DELTA: Validate GPU Memory Profiling with Microbenchmarks

Xianwei Zhang, Evgeny Shcherbakov
{xianwei.zhang,evgeny.shcherbakov}@amd.com
Advanced Micro Devices, Inc.

ABSTRACT

With the advent of GPU computing, profiling tools are now widely used to assist developers in identifying and solving performance bottlenecks. Those tools are commonly relying on hardware performance counters to grant users the access to low-level activities. Considering the increasing complexity of modern GPUs and also the efforts needed to associate program behaviors with hardware events, it is not trivial to construct the profiling tools with assured correctness. As a result, profiling tools should be strictly validated to make sure that the program behaviors and resource usages can be correctly captured and analyzed. To aid the validation, we create a testing prototype *DELTA*, on top of the open-source Radeon Open Compute platform (ROCm), to investigate the values of the derived profiling metrics and their underlying basic counters. The tests of *DELTA* are generally based on the classical microbenchmarks, which are capable to control program behaviors to generate predictable statistics. Differing from prior dissecting works, our tests are to examine the profiled results and to compare against desired patterns, reporting whether the profiling tools are correctly working with appropriate data collection and processing. This paper presents the validation methodology and experimental results of cache and main memory on the recent GPUs and ROCm platform, and the case studies demonstrate that the tests are helpful to scrutinize the profiling tools.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; • **Software and its engineering → Software verification and validation**.

KEYWORDS

GPU, Memory, Microbenchmark, Profiling, Validation

ACM Reference Format:
Xianwei Zhang, Evgeny Shcherbakov. 2020. DELTA: Validate GPU Memory Profiling with Microbenchmarks. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3422575.3422784

MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA

ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422784

1 INTRODUCTION

In the past decade, graphics processing units (GPUs) have been rapidly evolving from graphics processing to general purpose computing. Offering massive parallelism and high energy efficiency, GPUs are now widely deployed to accelerate emerging high performance computing and machine learning applications. Particularly, plenty of existing and planned supercomputers are adopting GPU-centric nodes to meet the computation demands, such as the currently fastest Summit [29] and the scheduled exaFLOPS Frontier [28].

With excessive computation resources on GPUs, developing efficient applications is extremely critical and requiring a lot of expertise and efforts. Without careful designs, the GPU execution units and memory system are likely to be idling or ineffectively exploited. As such, mainstream computing platforms like CUDA and OpenCL are actively improving to ease the programming burden, and extensive models like RAJA [14] and OpenMP target [15] are being developed to further abstract out details. While with much programming convenience, those high-level models make it even more difficult for developers to pinpoint performance bottlenecks by associating the key low-level details (e.g., instruction counts and cache misses) with observed program behaviors (e.g., execution time and memory footprint). To address the problem, profiling tools (e.g., nvprof [26] and rocprof [11]), are thus provided to enable developers to understand fundamental properties of the applications and further identify optimization opportunities.

The profiling generally relies on underlying hardware counters, which are collected during program execution, then post processed and presented as high-level metrics and analysis to developers. To precisely depict the program behaviors, the profiling tools are required to be correct on all levels, including the counter setting, value collection and further metric calculation. However, it can be problematic in any phase, and thus the profiling may eventually give misleading results to the users. Therefore, it is significantly critical to validate the profiling tools on result correctness with dedicated testing frameworks.

Aiming at the validation, this paper proposes a set of tests based on classical microbenchmarks and then develops some well-understood metrics to calibrate the profiling results at end user level. With the tests, the hidden profiling issues can be easily captured and examined. The focus of this paper is placed onto the cache and memory system, which are very precious resources highly competitive among massive threads and are thus crucial to the overall performance of GPUs [1, 17, 27]. The contributions of this paper are:

- we highlight the importance of validating GPU profiling tools and present a testing methodology using microbenchmarks. To the best of our knowledge, this is the first work focusing on GPU profiler validation.
- our designed testing prototype, which we call *DELTA*, well covers GPU cache and memory behaviors from different perspectives, including misses, data size and access latency.
- as case studies, the validation tests on recent GPUs effectively identified the hardware configuration details, including the existence of optimized cache policy and memory changes across GPU generations.

The rest of the paper is organized as follows. Section 2 introduces GPU models, profilers and microbenchmarks. Section 3 elaborates the motivation and testing designs. Section 4 presents the experimental methodology and analyzes our experimental results. Section 5 discusses related work. Section 6 concludes the paper.

2 BACKGROUND

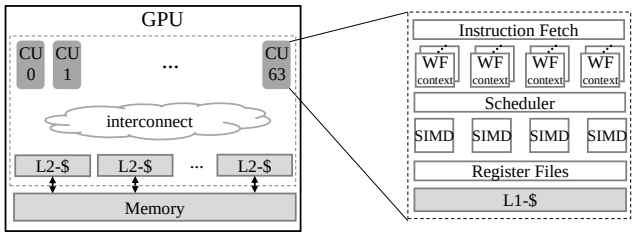


Figure 1: GPU high-level organization.

