In the Trenches: An Exploratory Study of GPU Atomic Performance

Gurpreet Dhillon Spring 2024

1 Introduction

GPU performance is difficult to reason about and characterize as there is a huge variance in architectural design among GPUs across multiple vendors. In this paper, I will talk about my investigations into characterizing GPU atomic performance, with the goal of helping application programmers reason about its performance. Results are classified into three groups.

1. Performance Differences between Vulkan and CUDA
2. Stabilizing Atomic Throughput across Vendors
3. Android Results

Each of these groups represent investigations carried out, stemming from my overall goal to create a generalized atomic performance model and outline methods to structure atomics for optimal performance.

2 Motivation

The main intention was to create tooling to trace atomic accesses and extrapolate atomic performance on various devices. Though the project changed more into an exploratory study rather than an implementation of a generalized performance model, as we discovered more uncertainties in reasoning about GPU performance in frameworks Vulkan and CUDA.

With an aim to create a performance model, research and development in CUDA was done to work towards manually instrumenting the atomic accesses via redirection to callbacks. The initial CUDA implementation led to different trends from the existing work, which led to a further investigation of both frameworks.

3 Background

One of the most fundamental primitive operations for synchronization are atomic read-modify-write (RMW) instructions. On CPUs, these instructions have well-defined performance profiles consistent across many devices. However, due to the wide variance in architectural design as well as the complexities of scheduling and caching, atomic RMW performance on GPUs is highly variable and difficult to reason about as an application programmer.

My main project is a parameterized microbenchmark suite for characterizing performance profiles of RMWs on GPUs, written in two GPU frameworks, Vulkan and CUDA. The goal of the microbenchmark suites is to characterize GPU atomic RMW performance results, towards the direction of providing application programmers with the tools to reason about their performance.

```
for (uint i = 0; i < *iter; i++) {
    atomic_fetch_add_explicit(&res[index], 1,
    memory_order_relaxed);
}
```

(a) OpenCL kernel loop

```
for (uint i = 0; i < iter; i++) {
    atomicAdd(&res[index], 1);
}
```

(b) CUDA kernel loop

```
for (uint i = 0; i < iter; i++) {
    asm("red.global.add.u32 [%0], 1;" ::
    "l"(&res[index]));
}
```

(c) CUDA kernel loop (inline PTX)

Figure 1. Comparison of kernel setup in OpenCL and CUDA

3.1 Microbenchmarking Suite

The microbenchmark measures throughput of atomic RMWs at the occupancy bound for various thread access patterns and configurations for various parameters, including contention and padding. Vulkan is used to execute compute kernels on the GPU, and OpenCL [3] to write these Vulkan shaders, then transpiled through clspv [1], a Google-supported compiler, to SPIR-V [2], a binary IR to natively represent compute kernels. I have also developed the same microbenchmark in CUDA.

To measure RMW throughput, I execute tests at a fixed number of RMW iterations, recording the dispatch timing of how long it takes to complete a number of atomic operations. Trials are repeated for a sufficient number of iterations to obtain a consistent value. Each test is a short program which, given a value for contention, padding, thread access pattern and instruction type, will execute a preset number of RMW instructions, then measure the time taken to do so. This is then converted to a number for RMW throughput, measured in atomic operations per μs. Figure 1 shows kernel code for the benchmarks. In the kernel, each thread will execute an atomic fetch add operation for a preset number of RMW iterations, where the index represents the thread access pattern and the result buffer is preconfigured by the given value of contention and padding.

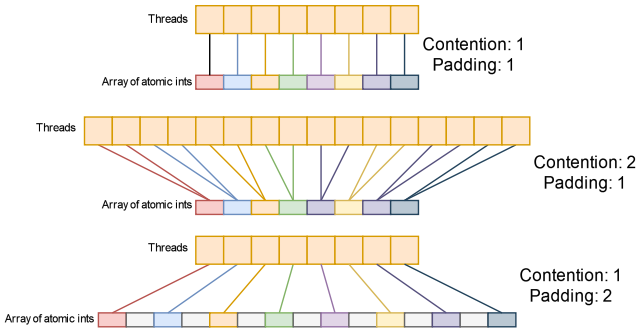


Figure 2. Atomic RMW configuration of parameters contention and padding

3.2 Parameters

Two parameters for microbenchmarking RMW performance, seen in figure 2. The first, padding, refers to the number of machine words between those accessed (which is 32 bits in all GPUs we examined). For example, if padding is set to 2, then 1 in every 2 atomic integers is accessed. The second, contention, refers to the number of threads contending on the same word. For example, if contention is set to 2, then for each word accessed, two threads will be attempting to perform an atomic RMW on that word. Both parameters being set to 1 indicates each thread accesses their own location and the locations are contiguous.

