Are Your GPU Atomics Secretly Contending?

Peter Maucher
Karlsruhe Institute of Technology
Germany

Nick Djerfi
Karlsruhe Institute of Technology
Germany

Lennard Kittner
Karlsruhe Institute of Technology
Germany

Lukas Werling
Karlsruhe Institute of Technology
Germany

Frank Bellosa
Karlsruhe Institute of Technology
Germany

Abstract

GPU applications use atomic operations to coordinate data access in highly parallel code. However, relying on previous experiences and due to limited documentation, programmers resort to guidelines instead of concrete metrics to limit potential performance influences.

In this paper, we introduce a GPU memory-subsystem microbenchmark suite for analyzing GPU atomic operations. Based on the benchmark results, we discuss two particular guidelines, namely: "use only one thread per warp to access an atomic" and "place two atomic variables on different cache lines to avoid contention." We demonstrate where these guidelines are effective and where actual hardware behavior diverges.

CCS Concepts: • **Computer systems organization → Processors and memory architectures**; • **Computing methodologies → Shared memory algorithms**; **Graphics processors**; • **Software and its engineering → Process synchronization**; • **General and reference → Measurement**.

Keywords: GPU, Atomic Operations, Atomic Contention, Synchronization, Microbenchmarks

ACM Reference Format:
Peter Maucher, Nick Djerfi, Lennard Kittner, Lukas Werling, and Frank Bellosa. 2025. Are Your GPU Atomics Secretly Contending?. In *13th Workshop on Programming Languages and Operating Systems (PLOS '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3764860.3768338

1 Introduction

GPUs provide highly parallel computing capabilities and are widely used in high-performance and AI applications [10]. Various parallel GPU algorithms [3, 13–16, 29, 30, 38] used in areas such as medicine [17], image processing [9, 11], AI training [37], or operating systems [25] implement fine-grained communication between threads with atomic operations. On GPUs, atomic operations provide read-modify-write operations on single words (up to 128 bits) [21]. The design characteristics of atomic operations on GPUs differ from those on CPUs due to the higher level of parallelism. For example, modern GPUs incorporate dedicated *atomic units* at the global cache level (L2 or L3), which serialize atomic operations [5, 6, 24, 26]. This unique role means that the correct usage of atomic operations can strongly influence the performance of GPU applications, especially since the degree of parallel accesses and thus contention strongly influences runtime behavior. Prior works have explored the performance implications [4, 6, 13, 19, 32]. However, we find that these studies are limited in scope, with imprecise benchmarks on outdated hardware.

As a result of both limited vendor documentation and limited prior work, programmers instead rely on guidelines to reduce the effects of contention in their programs. We built a suite of microbenchmarks to evaluate some of these guidelines, which we used to construct synchronization primitives for accelerating operating system components by moving them to the GPU. We found that these guidelines lack the nuances that actual hardware exhibits.

In this paper, we present results from our suite of microbenchmarks executed on both AMD and NVIDIA GPUs. Our main goal is to help programmers develop a better understanding of atomic operation performance. The results in this paper and in the additional benchmarks show some unexpected behavior with atomic operations and memory fence operations which can influence design rules. Thus, we encourage developers to challenge and benchmark their assumptions regarding atomic operations on GPU. We show how to transfer guidelines into microbenchmarks, and how to determine whether these guidelines hold.

Together with this paper, we publish our our suite of benchmarks[1]: We provide benchmarks for atomic operations, memory fences and special memory operations. Each of these categories is tested with different access pattern to develop a description of the behavior of the memory subsystem. However, we leave the construction of a model of the underlying hardware for future work.

PLOS '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2225-7/25/10
https://doi.org/10.1145/3764860.3768338

[1] https://github.com/KIT-OSGroup/GPUAtomicContention

In our discussion, we focus on atomic operations given their use even without memory fences. In particular, in addition to measuring baseline performance, we discuss the following guidelines to increase the performance of atomic operations:

- *Avoid accessing a single atomic variable with multiple threads in the same warp:* On the GPU, multiple threads work in lockstep in an organizational unit called a warp [21]. To reduce the number of threads contending on an atomic variable, previous works build warp-level synchronization schemes to manually coalesce accesses [4, 32]. However, hardware units have some capacity to reduce the serialization costs of parallel accesses, thus sidestepping this manual work under certain circumstances.
- *To avoid contention, place your atomic variables in different cache lines*: Atomic operations on GPUs are usually implemented in atomic units attached to global (L2 or L3) caches [6, 24, 26]. Therefore, one might assume that placing atomic variables into different cache lines routes the requests to different atomic units, which should avoid contention resulting from requests to two different atomic variables hitting the same unit. Again, this common assumption [19, 25, 32] proves somewhat correct. However, *cross-contention* between independent atomic variables spaced at least one cache line apart can still occur in multiple, unexpected cases.

We present one possible solution when implementing future primitives, which exploits an unintuitive speedup when accessing adjacent atomic variables from the same warp: Real hardware is able to parallelize these accesses from the same warp, so distributing the communication over multiple atomic variables densely packed in memory reduces the number of required accesses while keeping said accesses fast.