2.1 GPU Organization and Programming

Figure 1 shows the high-level architecture of a general-purpose GPU, based on recent AMD GCN design [4]. A GPU is composed of multiple compute units (CUs), also known as streaming multiprocessors (SMs). As the basic unit of computation, each CU consists of a collection of SIMD units, each of which typically contains 16 stream processors (SP cores, not shown in the figure), to execute instructions of different types like arithmetic and load/store. In terms of the memory hierarchy, a CU contains a private vector data cache (L1) and shares a scalar cache and instruction cache with several other CUs. All the three caches are then supported by a shared L2 cache, which in turn connects to main memory, which can be GDDR or HBM. Note that L2 and memory are typically partitioned into multiple slices or channels to achieve higher bandwidth and lower access latency.

To use GPUs, programmers define the parallel portions of applications as kernels and offload them onto GPUs with the help of device driver and runtime. GPU hardware schedulers then dispatch the kernels onto CUs in work-group (WG) granularity. Each WG is a set of threads, which are grouped into bundles of 32 or 64, known as wavefronts (WFs) or warps, to be assigned to a SIMD unit. Threads of a WF execute the kernel code in lockstep by going through multiple pipeline stages (e.g., instruction fetch, decoding and scheduling) and generate requests to access the memory hierarchy.

2.2 GPU Profiling and Dissecting

To characterize GPU programs, multiple software profiling tools were developed, such as NVBit [31] and VTune Amplifier [21]. Whereas with flexibility, software-based profiling may involve prohibitive overheads and even distort the program dynamic behaviors due to the inserted instrumentation codes [21]. As a result, hardware profiling tools, such as nvprof [26] and rocprof [11], are popularly used to characterize applications and tune performance. Compared to software, those hardware-based tools are of much lower overheads and are more transparent to users. As a good balance on flexibility and overhead, the open source rocprof, a part of Radeon Open Compute platform (ROCm) [10], enables third-party users to easily extend and customize the tool and runtime for particular scenarios.

For hardware-based profiling, performance counters are provided by GPU manufacturers to tally specified actions (e.g., instruction stalls, cache requests and elapsed cycles) happening in the components. Those counters enable users to measure and understand the low-level hardware activities. To quantify the hardware events, metrics (i.e., statistics based on the counters) are further developed to analyze the behaviors. To bridge the raw counters to high-level metrics, profiling tools are thus provided to closely monitor kernel executions and then wrap up the counter processing. The rocprof is one such tool to characterize compute applications on a broad set of GPU architectures. Particularly, rocprof allows to easily add new counters and metrics for customized usages, and thus it serves as our experimental platform in this paper.

Whereas expertise and details of GPU components are necessitated for GPU performance profiling and tuning, some architectural parameters are typically not publicly released by GPU vendors, thus motivating ample researchers to resort to demystify using microbenchmarks. Wong *et al.* [32] investigated the hardware of Nvidia GT200 GPU by creating a suite of microbenchmarks, which were later used to dissect the cache structure of Fermi GPU [13]. To cover previously unknown characteristics, Mei and Chu [25] proposed a novel fine-grained pointer chasing (P-chase) benchmark to examine the GPU memory hierarchy. Jia *et al.* [22] continued the line of research by benchmarking the Nvidia Volta architecture. While being utilized to reverse engineering GPU hardware parameters in prior works, the microbenchmarks in this paper are instead used to validate the performance counters and metrics.

2.3 Microbenchmark: Pointer Chasing

Table 1: Parameters of cache and P-chase test

Sym.	Description	Sym.	Description
C	cache capacity (unit: *int*)	N	array size (unit: *int*)
b	cache line size (unit: *int*)	s	stride size (unit: *int*)
a	cache associativity	M	#accesses to the array
T	number of cache sets	r	cache miss ratio

In this paper, P-chase will be primarily used for our studies on cache and memory system. P-chase is to perform a sequence of dependent accesses in varying strides onto different sized arrays and it is capable to reflect cache configurations. For ease of explanation, we will use the notations listed in Table 1. The P-chase is traversing

an array with size N and stride s in M total number of accesses. For illustration purpose, we will use 4-byte integer as the unit [1] for both the array and cache (i.e., stored elements are 4-byte integers). The cache is capable to hold C integers, a line size is of b integers and the associativity is a, giving the number of cache sets $T = C/ab$. Suppose that the cache is regularly organized (i.e., with basic replacement algorithm LRU and lowest address bits on set mapping) [2], then the stride allows to change the miss ratio by controlling the number of consecutive accesses to the same cache line, cache set, etc. Specifically, cache miss patterns are a function of array size (N) and accessing stride (s), as shown in Figure 2:

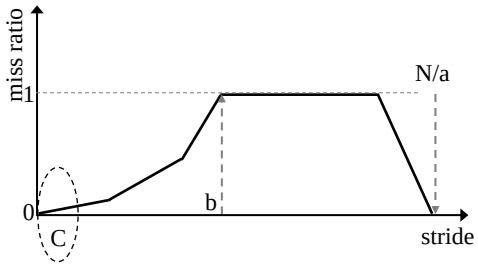


Figure 2: With varying traverse strides onto different sized arrays, P-chase microbenchmark exposes changing and predictable miss ratios, which reflect the cache configuration details.