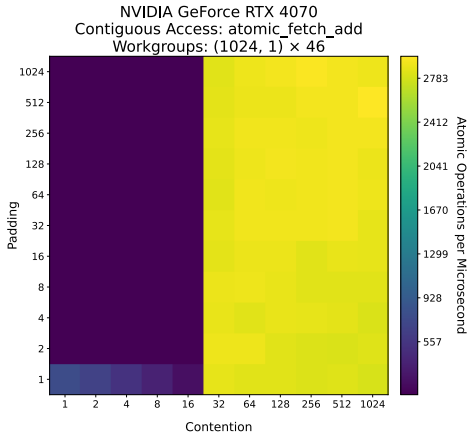
The test configurations sweep through predetermined ranges for contention and padding and create two-dimensional heatmaps for RMW throughput. For each configuration of contention and padding, indices into buffers of atomic integers corresponding with thread access patterns are calculated by the CPU and copied to GPU memory. In the test kernel, each thread retrieves its index corresponding to its thread ID from GPU memory and uses it to index into the aforementioned atomic integer buffer. These index calculations are done on the CPU to avoid compiler optimizations affecting atomic accesses, e.g., which should prevent the compiler from automatically coalescing atomic operations across threads.

4 Results

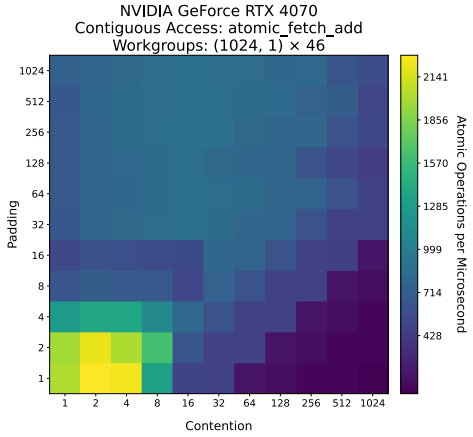
This section is divided into three parts: the first two pertain to the exploratory study carried out, plus a third section to detail an extension of the existing microbenchmark testing domain.

4.1 Performance Differences between Vulkan and CUDA

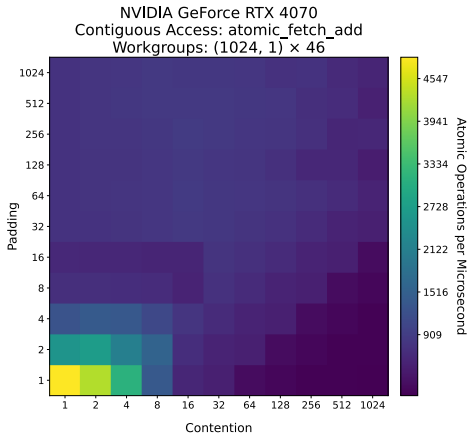
I ran the implementations from both Vulkan and CUDA on an NVIDIA GPU to outline the differences between the two frameworks. In both cases, threads were assigned to atomic integer locations in a contiguous mapping, e.g., where threads in the same warp would be the first to have contention. I made the implementations as close to each other as I can possibly could make them, however, I observe striking differences in performance, with the results being shown in Figure 3.



(a) Vulkan results



(b) CUDA results



(c) CUDA results (inline PTX)

