically be used to tackle such bottleneck, but require an efficient transpose that is lower overhead than embedding the strided access.

In general, there are two sources of loss of occupancy. One is reported by the `nvprof` profiling tool as achieved occupancy during the course of executing a kernel, and is defined as the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor [13]. We refer to

Fig. 5: Occupancy of CUDA Kernels with various Classes. LU suffers occupancy loss as we change the SM count, while BT has a stable occupancy for all concurrency levels.

Fig. 6: CUDA LU scaling trajectories for individual kernels. The potential for performance scaling is significantly influenced by the change in arithmetic intensity for rhs_kernel_y, z, caused by strided access, compared with rhs_kernel_x.

this type as SM occupancy. The second is due to the idle time between launching kernels due to kernel invocation overhead, waiting for data movement between the host and the GPU, etc. We refer to this type as GPU occupancy. Ideally, we need to minimize the second type of occupancy loss because an idle GPU results in no use for all the SMs. Sources of this loss of occupancy include launch overhead and CPU pre-processing to launch a kernel. To quantify the CPU overhead to invoke a kernel on the GPU, we multiply the number of invocations by the uncontended latency for an invocation. We estimate the uncontended latency using the minimum observed at run time. We account not only for kernel invocation but also for other CUDA routines, such as cudaMemcpy. Our aim is to avoid considering the contention due to busy activity on the GPU as part of the GPU idle time.

Depending on the programming model, the number of device invocations could be significant. In general, we observed higher kernel invocation count in the OpenACC port compared with the CUDA port due to the incremental approach for porting each kernel. To preserve the loop structure for the CPU and device offload case when porting a routine with multiple loops using OpenACC, developers must individually annotate each of the loops, and the compiler must generate a kernel for each loop. The application most impacted by the invocation count is OpenACC LU. The OpenACC implementation does not leverage the shared memory and as such can achieve high occupancy at low concurrency.

Fig. 7: Left two figures: execution time decomposition for both OpenACC and CUDA for the LU benchmark - Class B. Right two Figures: Kernel invocation count for different Classes. The OpenACC invocation is roughly $5\times$ the CUDA version, leading to smaller granularity for kernels, which affects the occupancy and exacerbates the impact of CUDA runtime overheads.

Figure 7 depicts the invocation count for both the CUDA and the OpenACC port. Both have high invocation count with the OpenACC port is about $6\times$ the CUDA invocation count. As such, the percentage of execution time spent

in these overheads could be a significant time of the total execution time, as shown in Figure 7. The time for overhead for the OpenACC version is more than $5\times$ the overhead for the CUDA version. To distinguish the two kinds of loss of occupancy, we augment the scaling trajectory with an Amdahl shadow curve, which shows the maximum attainable throughput if all CUDA overheads are removed. The remaining loss of occupancy is due to limited concurrency during the kernel scaling. As shown in Figure 9 for LU application, when the shadow curve hits the roofline, like the CUDA case, the application is suffering solely from GPU occupancy issue (no GPU activity due to overheads). When the shadow curve does not hit the roofline, like the OpenACC case, the application is suffering from active SM occupancy issue in addition to the inactive GPU issue.

Fig. 8: Tuning tile size for BT and its impact on occupancy and warp efficiency. The best performing variant at the bottom sees improved warp efficiency at the cost of reduced occupancy, but the occupancy does not degrade with SM count.

7.1 GPU Cache Impact on Scaling Trajectories

We note that the overall capacity of the L1/shared memory¹ exceed the L2 capacity starting with a concurrency level of 48 SMs. Similarly, the register file capacity² exceed the L2 capacity starting with a concurrency level of 24 SMs. As such, leveraging the L2 temporal locality becomes hard to achieve beyond a certain concurrency level, unless the compiler or the programmer annotates memory accesses with appropriate cache hints to distinguish streaming memory accesses from accesses likely to leverage temporal locality. Moreover, while L2 could be effective in filtering traffic to the memory at low concurrency, the L1 surpass the the L2 in filtering effect at high concurrency leading to the improved arithmetic intensity applications with high SM concurrency. This arithmetic intensity trajectory reversal appears in multiple kernels including the Class C LU `rhs_kernel.y` routine, which we initially thought to be a performance anomaly.