To improve performance without requiring in-depth hardware knowledge, programmers can use higher-level abstractions. In parallel applications, examples include synchronization primitives like barriers and mutexes [4, 28, 31, 32]. Building such abstractions falls onto the operating systems and programming languages communities. We also show some design implications for higher-level abstractions given the hardware behavior.

The remainder of this paper is structured as follows: First, we introduce GPU atomics and present prior work on GPU atomic performance and usage. Afterwards, we show the benchmarks designed to study the assumptions presented above. Finally, we show the benchmark results and discuss the implications for programmers using the aforementioned guidelines, and conclude this paper.

2 Background

GPUs are much more optimized for throughput than CPUs, which leads to differences in compute architecture and memory layout. These differences inform the hardware design as well as the application usage of GPU atomics.

To facilitate high parallelism with minimal hardware, 32 threads are bundled into a *warp*, with each thread in a warp sharing the instruction pointer, thus executing the same instruction. Classically, inside a warp, no synchronization was needed as all instructions run in lockstep. However, NVIDIA recently introduced Independent Thread Scheduling [21], which somewhat relaxed this guarantee. Multiple warps make up a *thread block*, an independent unit of work. Each thread block shares a slice of *local memory*, a small, directly addressable, and fast cache, and can synchronize execution using hardware-assisted barriers. In hardware, each block is scheduled onto the GPU equivalent of a CPU core, called a streaming multiprocessor (SM) on NVIDIA [21] or a compute unit (CU) on AMD [2]. GPU programs, called *kernels*, consist of multiple blocks organized into a *grid*, which are distributed over the entire GPU. One important limitation here is that the programming APIs do not expose synchronization primitives between different thread blocks, which programmers must construct themselves from atomic operations and optionally fences [21].

AMD and NVIDIA GPUs both expose a relaxed memory order consistency model, which requires memory fences for memory consistency. Additionally, to further reduce memory traffic, fences and atomic operations have scopes [22] that limit the set of threads influenced by the fence or that observe the result of an atomic operation. These scopes are informed by both the memory and compute hierarchy. *Local* scope is limited to a single thread block, *Device* covers the whole GPU, including caches and video memory (VRAM), and *System* includes the remaining devices in a compute system, including the CPU DRAM. For consumer GPUs like the ones we tested, this requires a pass through the PCIe bus, and not all operations are supported on all GPUs, especially atomic operations. We focus on device-level atomic operations, as in-depth local memory studies already exist [34] and the device level is most important for kernel-level synchronization. We also exclude fences in this discussion to reduce complexity and since atomic operations are useful without fences. However, our benchmark suite does contain in-depth analysis of fence behavior.

3 Related Work

We present prior work on atomic operations for GPU programs.

Various previous authors have benchmarked atomic operations on GPUs before. Stuart et al. and Elteir et al. discuss older architectures that differ significantly from modern GPUs [7, 32]. These two, as well as Jia et al., [12] discuss only

a single vendor, and do not go into the specifics of contention on modern GPUs.

Our work follows a similar approach to McKee et al., as they study modern GPUs from different vendors using microbenchmarks [19]. However, the details differ quite significantly: They use the higher-level Vulkan API [8] instead of the low-level HIP API [1], and therefore need to use more coarse-grained timings. Additionally, in Vulkan, compilers have more influence on the code compared to HIP. Hence, while they find results supporting the guidelines mentioned above, they do not identify the complexities we discover.

Given the importance of synchronization, various higher-level primitives built on GPU atomic operations exist [4, 28, 31, 32]. These works benchmark their implementations, but on a higher level compared to our microbenchmarks.

Jin et al. discuss the trade-off between inner-warp synchronization and atomic contention [13], and Dalmia et al. discuss further assumptions that influence GPU program design using atomic operations [4].

GPUs are an attractive target for offloading operating system functionality. Previous works in this space used GPU atomic operations extensively for synchronization: In BaM [25], Qureshi et al. use atomic locks to manage a GPU-side SSD cache for direct storage access. Pandey et al. [23] implement GPU-side logging and checkpointing to persistent memory using atomic operations. Yeh et al. [36] incorporate atomic operations into a warp scheduling algorithm for low-latency tasks with limited parallelism. Additionally, Silberstein et al. [27] suggest using atomic operations via the PCIe bus for CPU-GPU communication. Atomic operations through the PCIe bus were not available when that paper was published, but some of our tested GPUs (A1, A2) now support this feature. Maucher et al. use PCIe atomics in GPU4FS [18], to let CPU-side clients communicate with the file system, which is running on the GPU.

4 Microbenchmarks

To study the alignment between programmer assumptions and actual hardware behavior, we employ microbenchmarks specifically targeted at the atomic subsystem. We discuss both the benchmark setup and the challenges inherent in measuring on the GPU.