Figure 3. Comparison of NVIDIA results

```
.func  (.param .b32 func_retval0) __uAtomicAdd(
        .param .b64 __uAtomicAdd_param_0,
        .param .b32 __uAtomicAdd_param_1)
{
        .reg .b32         %r<3>;
        .reg .b64         %rd<2>;
        ld.param.u64         %rd1, [__uAtomicAdd_param_0];
        ld.param.u32         %r1, [__uAtomicAdd_param_1];
        atom.add.u32         %r2, [%rd1], %r1;
        st.param.b32         [func_retval0+0], %r2;
        ret;
}
.func  (.param .b32 func_retval0) _ZN49_INTERNAL...(
        .param .b64 param_0,
        .param .b32 param_1)
 {
        .reg .b32         %r<3>;
        .reg .b64         %rd<2>;
        ld.param.u64         %rd1, [_ZN49_INTERNAL_..._param_0];
        ld.param.u32         %r1, [_ZN49_INTERNAL_..._param_1];
        {
        .reg .b32 temp_param_reg;
        .param .b64 param0;
        st.param.b64         [param0+0], %rd1;
        .param .b32 param1;
        st.param.b32         [param1+0], %r1;
        .param .b32 retval0;
        call.uni (retval0),
        __uAtomicAdd, (param0, param1);
        ld.param.b32         %r2, [retval0+0];
        }
        st.param.b32         [func_retval0+0], %r2;
        ret;
}
.visible .entry _Z9fetch_addPjjS_PfS_(...) {
...
$L__BB1_2:
        ...
        _ZN49_INTERNAL...(param0, param1);
        ...
...
}
```

Figure 4. Generated atom PTX instruction

In Figure 3a, the Vulkan results suggest the GPU might be coalescing atomics across the entire warp. That is, once contention goes above 32, we see roughly a 32× increase in throughput. This suggests that the hardware might be combining the 32 operations into one operation, but only when every thread in the warp is performing the operation to the same atomic location. However, the results from CUDA, figure 3b and 3c, shows a completely different trend compared to Vulkan, where peak throughput occurs when threads from the same warp are accessing different (although contiguous) atomic locations. I was expecting to see the same trends from the Vulkan results as the CUDA implementation follows the same structure and mapping of index calculations done on the CPU. As a further precaution, optimizations were turned off for the compiler frontend and backend.

For further analysis, I generated PTX code from the CUDA implementation of figure 1b and believe that it's not being coalesced in the compiler level, seen in figure 4. The CUDA PTX appears to be just calling atomic add (which has a return value). Unlike from what was seen with the results from the atom instruction, reductions being what the hardware is coalescing was hypothesized, since they don't have a return value, and hence redesigned the CUDA kernel to perform an atomic reduction operation through directly accessing the instruction via inline PTX, seen in figure 1c. From this redesign, I thought that the `red` instruction in PTX might match the Vulkan results, but instead led to a similar trend to the initial CUDA kernel setup, seen in figure 3c. Hence, the CUDA results from both atom and red instructions lead to trends that don't match the Vulkan results.

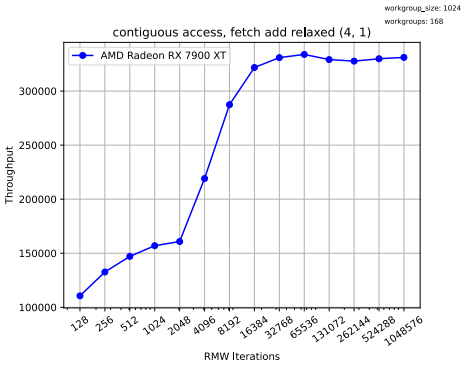
In the Heterogeneous Programming Lab, we reached out to Tyler's colleague at NVIDIA for an explanation of the differences in RMW performance between Vulkan and CUDA. He informed us that the compiler has a path where it will add code to dynamically detect that the addresses are all the same, and then do the coalesced atomic. Also that this optimization might not be enabled in CUDA. Furthermore, the compiler uses a voting system where all threads in the warp check to see if atomics are present and if their address is all the same, then branch to the coalesced path. With this understanding, a further examination is required for the Vulkan NVIDIA results. For example, testing the hypothesized voting system with thread instruction masks or an additional level of indirection calculated on the CPU.

4.2 Stabilizing Atomic Throughput across Vendors

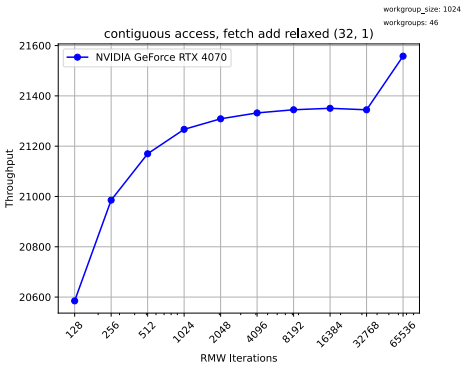
Table 1. Point of Stabilization across Vendors

GPU	Atomic RMW Iterations
NVIDIA GeForce RTX 4070	>128
AMD Radeon RX 7900 XT	16384
Intel Arc A770 Graphics	16384
Samsung Xclipse 920	>128
ARM Mali-G710	32768