Fig. 9: Scaling trajectories with occupancy limits due to Amdahl’s law serialization factor for LU using OpenACC and CUDA. Hitting the roofline with Amdahl shadow curve indicates inactive GPU occupancy issue (CUDA LU-Class B), otherwise active SM occupancy is an additional factor in the observed performance degradation.

8 Performance Tuning

While the main focus of this study is to develop a performance diagnostic technique, we explored the performance tuning for a couple of cases of the studied kernels, the CUDA BT and OpenACC FT. For the CUDA BT, the scaling trajectories reveal inefficiency at low concurrency, a symptom of a warp inefficiency issue. Inspecting the code, we found that a tuning parameter controls the number of 5×5 blocks that are assigned to a thread block and could be loaded into the shared memory. Increasing the block count leads to a reduced occupancy, as

¹ Volta configurable L1 cache/shared memory capacity is 128KB per SM.

² Volta register file is 256KB per SM.

fewer threads blocks could share an SM, but it increases the efficiency. We explored different value for this parameter, and we found the best blocking value at $4\times$ the default value. The performance improvement ranged from 32-41% at full concurrency for the three problem sizes. In Figure 8, we present the change in occupancy and warp efficiency before and after changing the blocking factor. Improving warp efficiency is key to performance. The reduction in occupancy does not hurt the performance as long there is no drop in occupancy as we change the SM count. This reduction in occupancy in this particular case is due to the decrease in thread block count and warps sharing an SM. The improvement in efficiency is $2.47\times$, while the drop in occupancy is $1.33\times$.

The second example is the OpenACC FT implementation, which is one of the applications that did not show efficient scaling due to poor GPU occupancy. Analyzing the kernel-wise scaling trajectory, we found that the bottleneck is part of the initialization routine, which is included in the benchmark execution time. On earlier generations of GPUs, the overhead of this phase is small compared with the rest of the execution time. On Volta, parallelization of the initialization part provided a $2\times$ speedup for the whole benchmark. Overall, identifying the source of the performance degradation is half the road to tuning the application. The main limitation of the proposed technique is that it is mainly applicable at the kernel-level granularity.

9 Related Work

Performance analysis techniques can be categorized into two complementary classes: microscopic analysis, which relies on hardware events associated with a kernel invocation or a line of the source code, and macroscopic analysis, which focuses on system utilization or use efficiency.

Attributing hardware events—such as cycle count, cache misses or vectorization efficiency—to the source code steers the optimization efforts to hotspots within the program. Numerous tools, such as Intel Vtune [11] the CrayPAT [7], HPCToolkit [1] Tau [16] Scalsca [3], provide effective communication of underlying events and typically rely on sampling techniques to reduce overhead. These tools may not assess the potential for performance improvement.

The roofline technique can be classified as a macroscopic analysis technique that compares the application performance relative to realistic architectural limits that vary with the application arithmetic intensity. It identifies when the performance hits the machine limits. Various methods are typically used to characterize the application arithmetic intensity, including DRAM-based roofline technique [18] and the cache-aware roofline Model (CARM) [8], which considers the data movement to the L1 cache. The roofline method has been extended to different cache levels, incorporated into production quality tools such as Vtune [8], and is recently extended to the GPU architectures [4], but the main focus of these efforts is for the performance at the full node concurrency.

The roofline scaling trajectory [10] has recently been introduced for CPU-based architectures, where it is used to study the interaction with the cache

hierarchy. In this work, we extend its use to the performance analysis for GPU architectures and show its effectiveness in exposing various performance bottlenecks such as warp efficiency and SM occupancy. We also show the influence of the programming model on various performance bottlenecks.