- $r = (\frac{N}{x})/M,\ x = max(s, b)\ \ if\ N \leq C$
  The array fits in cache, and thus misses are just from cold start, which is $\frac{N}{x}$. Apparently, if there are sufficient accesses (i.e., $M >> \frac{N}{x}$), then miss ratio is zero.
- $r = \frac{s}{b},\ \ if\ N \geq 2C\ and\ 1 \leq s < b$
  If array is larger than the cache ($N \geq 2C$) and stride is less than line size ($1 \leq s < b$), then there are b/s consecutive accesses to the same cache line. The first access to a line is always a miss, as every line is displaced from the cache before it can be re-used in subsequent iterations. Therefore, the reuse only happens within a line (i.e., in the consecutive accesses).
- $r = 100\%,\ \ if\ N \geq 2C\ and\ b \leq s < \frac{N}{a}$
  If accessing stride goes beyond the line size ($s \geq b$) but is still smaller than the set size ($b < \frac{N}{a}$), then the reuse within a line goes away and meanwhile each line still gets replaced before reuse, thus forcing all accesses to be misses.
- $r = 0\%,\ \ if\ N \geq 2C\ and\ s \geq \frac{N}{a}$
  In this scenario, the number of addresses mapped to a single set is no larger than the set associativity, thus no more misses once the array is loaded. Again, the miss ratio is zero with enough accesses ($M >> \frac{N}{b}$).

3 MOTIVATION AND DESIGN

Whereas with proper hardware counters being exposed to profilers, guaranteeing precise profiling can be still tricky due to multiple potential pitfalls. First, newer and faster GPU products are released at a rapid pace, and there can be significant architectural changes on the chip layout, execution pipelines and memory system, etc. For example, as listed in Table 2, compared to AMD Radeon™ RX Vega GPU [3] (Vega 10), the leadership Radeon™ VII [5] (Vega 20) provides much higher memory bandwidth, by widening HBM interface and raising frequency. However, profilers are typically abstracted from specific architecture, and are thus required to support varied generations by handling the hardware divergences, which may be unnoticed for users.

Table 2: Comparison of two recent GPUs

GPU Product	AMD Radeon VII [5]	AMD Radeon RX Vega [3]
Architecture	Vega 20	Vega 10
Code Name	gfx906	gfx900
Technology	7 nm	14 nm
SP Cores	3840	4096
Base Clock	1400 MHz	1247 MHz
Peak FP32	13.8 TFLOPs	12.7 TFLOPs
Memory	HBM2 (16GB, 1TB/s)	HBM2 (8GB, 484GB/s)

Algorithm 1 Example P-chase program written in HIP [9]

```
1  /* setup: initialize array on CPU with the stride*/
2  for (i = 0; i < N; i++) {
3      array[i] = (_TYPE)((i+stride) % N);
4  }
5
6  /* copy array to GPU, launch kernel (not shown) */
7
8  /* kernel: serially reading array elements */
9  __global__ void pchase_RO (_TYPE* array,
10             int array_length, int m,
11             uint64_t* duration) {
12     _TYPE j = 0;
13
14     int k;
15     uint64_t time = clock();
16
17     for (k = 0; k < m; k++) {
18         j = array[j];
19     }
20
21     duration = clock() - time;
22 }
```

Meanwhile, for each GPU component (e.g., L1 cache), there usually exists tens of performance counters, requiring the proper ones to be selected for each specific characterizing goal. This is achieved by setting the configuration registers inside profiler. In addition, certain counters are designed to run in multiple modes to monitor varying events with minimized implementation overheads. For example, memory request counters can be incremented sequentially or intermittently over the elapsed cycles. Besides, high-level metrics are typically defined on the basis of one or more basic counters,

[1] Without specific comment, all sizes are in integer unit in the rest of paper (e.g., size of $4k$ is equivalent to 16KB).
[2] While optimized policies might be used to improve cache data usages, the regular mode is typically kept as a baseline to verify the cache logics, and it can be easily enabled by programming the configuration registers.