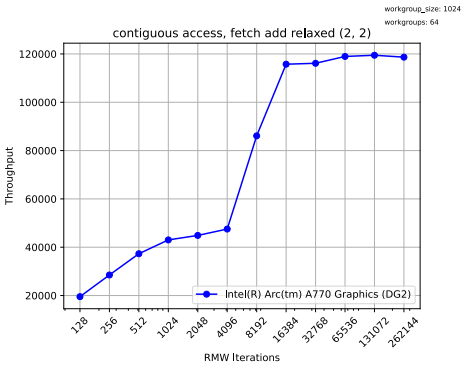
Following the observations and findings from Vulkan vs. CUDA, I also chose to further examine the collection of atomic throughput across discrete chips seen in figure 5 and across mobile devices seen in figure 6. I chose to focus on a single configuration of contention and padding that observes



(a) AMD results

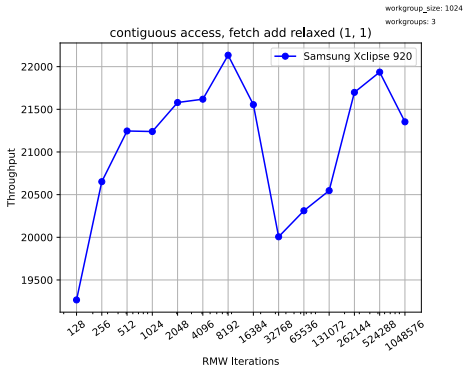


(b) NVIDIA results

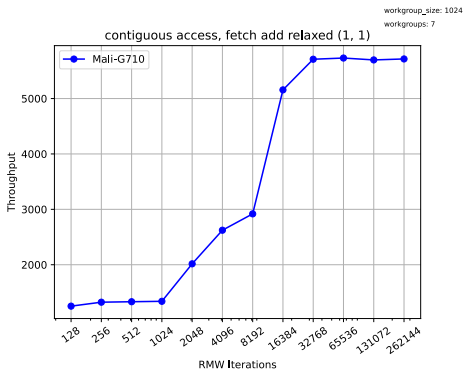


(c) Intel results

Figure 5. Comparison of stabilization at peak throughput across vendors



(a) Samsung results



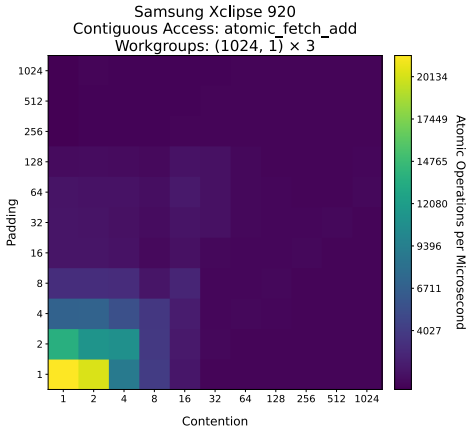
(b) ARM results

Figure 6. Comparison of stabilization at peak throughput across Android devices

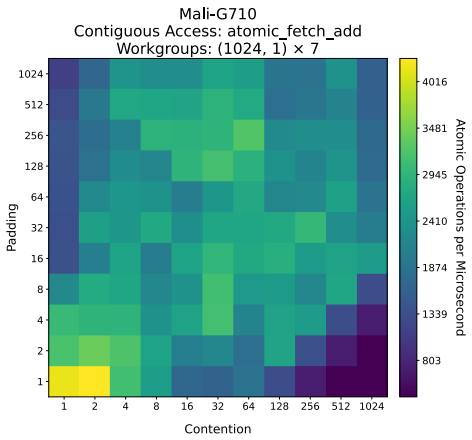
peak throughput on each GPU. At each pair, I run at occupancy bound for each RMW iteration. Initial values suggest some amount of kernel launch overhead was measured, so an increase in RMW iterations needs to be observed until the atomic throughput flatlines (suggesting stability).