4.1 Challenges

To precisely measure the runtime of atomic operations, one would attempt to time a single access. However, measuring time on the GPU is difficult: On NVIDIA, a counter of executed instructions is available [21]. While this technique is used in previous works [20, 35], we require actual timing information as device-level atomic operations may leave the clock domain of a single streaming multiprocessor. Additionally, this functionality is not present on AMD GPUs. An alternative solution would be to time the complete kernel execution using the functionalities offered by the APIs. Again, this technique is widely used in prior work [7, 13, 16, 33], but is orders of magnitude less precise than the alternatives.

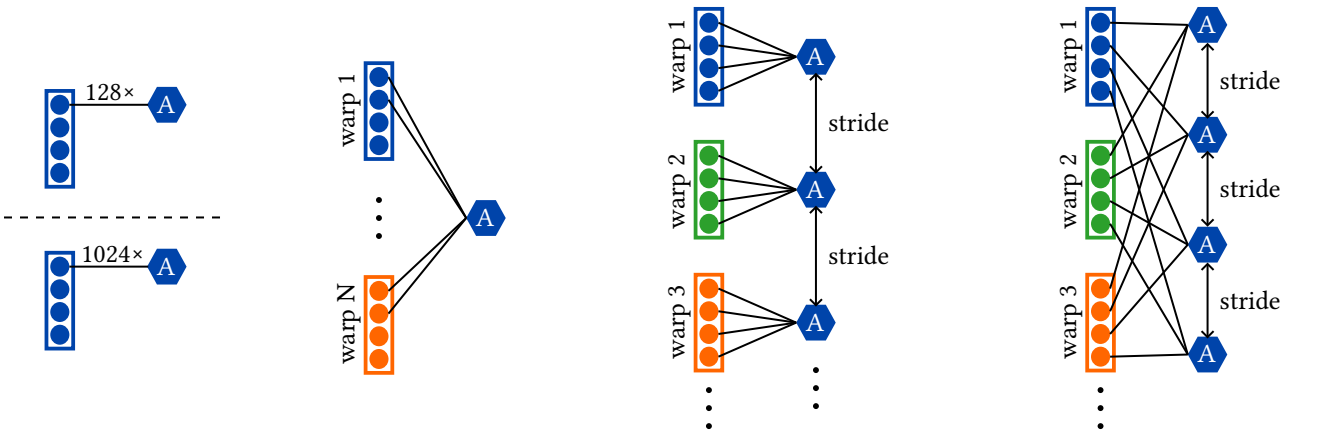
Instead, we use a hardware feature present in both AMD and NVIDIA GPUs, which provide real-time clocks in their hardware. The resolution differs slightly between the manufacturers, with 40 ns on AMD and 32 ns on NVIDIA, resulting in a resolution of approximately 100 clock cycles. However, the resolution is too coarse to time a single access, with the fastest we measured being 2.4 ns, so we need to repeat the accesses multiple times per measurement. To determine the validity of this approach, we devise a benchmark as described in Section 4.2. We find that atomic operation timing does not scale linearly on all GPUs, instead showing *self-contention*, where one thread can contend with its own requests. The best-case results we measure are close to those reported in literature for similar architectures [12]. However, even in scenarios showing self-contention in our microbenchmarks, we expect the same effect to appear in real applications.

4.2 Baseline

To establish trust in our microbenchmarking framework as well as to quantify the effects of contention, we construct a baseline test with the aim of being as uncontended as possible. For this measurement, we use an unrolled loop of 128 and 1024 `atomicAdd(1)` accesses to a device-level atomic variable. We run this benchmark with a single thread of a single warp, as seen in Figure 1 (a). We measure atomic accesses instead of load/store cycles to avoid interference from compiler and hardware optimizations. To avoid contention, a single thread of a single warp accesses a single device-level atomic variable, with the rest of the GPU being idle. We compare the time per access between 128 and 1024 iterations to check whether accesses are actually uncontended, as then both microbenchmarks should show the same result.

4.3 Threads per Warp Scaling

The first guideline we study is *Avoid accessing a single atomic with multiple threads in the same warp*. However, some GPUs may implement optimizations and access coalescing in hardware, as suggested by Dong et al. [6], in which case, programmers can avoid manual coalescing. To investigate this guideline, we test a fixed number of atomic operations on the same atomic variable for all threads of all warps. We vary the access distribution between the number of warps, as presented in Figure 1 (b). Hence, by timing the accesses, we can infer the slowdown due to contention. Crucially, we include the case of one warp and one thread repeatedly accessing the atomic variable, which extends the data points given by the baseline. To avoid race-conditions with threads finishing before others start and thus not actually contending, we use multiple accesses per thread.



(a) Baseline (b) Threads per Warp (c) Direct Cross-Contention (d) Scattered Cross-Contention

Figure 1. Overview over our microbenchmarks.

4.4 Atomic Cross Contention

In addition to contention tests on a single atomic variable, we also investigate cross-contention between multiple atomic variables, depending on their placement in memory. Again, we use a fixed number of atomic operations, but we change the access pattern: We distribute the atomic variables in memory with a given stride, and assign each thread to access a single atomic variable. On all of our GPUs, a cache line is 128 B, so the access time should decrease for a stride of at least 128 B. We execute this test in two configurations as shown in Figure 1 (c) and (d): First, with adjacent threads accessing the same atomic (c), and second, with adjacent threads always accessing different atomic variables (d) as initially suggested by McKee et al. [19].