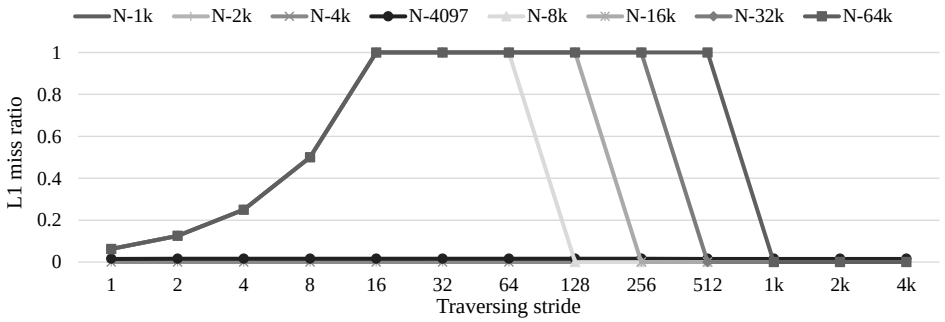
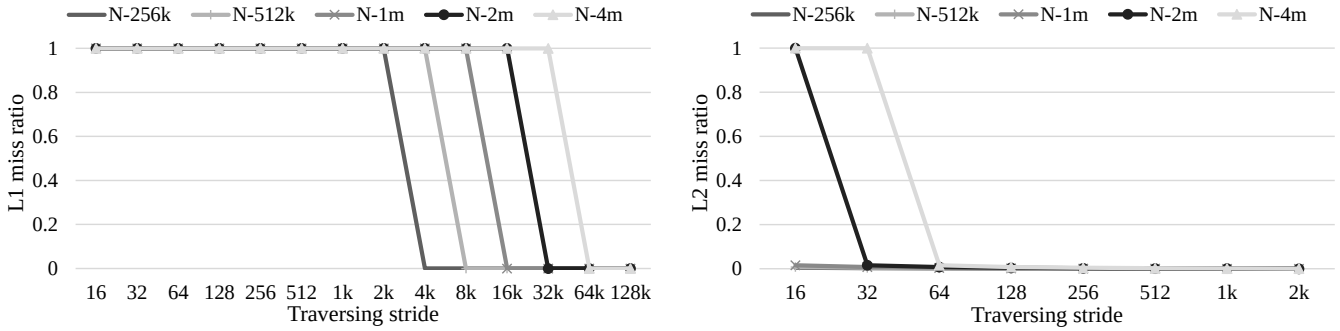


Figure 3: L1 miss ratios with varying array sizes and strides (512k total accesses).



(a) L1 miss ratio

(b) L2 miss ratio

Figure 4: L1 and L2 miss ratios with varying array sizes and strides (4m total accesses).

with the example of cache miss depending on counts of misses and total requests. To this end, appropriate counters and modes must be specified to ensure metrics' correctness. Further, counter collection and results presentation commonly involve intensive data processing, storage management and algorithm designs, which may easily introduce bugs or negligence.

Given the potential implementation issues, it is urgently demanded to validate the profilers at end-user level to make sure the collected results are faithfully representing the underlying resource usages. In this paper, we choose to verify profiling with focuses on cache and memory system, using the P-chase benchmark. Primarily, the read P-chase, as shown in Algorithm 1, will be frequently used in our studies with varied input parameters to stress certain memory levels.

We will first run the P-chase tests on Vega 10 GPU to see whether they can successfully obtain/verify key hardware details, which are critical parameters to direct the subsequent validation studies. Particularly, we collect cache counters and metrics by sweeping over a series of array sizes and strides, and then compare against the desired patterns discussed in Section 2.3.

3.1 L1 Cache

As aforementioned, there are only cold misses if the P-chase array fits in cache; and misses then change with respect to the accessing stride with larger array sizes. As such, we run the microbenchmark on L1 cache by gradually increasing the array sizes ($N = [64,\ 64k]$, i.e., 256B-256KB) and the traversing strides ($s = [1,\ N]$). The number of accesses to the array is fixed at 512k, which is much larger than the cold miss counts.

Figure 3 reports the miss ratios of different array sizes with varying strides. From the figure, we see that if $N \leq 4k$, the miss ratio is close to zero and is just contributed by cold misses; for larger Ns (e.g., $8k$ and $16k$), the miss ratios become much higher even with strides less than 16, and reaches 100% when stride is 16, indicating that each loaded line is evicted before being reused. Thus, the cache capacity should be no less than $4k \times 4B$ and no larger than $8k \times 4B$. To get the exact size, we can exhaustively search the range $N = [4k,\ 8k]$, and we do see that even $N = 4097$ leads to replacement misses, indicating that the capacity is 16KB (i.e., $4k \times 4B$). For $N \geq 8k$, the miss ratio curve exactly tallies with the expected one shown in Figure 2, thus giving $b = 16$ (i.e., 64B).

Putting together, the L1 configurations we get are 16KB capacity and 64B line size, which perfectly match the released hardware details [24] and parameters reported by the `rocminfo` command [12].

3.2 L2 Cache

To characterize L2 cache, we keep using the prior testing kernel. However, considering that L2 involves more cold misses and extra instruction cache misses, we enlarge the number of accesses from $512k$ to $4m$. Meanwhile, to exclude the filtering effects of L1, we apply strides larger than 16, which indicates the cache line size.