10 Conclusions

In this paper, we introduce the use of the Roofline Scaling Trajectory technique for analyzing GPU-accelerated workloads. The technique leverages a new NVIDIA Volta capability that allows controlling the number of SMs. The introduced method intuitively visualizes various performance bottlenecks such as warp inefficiency, suboptimal SM occupancy, in addition to characterizing the efficiency of utilizing the cache hierarchy for capturing locality. We used this analysis technique to study two implementations of NAS benchmarks using the CUDA and the OpenACC programming models. Our analysis reveals the performance bottlenecks of the studied applications and the influence of the programming model on warp efficiency, SM occupancy, and memory access. We leveraged these insights to tune two of the studied applications, achieving up to a $2\times$ improvement in run time.

Acknowledgment

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

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: Tools for performance analysis of optimized parallel programs <http://hpctoolkit.org>. *Concurr. Comput. : Pract. Exper.* **22**(6), 685–701 (Apr 2010)
2. Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-010, NASA Ames Research Center (1995)
3. Calotoiu, A., Hoefer, T., Poke, M., Wolf, F.: Using automated performance modeling to find scalability bugs in complex codes. In: SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2013)
4. Charlene Yang and Thorsten Kurth and Samuel Williams: Hierarchical Roofline Analysis for GPUs: Accelerating Performance Optimization for the NERSC-9 Perlmutter System. In: Cray User Group (CUG) (May 2019)

5. Chatterjee, N., O'Connor, M., Loh, G.H., Jayasena, N., Balasubramonia, R.: Managing dram latency divergence in irregular gpgpu applications. In: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 128–139 (2014)
6. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2013)
7. Cray: The Cray Performance Measurement and Analysis Tools. <https://pubs.cray.com/content/S-2376/6.4.0/cray-performance-measurement-and-analysis-tools-user-guide-640/craypat>
8. Ilic, A., Pratas, F., Sousa, L.: Cache-aware roofline model: Upgrading the loft. *IEEE Comput. Archit. Lett.* **13**(1), 21–24 (Jan 2014)
9. Jörg Dümmler: A CUDA version of NPB 3.3.1. <https://www.tu-chemnitz.de/informatik/PI/sonstiges/downloads/npb-gpu/index.php.en>
10. Khaled Z. Ibrahim, Samuel Williams, Leonid Oliker: Roofline Scaling Trajectories: A Method for Parallel Application and Architectural Performance Analysis. In: International Conference on High Performance Computing & Simulation (HPCS) (Jun 2018)
11. Marowka, A.: On performance analysis of a multithreaded application parallelized by different programming models using intel vtune. In: Proceedings of the 11th International Conference on Parallel Computing Technologies. pp. 317–331. PaCT'11 (2011)
12. Measuring Roofline Quantities on NVIDIA GPUs: Portability Across DOE Office of Science HPC Facilities. <https://performanceportability.org/perfport/measurements/gpu/>
13. nVidia: CUDA Profiler Users Guide. https://docs.nvidia.com/cuda/pdf/-CUDA_Profiler_Users_Guide.pdf
14. nVidia: NVIDIA TESLA V100 GPU ARCHITECTURE. <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
15. OpenACC STANDARD Organization: OpenACC Application Programming Interface. <https://www.openacc.org>
16. Shende, S.S., Malony, A.D.: The tau parallel performance system. *Int. J. High Perform. Comput. Appl.* **20**(2), 287–311 (May 2006)
17. Top 500 Supercomputers: <http://www.top500.org>
18. Williams, S., Watterman, A., Patterson, D.: Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM* (April 2009)
19. Xu, R., Tian, X., Chandrasekaran, S., Yan, Y., Chapman, B.: Nas parallel benchmarks for gpgpus using a directive-based programming model. In: Brodman, J., Tu, P. (eds.) *Languages and Compilers for Parallel Computing*. pp. 67–81. Springer International Publishing, Cham (2015)