4.5 Benchmarks Not Discussed in this Paper

Our benchmark suite includes more benchmarks than we discuss in this paper. First, we implement all benchmarks in this paper for all applicable scopes, typically both local scope and system scope through the PCIe bus. We optionally add memory fences for acquire or sequential memory consistency and thread fences to synchronize the warp. We also include a study of a simple lock constructed from atomic operations, implemented both using *bitwise or* and *compare and swap*. These two options enable comparisons in performance and hardware behavior. We found that our initial implementation deadlocked on GPUs A1 and A2, possibly due to missing Independent Thread Scheduling [21], and so our implementation contains a version working on all our tested GPUs. We benchmark these locks with added fences for acquire and release consistency to gather some understanding of fence performance in scenarios containing in-memory IPC.

To work out whether our GPUs correctly support atomic operations through the PCIe bus [27], we built a simple test that accesses the same memory location in DRAM atomically from both the GPU and the CPU, which we also include in our released code.

Other than fences, we provide benchmarks for additional memory operations. We implement two features implemented by the GPUs but not directly exposed in the programming APIs: store operations with explicit relaxed or sequential memory consistency semantics and an MMIO store operation, which bypasses the caches and disables write combining, allowing for proper device interaction from the GPU (e.g., as used by BaM [25]). Lastly, we also time accesses to memory marked as volatile to explore the actual performance impact.

5 Results

We show the results of the benchmarks introduced above.

5.1 Test Systems

We test on four different GPUs, two from AMD and two from NVIDIA, with each having a different GPU architecture to get results that are transferable. We pick consumer-class GPUs due to their abundance, which we list in Table 1.

Table 1. Evaluated GPUs

ID	GPU	#SM/CU	max clock
A1	AMD RX 6950 XT	80	2310 MHz
A2	AMD RX 7900 XTX	96	2500 MHz
N1	NVIDIA RTX A4500	56	1650 MHz
N2	NVIDIA RTX 4070	46	2475 MHz

We distribute the GPUs over two different host systems to enable parallel benchmarking of all GPUs, even with a limited number of PCIe slots per system. We do not expect the different hosts to influence the results, as all benchmarks run completely on the GPU. System H1 with an AMD EPYC

9124 CPU and 64 GB of DRAM hosts GPUs A1 and N1, and system H2 with an Intel Xeon Silver 4215 CPU and 128 GB of DRAM hosts GPUs A2 and N2.

5.2 Baseline

In Figure 2, we establish a baseline: We show 1024 different measurements in the order that they were conducted. In each measurement, a single thread issues either 128 or 1024 atomicAdd(1) operations on a single atomic variable as described in Section 4.2. The two manufacturers show very different behavior here: On NVIDIA, an uncontended access takes between 2.4 ns (N1) and 3.5 ns (N2), with no measurable difference between 128 and 1024 accesses. Additionally, the results are very stable, with only a small increase in runtime possibly related to downclocking.

On AMD, however, the measurements for 128 repetitions are higher on average, at about 6.5 ns (A1) and 7 ns (A2) respectively, but the measurements are quite noisy and can fall below 3 ns for both GPUs. Surprisingly, when using 1024 repetitions, the GPUs take significantly longer per access, about 85 ns and 100 ns respectively, and the results are again noisy. These results suggest some form of self-contention in the AMD GPUs that also appears in later plots.

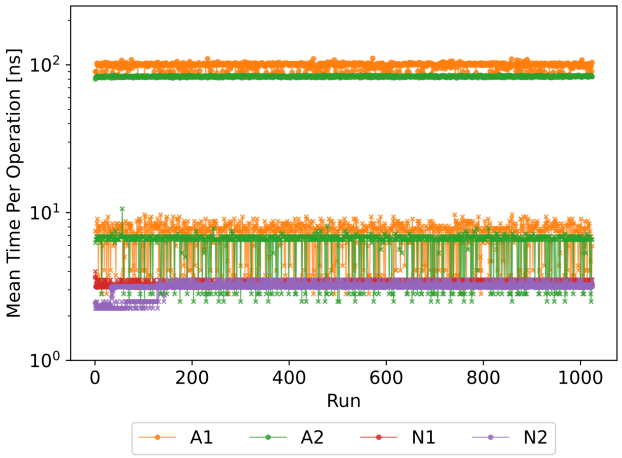


Figure 2. Baseline timing over 1024 measurements on our four GPUs, with 128 (x) and 1024 (•) iterations of one thread accessing one atomic variable (see Figure 1 (a)).

From these results, we draw three conclusions: First, uncontended atomic operations are quite fast on both studied manufacturers, so uncontended accesses to atomic variables alone do not hurt performance significantly. Secondly, there is an apparent possibility for a thread to contend with itself, an effect that NVIDIA's GPUs will also show in later experiments. Third, while somewhat noisy for AMD, the results are still within a few nanoseconds, so we will show means over 10 runs for the remainder of this paper.