The miss ratios of L1 and L2 are reported in Figure 4. As shown in Figure 4(a), if stride is in $[16,\ 2k]$, all requests are missing on L1 for the chosen array sizes (256k, 512k, ... , 4m). And thus, we just narrow the stride range down to $[16,\ 2k]$ to observe L2 miss behaviors, which is as illustrated in Figure 4(b).

From Figure 4(b), we see that N-$2m$/$4m$ show regular miss patterns, similar to L1 covered in Figure 3. Replacement misses are not happening until the array size goes beyond $1m$, indicating the L2 capacity is 4MB [24].

4 VALIDATION TESTS AND RESULTS

The P-chase runs on L1 and L2, as covered in Section 3, demonstrate that the benchmark is capable to demystify cache parameters, as presented in [25, 32], which are valuable to direct test settings to control the access behaviors for validation purpose. As a result, we can flexibly configure the testing kernels to expose varying access patterns, which can be predicted beforehand, onto each cache or memory level, and then profile the configured kernel with specified counters and metrics. After getting both the predicted (i.e., ground truth) and profiled results, we can then compare to see whether the profiling is properly working.

With the validation methodology, we choose to vary array sizes and strides of the P-chase kernels to verify cache and memory profiling, in terms of misses, data size and access latency. The tests are performed on the Vega 10 and Vega 20 GPU architectures with recent release of ROCm v3.3 [8]. While identical tests are performed on both GPUs, the data from newer Vega 20 will be primarily reported and analyzed.

4.1 Cache Misses

Cache misses are generally determined by the chosen array size and stride, and thus we run validation with varied inputs to generate different misses. For the 16KB (i.e., 4k) L1, the selected configurations are array sizes $N = 64$ and $N = 64k$ with strides in $[1,\ N]$. The profiled counts of L1 misses are reported in Figure 5(a), which shows miss counts as calculated in Section 2.3. Note that for lower strides (i.e., $s \leq 16$), small array ($N = 64$) keeps constant number of misses (i.e., cold-start misses), whereas large array's ($N = 64k$) induced misses change with respect to strides.

Regarding L2 testing, as illustrated in Figure 4, we choose larger array sizes ($N > 4k$, which is L1 capacity) and strides ($s > 16$, i.e., cache line size) to force all requests to reach onto L2, thus excluding L1 noises. Like L1 tests, both small and large arrays are chosen, ranging from $N = 256k$ to $N = 4m$. From Figure 5(b), we see that $s = 256$ always causes cold-only misses, giving a constant miss ratio across the studied array sizes. Differently, miss counts of $s = 16$ shows a straight line (i.e., constant miss ratio) when $N < 1m$ but jumps at $N = 2m$ with all requests are missing, confirming that L2 capacity is $1m$ (i.e., 4MB).

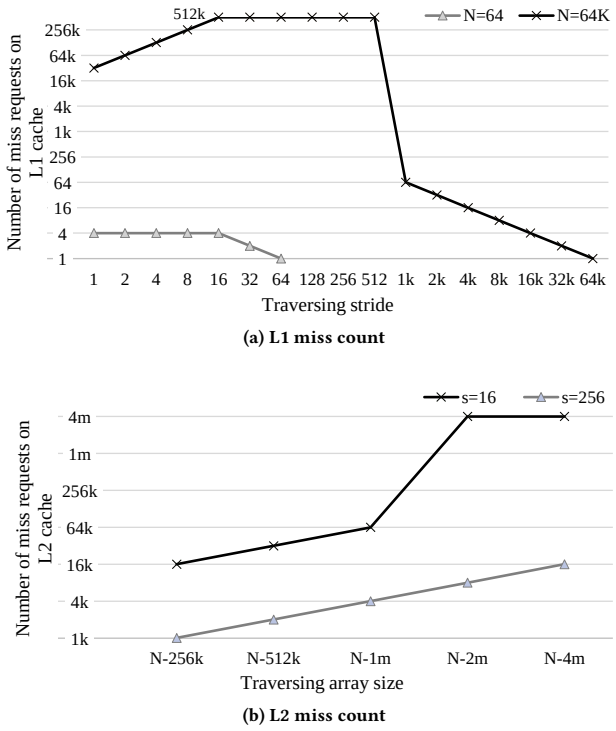


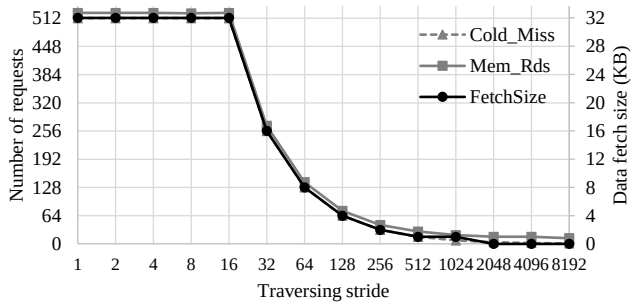
Figure 5: Profiled L1 and L2 misses of P-chase, with varying array sizes and traversing strides.