In table 1, the number of RMW iterations needed for stability varies across GPUs, and that the number for every configuration of contention and padding within the same GPU will vary as well, as kernels that run longer (lower throughput) are more stable. The AMD, Intel, and ARM GPU all follow a similar trend where overhead is measured until a stability point is reached within 16384-32768 iterations, which might suggest similar characteristics with how their execution units handle atomics. The same goes with the NVIDIA and Samsung GPU as their trends lead to only needing 128 iterations (or fewer) to stabilize, with little to no overhead recorded, suggesting their robustness to atomic performance. It is clear that with this configuration (contiguous access pattern on top of an atomic fetch add), there are two distinct groups that can be drawn in terms of stabilization (not accounting for the comparison between atomic throughput numbers), which gets us a step closer in characterizing atomic performance. A further investigation is needed into the architecture specifics on why the trends are so distinct.

However due to these variations, microbenchmarking atomic performance would have to be redesigned to test an X amount of RMW iteration values per point of contention and padding until stability is found, per thread access pattern, atomic instruction, and GPU, which adds more complexity in reasoning about performance.



(a) Samsung results



(b) ARM results

Figure 7. Comparison of Android results

4.3 Android results

I ran the existing microbenchmark suite on two Android devices, extending the domain of testing to mobile GPUs. As seen in figure 7, data was gathered from the 2022 Samsung Xclipse 920 GPU with RDNA 2 architecture and from the 2021 ARM Mali-G710 GPU with Valhall architecture. Only two graphs under the same configuration are shown as 30+ graph configurations per GPU can be generated and would be difficult to display in the report. The stabilization tests were also investigated on these devices, seen in figure 6. The ARM Mali-G710 starts to experience stabilization at 32768 iterations and the Samsung Xclipse 920 GPU experiences stabilization at 128 iterations, as the subsequent measurements have a very small difference in atomic throughput. This suggests that the Samsung Xclipse 920 GPU is much more robust to performing atomic operations, as little to no overhead is recorded to find a stabilization point. Also to note, between the two competitors, Samsung Xclipse 920 reaches its stabilization point much faster and averages around 3x more atomic operations per μs than ARM Mali-G710.

5 Related work

Previous works have proposed efficient synchronization and atomic implementations, microbenchmarking techniques, and difficulties in reasoning about portable optimizations across GPUs

Stuart and Owens [5] revisit synchronization primitives like barriers, mutexes, and semaphores for GPUs, highlighting the inadequacy of previous implementations due to hardware and programming model disparities. They studied the roles of spin waiting and sleep waiting in each primitive and how their performance varies based on the machine abstraction.

Wong and Papadopoulou [6] explore microbenchmarking techniques and reverse-engineering approaches to uncover hardware features of GPU microarchitecture, benchmarks programmed in CUDA.

Sorensen, Pai, and Donaldson [4] discuss the methods in extending graph algorithm optimizations to various GPUs across multiple vendors, outlining the challenges involved in reasoning about the effects of optimizations across diverse architectures. Their GPU graph compiler implements rmw-coalescing as one of its optimizations, for exploring the impact of optimization transformations on graph algorithms executing across GPUs and to measure tradeoffs between specialization and portability across parameterization of architecture, applications, and input.

6 Conclusion

I shared my experiences arising from researching on the development of a generalized performance model with an aim to help GPU programmers reason about atomic performance in their applications. My work was divided into three categories: performance differences between Vulkan and CUDA, stabilizing atomic throughput across vendors, and Android results. The outcome of this investigation shows that the overall story for atomic RMW performance on GPUs is very complex. These findings are difficult to assess and require architecture specific knowledge to reason about. However, this work contributes towards the direction of creating a generalized atomic performance model and shows that there are opportunities to optimize atomic dependent workloads on GPUs through reordering or rewriting atomics in a way that provide performance optimizations.

6.1 Source code

The following are links to the microbenchmark suite which contains the Vulkan/CUDA implementation and instructions to setup: https://github.com/ucsc-chpl/epiphron/tree/main/benchmarks/atomic-rmw and https://github.com/ucsc-chpl/gpu-atomic-rmw-microbenchmark.

References

[1] Google. Clspv. https://github.com/google/clspv, 2023.

[2] Khronos Group. SPIR-V specification version 1.6, revision 1. https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.html, 2021.

[3] Khronos Group. The opencl c specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html, 2022.

[4] Tyler Sorensen, Sreepathi Pai, and Alastair F Donaldson. One size doesn't fit all: Quantifying performance portability of graph applications on gpus. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 155–166. IEEE, 2019.

[5] Jeff A Stuart and John D Owens. Efficient synchronization primitives for gpus. *arXiv preprint arXiv:1110.4623*, 2011.

[6] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, 2010.