5.3 Threads per Warp Scaling

We present results for both A2 in Figure 3 and N2 in Figure 4, as these are the faster GPUs per vendor in our test set, with the other two closely following these results. Except with very few threads, the runtime is proportional to the number of threads, but is independent of the thread distribution into warps. Both GPUs reach an upper limit in contention at about 4096 active warps, which we attribute to the GPU limit of warps that can be concurrently scheduled. As GPU threads run to completion, later scheduled threads do not contend with earlier threads and the timing stays constant.

For 32 or fewer threads, both AMD and NVIDIA GPUs show a flat line, which hints at a hardware mechanism hiding serialization costs for parallel accesses of up to 32 requests. Given that this behavior is warp-independent, we assume this mechanism is located in the atomic unit, not in the GPU core. Additionally, the plots suggest self-contention also for NVIDIA: The baseline established earlier was approximately 3 ns, however the lower limit (for both GPUs) here is 30 ns.

The plots also show some unexpected behavior, demonstrating that simple models do not suffice for a complete performance picture. N2 is somewhat slower for very few threads, a pattern not observed in N1. A2 shows two notable characteristics: Firstly, one warp with 32 threads is slower than one warp with 16 threads, but four warps with eight threads each are as fast as one warp with 16 threads. A1 shows a similar behavior with 64 threads, where two warps with 32 threads are slower than one warp with 32 threads but as fast as four warps with 32 threads. We conclude that A2 and A1 indeed have 32 or 64 independent atomic units, respectively, but experience difficulty distributing these resources to very few warps. Additionally, both AMD GPUs show similar groupings of 32, 64 and 128 warps which do not scale as linearly as other warp groups. When special-casing for AMD GPUs, one can exploit this grouping behavior as an area of relatively consistent contention independent of the number of warps.

Using these plots, we mostly confirm the common wisdom of reducing contention on the programmer side. However, the hardware includes some optimizations that we can exploit: In the case of lightly contended atomic variables, taking extra care to manually consolidate and later spread accesses may not be worth the additional time, especially if only a few threads in the warp take part in the access.

5.4 Atomic Cross Contention

Having studied the effects of a single atomic variable, an additional consideration for synchronization primitives is to avoid cross-contention between otherwise independent atomic variables. We show our results in Figure 5 as the mean of 10 runs. We observe measurement noise of approximately a factor of two per measurement, but the results are stable across repeated executions. We expect the access time

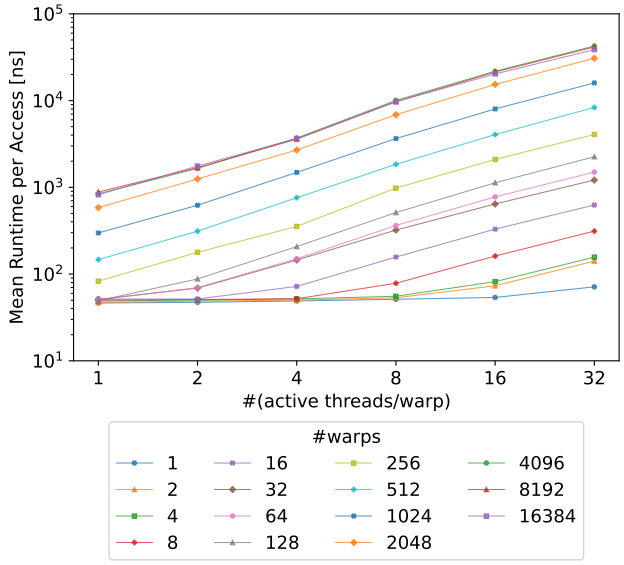


Figure 3. Timing per atomic access on GPU A2, with varying numbers of threads per warp accessing the same atomic variable (see Figure 1 (b)).

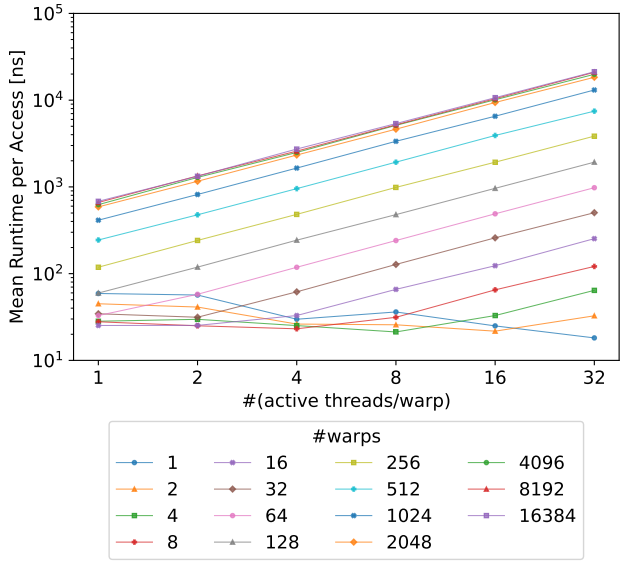


Figure 4. Timing per atomic access on GPU N2, with varying numbers of threads per warp accessing the same atomic variable (see Figure 1 (b)).

to decrease for a stride of 128 B between atomic variables as they are then located on different cache lines. However, contention actually only decreases for a stride of 256 B, with 128 B showing the same behavior as 0-64 B. Similarly, contention between four atomic variables only decreases for a stride of 128 B. This implies that only warps 0 and 1 as well as 2 and 3 contend with each other, but not all four.