Observation: default policy of L2 is not LRU. The above validation tests are performed after programming the configuration register to enable basic LRU mapping, as mentioned in Section 2.3. However, the same test runs can help identify L2's non-LRU policy if the register is restored to the default value, which applies an optimized addressing to effectively distribute requests among cache lines. With the policy, cache behavior is not as regular as reported in Figure 5(b). For example, replacement misses may happen on smaller arrays like N-$512k$ and N-$1m$. For profiler validation purpose, we simply programmed the configuration register to utilize LRU to make cache show clear patterns. However, for normal usages, we suggest to adapt the cache policy with respect to application behaviors to maximize performance gains.

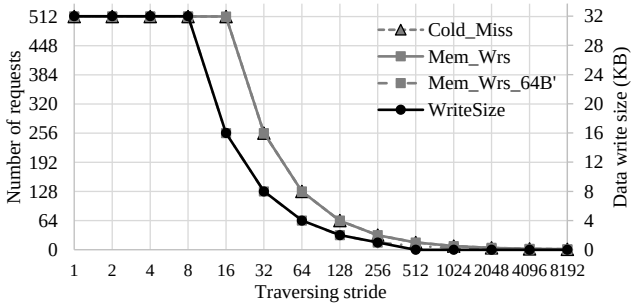
4.2 Memory Traffic

In a GPU, main memory is being shared by all the CUs, and it is of limited capacity and throughput. Therefore, profiling and optimizing memory usage is critical to tune applications' performance. Targeting at the traffic related counters, we craft tests to examine memory data fetch and data write size.

Given that memory data size is decided by the cache misses, we thus configure the tests to issue known numbers of memory requests. For ease of validation, we further set the array with a size falling between L1 capacity and L2 capacity, indicating that the array fits into L2 and thus all memory requests are just contributed by the cold misses in the initial warm-up phase. Specifically, the



(a) Memory read data



(b) Memory write data

Figure 6: Correlation studies of profiled memory read and write data to the expected misses (`Cold_Miss`) and request counts (`Mem_Rds/_Wrs`).

array is sized as $N = 8k$, and traversed by $512k$ total requests with varying strides. To separately examine read and write sizes, we repeated the tests on two different kernels (one issues only reads, and the other issues only writes).

The results are as shown in Figure 6. For read, Figure 6(a) correlates `FetchSize` with `Cold_Miss` and `Mem_Rds`, which are presenting expected cache cold misses and profiled memory read requests, respectively. Apparently, the curves are well matching with each other, conveying that the profiling counters and metrics are well developed and processed to capture the read traffic. Differing from read, write in Figure 6(b) shows mismatched `Cold_Miss` and `Write_Size`, where written data is less than expected. By examining the detailed counters, we found that write requests can be of either 32B and 64B granularity, and if we calculate the effective 64B writes (i.e., `Mem_Wrs_64B'`), then we see the write size is just as expected.

Observation: Vega 20 owns more memory partitions (MPs) than Vega 10. The traffic metrics smoothly passed the validation on Vega 10 GPU, but failed on Vega 20 by surprisingly reporting significantly lower amounts, indicating that somehow a fraction of requests failed to be captured by the profiler. Through careful examination of the profiled data, we identified that more MPs were added on Vega 20 to achieve much wider interface and higher throughput [3, 5], but the traffic metrics are still based on elder Vega 10, causing the extra MPs accidentally escaped the profiling. As a fix [3], the traffic metrics are revised to uniquely define for each platform. The MP difference identified here is just an example of the constantly changing GPU hardware components, which highlights the importance to validate profiling results on every new architecture.

4.3 Access Latency

While GPUs are considered to have high latency-hiding ability thanks to the massive parallelism, memory latency is still critical to improve overall performance and boost resource utilization [18]. Thus, latency counters are frequently used to characterize GPU applications to seek optimization opportunities.

Considering that latency values are largely dependent on the applications and runtime environments, we thus keep using the P-chase kernel and utilize the provided latency counters to obtain the latency of each memory level, and then verify the values with cross validation. Available latency counters `cntA` and `cntB` are measuring the round-trip delays from memory pipeline to L1 and from L1 to L2, respectively. As listed in Table 3, dedicated array sizes and strides are selected to respectively stress L1, L2 and Memory: a). tests T0 - T2 are to obtain the latency of servicing requests on each level (i.e., V_{L1}, V_{L2} and V_M illustrated in Figure 7); b). tests T3 - T5 are measuring mixed latency as described in the table. With the values obtained from T0 - T2, we can then manually calculate the expected average latency based on the miss ratios on each level, and then compare against the profiled data to validate.

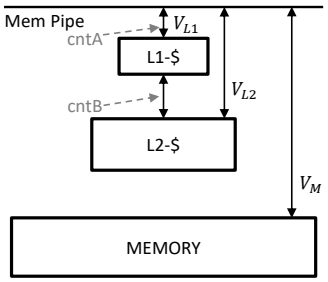


Figure 7: Access latency to cache and memory, and the associated counters.

In the validation, we first run tests T0 - T2 to collect latency values at program level using built-in `clock()` function, and then profile the latency numbers. The comparison demonstrates similar relative values among the cache and memory levels. Then, with profiling tests T3 - T5, we get the measured average latency values, which is then verified against expected ones based on the fraction of requests serviced on each level. As reported in Table 4, the tests show that the profiled latency values are of no more than 3% difference from the expected ones, conveying that the latency counters and metrics are working correctly among the studied GPU architectures.

Observation: Vega 20 is of shorter access latency than Vega 10. The tests demonstrate that the studied delays on Vega 20 are

[3]The fix has been incorporated into rocprof v2.8+ to customize the traffic metrics for Vega 20 [7].

Table 3: Tests to obtain and validate latency of cache and memory

Test ID	Test Notation	Setting <s, N, M>	Description
T0	ALL_L1_HIT	<16, 4k, 4m>	All hits on L1 (miss: L1≈0%)
T1	ALL_L2_HIT	<16, 8k, 4m>	All misses on L1, but hits on L2 (miss: L1≈100%, L2≈0%)
T2	ALL_L2_MISS	<16, 2m, 4m>	All misses on L1 and L2 (miss: L1≈100%, L2≈100%)
T3	L1_L2_MIX	<8, 8k, 4m>	Half hits on L1, half on L2 (miss: L1≈50%, L2≈0%)
T4	L1_MEM_MIX	<8, 2m, 4m>	Half hits on L1, half on mem (miss: L1≈50%, L2≈100%)
T5	L2_MEM_MIX	<32, 2m, 4m>	Half hits on L2, half on mem (miss: L1≈100%, L2≈50%)

visibly lower than those on Vega 10. Contributing factors might be the optimized micro-architectures, and also the widen memory interface.

Table 4: Latency validation results

Test ID	Expected	Profiling Difference
T3	$50\% \times V_{L1} + 50\% \times V_{L2}$	1.1%
T4	$50\% \times V_{L1} + 50\% \times V_{M}$	2.7%
T5	$50\% \times V_{L2} + 50\% \times V_{M}$	2.9%

4.4 Additional Studies

L1 is write-through: while focusing on kernels with only reads or writes, we also ran tests with mixed reads and writes to observe cache behaviors. The studies show that all writes are missing on L1 but may hit on L2 instead, indicating that the L1 is defaulted to be write-through.

L2 is address sliced with interleaving: in the studies, we examined the request distribution among L2 slices (i.e., partitions) by sweeping over strides in [1, *array size*]. With small strides, the requests are evenly distributed among the slices; with larger strides, certain slices gradually stop receiving requests, and if stride is large enough, all requests are routed to one single slice, implying the existence of address interleaving.

5 RELATED WORK

Computer designers and programmers rely on a wide set of tools to explore design alternatives and to tune application performance. The CPU world, after decades of evolution, has lots of mature tools including profilers (e.g., Pin [23], Instruction Sampling [2]) and debuggers (e.g., GDB [20]). Nonetheless, GPUs are still actively building the counterparts to unleash the capabilities of massive resources.

With GPU architectural complexities on the rise, developing highly efficient software are becoming significantly critical and challenging, motivating a plethora of work on program verification and performance profiling. For program correctness, multiple tools have been developed to verify the source codes of kernels [16, 19, 33]. To aid performance tuning, lots of software- [31] and hardware-based [11, 26] profiling tools were provided to identify bottlenecks and optimizations.

While analysis correctness is urgently needed, very few studies have been performed to verify the profiling tools. The most relevant work is Sen's assessment methodology [30], which is to rate the quality of power profiling mechanism. Differently, our goal is targeting at validating performance profiling with microbenchmarks.

In this paper, we are using microbenchmarks for validation. While the similar benchmarks were commonly adopted to reverse-engineering hardware details, they are instead used by us to predict program behaviors. As a result, the prior dissecting works [13, 22, 25, 32] are orthogonal to our proposed validation.