Another unexpected result is that contention on AMD GPUs rises again with a stride of 4096 B for two warps, with more warps showing an increase in contention as soon as two warps have a memory offset of 4096 B. Our explanation is that the GPU uses the least significant 12 bits as a key to select the respective atomic unit, with each warp accessing the same unit and leaving the other units unused, an effect also described by van den Braak et al. [34]. We verify this assumption by rerunning one benchmark for non-power-of-two memory strides, where the load is spread evenly and the timings are as low as expected.

In conclusion, two atomic variables should be at least 256 B apart to avoid cross contention between them. Additionally, on AMD hardware, avoid using high powers of two as the stride to distribute the load over all available atomic units.

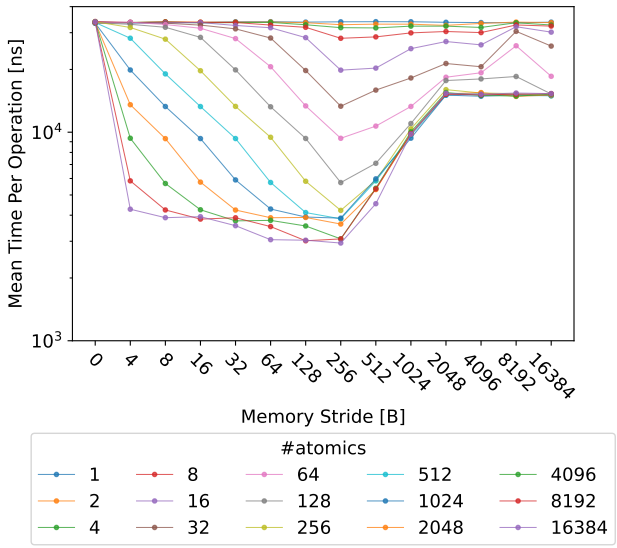


Figure 5. Timing for an atomicAdd on GPU A2 when accessing one to many atomic variables with different memory strides (see Figure 1 (c)).

We also study the same number of atomic variables with each thread in a warp accessing different atomic variables, as shown in Figure 6. First, note that for a memory stride of zero, in both Figure 5 and Figure 6, all threads contend on the same atomic variable, and the results shown are in fact identical. However, for a stride of four and eight bytes, the access time is quite a bit faster per thread as compared to the linear case. This behavior implies that the atomic unit can parallelize accesses by one warp if the accesses hit different memory areas. This hardware optimization evident in all four GPUs offers the possibility for fast communication: one warp can efficiently communicate individual state to

other warps in dedicated atomic variables, as long as each atomic operation hits the same unit, and other warps can then request this information. By limiting the amount of atomic units each warp accesses and by communicating via other warps using a shared atomic variable, the overall level of contention decreases.

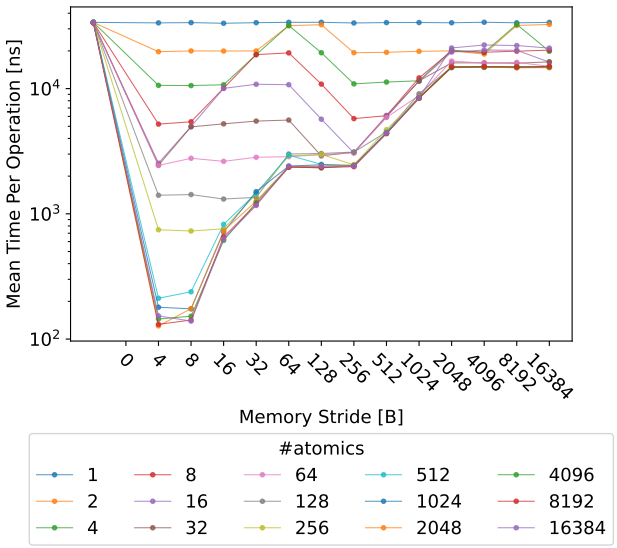


Figure 6. Timing for an atomicAdd on GPU A2 when accessing one to many atomic variables with different memory strides, but each warp accessing multiple atomic variables (see Figure 1 (d)).

6 Discussion and Future Research

We study two guidelines, none of which prove to be entirely correct or entirely incorrect.

First, using more threads in a warp does lead to increased contention. However, there is a base level where a small number of threads use hardware acceleration and show no slowdown. This observation is interesting when designing primitives that are not expected to be highly contended, like spinlocks for short critical sections.

Second, we disprove the common assumption that atomic variables on different cache lines do not contend, but show that maintaining sufficient distance does indeed decrease contention. However, even though a distance of 256 B consistently avoids contention for our four tested GPUs, we encourage testing the proper alignment for other GPUs. Additionally, we emphasize taking care to avoid other, unexpected sources of cross-contention, like placing an atomic variable at the same offset on different memory pages.

Third, we show that accessing multiple atomic variables with the same warp is fast, so using multiple atomic variables and spreading the work is an important building block for future primitives.

Having shown these results, we see directions for future research. Our analysis focuses on consumer GPUs, but evaluating the benchmarks on enterprise GPUs will show whether the behavior discussed in this paper is universal, as we would expect given that the microarchitectures are documented to be similar. Additionally, our work aims to highlight issues with the performance of atomic operations on the GPU. While we investigate some phenomena, we leave a microarchitectural deep dive, possibly including performance counters and hardware-specific microbenchmarks, as future work.

We see another direction for future research in using the results of this paper. First, given the obscurity and scarcity of proper documentation, conduct an extended study of assumptions and their real-world performance to offer a full set of applicable, nuanced guidelines. Additionally, our results can help to properly implement higher-level atomic primitives. We show that such a primitive should contain a fast path for uncontended atomics, should spread their communication over multiple atomic variables and should spread these variables in particular patterns: to reduce contention, spread atomic variables 256 B apart, and to increase the amount of information communicated without performance cost, use multiple threads in a single warp to access adjacent atomic variables. Our benchmarking suite already contains a straightforward spinlock implementation as a starting point.

7 Conclusion

We investigate common programmer guidelines regarding GPU behavior of atomic operations on both AMD and NVIDIA GPUs. We show that, while not entirely incorrect, real hardware exhibits different behavior. We hope that this paper spawns better abstractions tuned to actual hardware. Furthermore, our results emphasize the need to benchmark atomic interactions in GPU applications intensely.

References

[1] AMD. 2025. Heterogenious Interface for Portability. https://rocm.docs.amd.com/projects/HIP/en/latest/index.html

[2] AMD. 2019. *RDNA 1 White Paper.* https://web.archive.org/web/20190821193406/https://www.amd.com/system/files/documents/rdna-whitepaper.pdf

[3] Federico Busato and Nicola Bombieri. 2016. An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures. 27, 8 (2016), 2222–2233. doi:10.1109/TPDS.2015.2485994

[4] Preyesh Dalmia, Rohan Mahapatra, Jeremy Intan, Dan Negrut, and Matthew D. Sinclair. 2023. Improving the Scalability of GPU Synchronization Primitives. 34, 1 (2023), 275–290. doi:10.1109/TPDS.2022.3218508

[5] Preyesh Dalmia, Rohan Mahapatra, and Matthew D. Sinclair. 2022. Only Buffer When You Need to: Reducing on-Chip GPU Traffic with Reconfigurable Local Atomic Buffers. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022-04). 676–691. doi:10.1109/HPCA53966.2022.00056

[6] Rongcui Dong and Sreepathi Pai. 2025. Modeling Utilization to Identify Shared-Memory Atomic Bottlenecks. In *Proceedings of the 17th Workshop on General Purpose Processing Using GPU* (New York, NY, USA, 2025) *(Gpgpu '25)*. Association for Computing Machinery, 14–20. doi:10.1145/3725798.3725801

[7] Marwa Elteir, Heshan Lin, and Wu-Chun Feng. 2011. Performance Characterization and Optimization of Atomic Operations on AMD Gpus. In *2011 IEEE International Conference on Cluster Computing* (2011-09). 234–243. doi:10.1109/CLUSTER.2011.34

[8] Khronos® Group. 2022. *Khronos Vulkan Registry.* https://registry.khronos.org/vulkan/

[9] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. 2013. An Optimized Approach to Histogram Computation on GPU. 24, 5 (2013), 899–908. doi:10.1007/s00138-012-0443-3

[10] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach.* Elsevier.

[11] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. 2020. Compiling Generalized Histograms for GPU. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020-11). 1–14. doi:10.1109/SC41405.2020.00101

[12] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. arXiv:1804.06826 [cs.DC] https://arxiv.org/abs/1804.06826

[13] Zheming Jin, Jeffrey Vetter, and Jeffrey Vetter. 2023. A Study on Atomics-Based Integer Sum Reduction in HIP on AMD GPU. In *Workshop Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France and New York, NY, USA, 2023) *(ICPP Workshops '22)*. Association for Computing Machinery, Article 6. doi:10.1145/3547276.3548627

[14] Mohamed Esseghir Lalami and Didier El-Baz. 2012. GPU Implementation of the Branch and Bound Method for Knapsack Problems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (2012-05). 1769–1777. doi:10.1109/IPDPSW.2012.219

[15] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France and New York, NY, USA, 2010) *(Isca '10)*. Association for Computing Machinery, 451–460. doi:10.1145/1815961.1816021

[16] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why Gpus Are Slow at Executing Nfas and How to Make Them Faster. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland and New York, NY, USA, 2020) *(Asplos '20)*. Association for Computing Machinery, 251–265. doi:10.1145/3373376.3378471