6 CONCLUSION AND FUTURE WORK

Profiling tools are heavily used to tune GPU programs, and thus are required to be operating correctly across architectures. Accordingly, there is an urgent need for validation to assist the construction of GPU profiling infrastructures. To address the issue, this paper presents a practical testing prototype *DELTA* using fine-grained microbenchmarks. The preliminary results on cache and memory imply that the validation is very helpful to identify incorrectness of profilers. In the future, we plan to extend the tests to support the all new RDNA architecture [6] and cover additional components (e.g., execution pipelines and virtual memory), with a wide spectrum of microbenchmarks and dedicated kernels.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] J. Alsop and M. Sinclair, *et al.* 2019. Optimizing GPU Cache Policies for MI Workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 243–248.
[2] AMD. 2007. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. https://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf (accessed: 2020-08).
[3] AMD. 2017. AMD Radeon RX Vega Graphics. https://www.amd.com/en/products/graphics/radeon-rx-vega-64 (accessed: 2020-05).
[4] AMD. 2017. "Vega" Instruction Set Architecture Reference Guide. https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf (accessed: 2020-05).
[5] AMD. 2019. AMD Radeon VII Graphics Card. https://www.amd.com/en/products/graphics/amd-radeon-vii (accessed: 2020-05).
[6] AMD. 2019. RDNA Architecture. https://www.amd.com/system/files/documents/rdna-whitepaper.pdf (accessed: 2020-05).
[7] AMD. 2019. ROCProfile Metrics. https://github.com/ROCm-Developer-Tools/rocprofiler/blob/amd-master/test/tool/metrics.xml (accessed: 2020-05).
[8] AMD. 2020. AMD ROCm Release. https://github.com/RadeonOpenCompute/ROCm (accessed: 2020-06).
[9] AMD. 2020. HIP Programming Guide. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html (accessed: 2020-05).

[10] AMD. 2020. ROCm Open Ecosystem. https://www.amd.com/en/graphics/servers-solutions-rocm (accessed: 2020-05).
[11] AMD. 2020. ROCm Profiler. https://github.com/ROCm-Developer-Tools/rocprofiler/blob/amd-master/doc/rocprof.md (accessed: 2020-05).
[12] AMD. 2020. rocminfo. https://github.com/RadeonOpenCompute/rocminfo (accessed: 2020-05).
[13] S. S. Baghsorkhi and I. Gelado, *et al.* 2012. Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors. In *ACM SIGPLAN Notices*. vol. 47, no. 8, pages 23–34.
[14] D. A. Beckingsale and J. Burmark, *et al.* 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81.
[15] C. Bertolli and S. F. Antao, *et al.* 2014. Coordinating GPU Threads for OpenMP 4.0 in LLVM. In *LLVM Compiler Infrastructure in HPC*. 12–21.
[16] A. Betts and N. Chong, *et al.* 2012. GPUVerify: A Verifier for GPU Kernels. In *Object-oriented Programming, Systems, Languages, and Applications (OOPLSA)*. 113–131.
[17] X. Chen and L.-W. Chang, *et al.* 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 343–355.
[18] S. Dublish and V. Nagarajan, *et al.* 2019. Poise: Balancing Thread-Level Parallelism and Memory System Performance in GPUs Using Machine Learning. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 492–505.
[19] B. Ferrell and J. Duan, *et al.* 2019. CUDA au Coq: A Framework for Machine-validating GPU Assembly Programs. In *Design, Automation, and Test in Europe (DATE)*. 474–479.
[20] GNU. 2008. Debugging with GDB: The GNU Source-Level Debugger. http://docs.adacore.com/live/wave/gdb-9/pdf/gdb/gdb.pdf (accessed: 2020-08).
[21] A. V. Gorshkov and M. Berezalsky, *et al.* 2019. GPU Instruction Hotspots Detection Based on Binary Instrumentation Approach. In *IEEE Transactions on Computers*. vol. 68, no. 8, pages 1213–1224.
[22] Z. Jia and M. Maggioni, *et al.* 2012. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. In *Technical Report*. 1–66.
[23] C.-K. Luk and R. Cohn , *et al.* 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*. 190–200.
[24] M. Mantor and B. Sander. 2017. AMD's Radeon Next Generation GPU Vega10. In *The Symposium on High Performance Chips (HotChips)*. 1–44.
[25] X. Mei and X. Chu. 2012. Dissecting GPU Memory Hierarchy through Microbenchmarking. In *IEEE Transactions on Parallel and Distributed Systems*. vol. 28, no. 1, pages 72–86.
[26] Nvidia. 2020. CUDA Profiler. https://docs.nvidia.com/cuda/profiler-users-guide/index.html (accessed: 2020-05).
[27] M. O'Connor and N. Chatterjee, *et al.* 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 41–54.
[28] ORNL. 2019. Frontier Supercomputer. https://www.olcf.ornl.gov/frontier/ (accessed: 2020-05).
[29] ORNL. 2019. Summit Supercomputer. https://www.olcf.ornl.gov/summit/ (accessed: 2020-05).
[30] D. Sen and N. Imam, *et al.* 2018. Quality Assessment of GPU Power Profiling Mechanisms. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 702–711.
[31] O. Villa and M. Stephenson, *et al.* 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 372–383.
[32] H. Wong and M.-M. Papadopulou, *et al.* 2010. Demystifying GPU Microarchitecture through Microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 235–246.
[33] Y. Xing and B.-Y. Huang, *et al.* 2018. A Formal Instruction-level GPU Model for Scalable Verification. In *International Conference on Computer-Aided Design (ICCAD)*. 1–8.