[17] William Chun Yip Lo, Tianyi David Han, Jonathan Rose, and Lothar Lilge. 2009. GPU-accelerated Monte Carlo Simulation for Photodynamic Therapy Treatment Planning. In *Therapeutic Laser Applications and Laser-Tissue Interactions IV* (2009) *(Therapeutic Laser Applications and Laser-Tissue Interactions IV)*. Optica Publishing Group, 7373_13. doi:10.1364/ECBO.2009.7373_13

[18] Peter Maucher, Lennard Kittner, Nico Rath, Gregor Lucka, Lukas Werling, Yussuf Khalil, Thorsten Gröninger, and Frank Bellosa. 2024. Full-Scale File System Acceleration on GPU. In *Tagungsband Des FG-BS Frühjahrstreffens 2024* (2024). Gesellschaft für Informatik eV, 10–18420.

[19] Devon McKee, Tylor Sorensen, Ishita Chaturvedi, Gurpreet Dhillon, and Sean Siddens. 2024. GPU Atomic Performance Modeling with Microbenchmarks. (2024). https://vulkan.org/user/pages/09.events/vulkanised-2024/vulkanised-2024-devon-mckee.pdf

[20] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU Memory Hierarchy through Microbenchmarking. arXiv:1509.02308 [cs.AR] https://arxiv.org/abs/1509.02308

[21] NVIDIA. 2025. *CUDA C++ Programming Guide.* https://docs.nvidia.com/cuda/cuda-c-programming-guide

[22] NVIDIA. 2025. *CUDA Memory Model.* https://nvidia.github.io/cccl/libcudacxx/extended_api/memory_model.html

[23] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: Leveraging Persistent Memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022) *(Asplos '22)*. Association for Computing Machinery, 142–156. doi:10.1145/3503222.3507758

[24] David Patterson. 2009. *The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges.* The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. https://www.nvidia.com.tw/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf

[25] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-initiated on-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2023) *(Asplos 2023)*. Association for Computing Machinery, 325–339. doi:10.1145/3575693.3575748

[26] Tamal Saha, Abhishek Rawat, and Minh Le. [n. d.]. Fermi - A Complete GPU Compute Architecture by NVIDIA. ([n. d.]). https://www.cs.virginia.edu/~skadron/cs6354_f09_processors/Fermi.pptx

[27] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. 41, 1 (2013), 485–498. doi:10.1145/2490301.2451169

[28] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs. In *2017 IEEE International Symposium on Workload Characterization (IISWC)* (2017-10). 239–249. doi:10.1109/IISWC.2017.8167781

[29] Erik Sintorn and Ulf Assarsson. 2008. Fast Parallel GPU-sorting Using a Hybrid Algorithm. 68, 10 (2008), 1381–1388. doi:10.1016/j.jpdc.2008.05.012

[30] Rafał Skinderowicz. 2016. The GPU-based Parallel Ant Colony System. 98 (2016), 48–60. doi:10.1016/j.jpdc.2016.04.014

[31] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Portable Inter-Workgroup Barrier Synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands and New York, NY, USA, 2016) *(Oopsla 2016)*. Association for Computing Machinery, 39–58. doi:10.1145/2983990.2984032

[32] Jeff A. Stuart and John D. Owens. 2011. Efficient Synchronization Primitives for Gpus. arXiv:1110.4623 [cs.OS] https://arxiv.org/abs/1110.4623

[33] Ryan Taylor and Xiaoming Li. 2010. A Micro-Benchmark Suite for AMD Gpus. In *2010 39th International Conference on Parallel Processing Workshops* (2010-09). 387–396. doi:10.1109/ICPPW.2010.59

[34] Gert-Jan van den Braak, Juan Gómez-Luna, Henk Corporaal, José María González-Linares, and Nicolás Guil. 2013. Simulation and Architecture Improvements of Atomic Operations on GPU Scratchpad Memory. In *2013 IEEE 31st International Conference on Computer Design (ICCD)* (2013-10). 357–362. doi:10.1109/ICCD.2013.6657065

[35] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU Microarchitecture through Microbenchmarking. In *2010 IEEE International Symposium on*

Performance Analysis of Systems & Software (ISPASS) (2010-03). 235–246. doi:10.1109/ISPASS.2010.5452013

[36] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-grained GPU Resource Virtualization for Narrow Tasks. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA and New York, NY, USA, 2017) *(PPoPP '17)*. Association for Computing Machinery, 221–234. doi:10.1145/3018743.3018754

[37] Huan Zhang, Si Si, and Cho-Jui Hsieh. 2017. GPU-acceleration for Large-Scale Tree Boosting. arXiv:1706.08359 [stat.ML] https://arxiv.org/abs/1706.08359

[38] Yongpeng Zhang, Frank Mueller, Xiaohui Cui, and Thomas Potok. 2009. GPU-accelerated Text Mining. In *Workshop on Exploiting Parallelism Using GPUs and Other Hardware-Assisted Methods* (2009). ACM Press New York, 1–6.