Characterizing CUDA and OpenMP Synchronization Primitives

Brandon Alexander Burtchell
Department of Computer Science
Texas State University
San Marcos, USA
burtchell@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, USA
burtscher@txstate.edu

***Abstract*—Over the last two decades, parallelism has become the primary method for speeding up computer programs. When writing parallel code, it is often necessary to use synchronization primitives (e.g., atomics, barriers, or critical sections) to enforce correctness. However, the performance of synchronization primitives depends on a variety of complex factors that non-experts may be unaware of. Since multiple primitives can typically be used to complete the same task, choosing the best is often non-trivial. In this paper, we study the performance impact of these factors by measuring the throughput of OpenMP and CUDA synchronization primitives along multiple dimensions. We highlight interesting and non-intuitive behavior that software developers should be aware of when writing parallel programs.**

***Index Terms*—multithreading, parallel programming, synchronization, scalability, CUDA, OpenMP**

I. Introduction

Maximizing parallelism has become the primary method of speeding up program execution. Until the early 2000s, the performance of a program was mostly determined by the clock frequency of the processor, which, according to optimistic interpretations of Moore's Law [1] and Dennard/MOSFET scaling [2], was predicted to double approximately every 1.5 years. However, physical limitations have slowed this scaling [3] [4] and led to the multi-core revolution. As a consequence, the scalability of a program now depends on the fraction of the code that can run in parallel [5], making increasing the parallelism crucial—especially in the context of high-performance computing.

When writing parallel code, it is often necessary to use *synchronization* to prevent data races or to enforce a specific ordering. Parallel-programming APIs provide commonly-used parallelism constructs (e.g., barriers and atomics), which we refer to as *synchronization primitives*. These primitives come with their own complexities, such as whether an atomic operation implies a memory fence. Moreover, the performance of synchronization primitives may be impacted by hardware factors other than the number of cores. Yet, developers are not always aware of these ever-changing and system-dependent complexities, which can lead to programs that do not exploit the available parallelism well, resulting in poor performance and/or scaling.

Software developers often have to choose between multiple primitives that accomplish the same task. Each implementation provides a tradeoff between generality and performance. However, a programmer may not be aware of these tradeoffs and select a suboptimal solution. Such issues are exacerbated when writing parallel programs for both CPUs and GPUs. Fundamental architectural differences between these devices mean that primitives with the same net result may be implemented differently and, therefore, exhibit non-trivial behavioral and performance differences.

Seasoned programmers can internalize many of these intricacies over time, but there are too many possible pitfalls for non-experts to maximally exploit the potential benefits of parallelism. Hence, we need to identify the behavior and measure the performance of important synchronization primitives to guide non-experts in effectively choosing the best primitives for a given scenario.

This paper studies the throughput behavior of many commonly-used OpenMP and CUDA synchronization primitives across various parameters. We perform our measurements on several systems with CPUs and GPUs from different vendors and generations. To the best of our knowledge, a similar study has not been conducted before.

This paper makes the following main contributions.

- It describes a testing framework to measure the execution time of single synchronization primitives.
- It presents a detailed throughput analysis of various primitives across several parameters and systems.
- It analyzes the behavior of important synchronization primitives and provides recommendations for software developers.

The code and results are open-source and available [6].

The rest of this paper is structured as follows. Section II gives an overview of OpenMP and CUDA. Section III describes our measurement approach. Section IV outlines our experimental methodology. Section V presents and analyzes the results. Section VI discusses related work. Section VII provides a summary and conclusions.

This work has been supported in part by the National Science Foundation under Award Number 1955367 and by an equipment donation from NVIDIA Corporation.

II. Background

This section briefly introduces the parallel-programming APIs—OpenMP and CUDA—and their synchronization primitives that we evaluate. It also provides an example that demonstrates how the APIs evolve and how the performance behavior of different primitives can be non-intuitive.

A. OpenMP

OpenMP [7] is a popular open-source API that enables multithreading in C/C++ and Fortran programs. In C/C++, this is mostly accomplished with preprocessor directives. The rest of this subsection discusses the commonly-used OpenMP synchronization primitives that we evaluate.

1) Barrier: A barrier is one of the most basic synchronization primitives. It blocks every thread until all other threads have also reached it. Barriers can be explicitly called or are implicitly present after many other primitives.

2) Atomics: Atomics ensure that only one thread can read/ write a variable at a time. OpenMP includes several flavors of atomic operations, including update, read, write, and capture. Atomic read and write are self-evident. Atomic update reads a variable, performs an operation on it, and writes the result back to the variable—all atomically. Atomic capture is similar to atomic update, but a second variable "captures" the value of the updated variable (e.g., `v = x++`).

3) Critical Section: A critical section is a section of code that restricts execution to one thread at a time. As critical sections enforce serial execution, they are generally used to prevent data races on complex operations. Internally, OpenMP implements the critical section by having each participating thread acquire and later release a shared lock. The locking overhead can be substantial, making critical sections slow for simple operations.

4) Memory Flush: The *memory consistency model* specifies the allowed reorderings of reads or writes of one variable relative to the reads or writes of other variables in parallel programs. In cases where a thread reads and writes multiple variables without synchronization, the compiler and the hardware may reorder the accesses to the different variables, potentially breaking the code if the programmer intended for one variable (e.g., a flag) to guard the other variable (e.g., so that a consumer thread will only access the other variable after it is ready). *Memory fences* prevent such reorderings by ensuring that all memory operations before the fence finish before any memory operations start that appear after the fence. In OpenMP, a memory fence is called a *flush*. Flushes are implicit after many OpenMP primitives. They can also be invoked explicitly.

B. CUDA

CUDA [8] is a proprietary API and language based on C/C++ that allows programmers to write non-graphics code for NVIDIA GPUs. GPU code is written as *kernels*—special functions that are compiled for the GPU. Kernels are launched with a specified number of *thread blocks*, a logical group of up to 1024 threads. In hardware, these thread blocks are executed on *streaming multiprocessors* (SMs), which are essentially vector processors with their own cores, registers, and shared memory/L1 cache. The specifications of an SM vary with the architecture, and the number of SMs per GPU vary from device to device. CUDA allows for multiple blocks to run simultaneously on an SM, provided that the blocks do not exceed the resources of the SM. Notably, the maximum threads per SM can be more than the maximum threads per block.

GPUs achieve high throughput from their single-instruction-multiple-thread behavior. On the SM, thread blocks are composed of *warps*, or contiguous sets of 32 threads. A warp of threads executing the same instruction(s) will execute simultaneously—maximizing parallelism. However, if at least one thread in a warp is not executing the same instruction as the others (e.g., branching statements), this will result in *thread divergence*, which can significantly impact performance.

The rest of this subsection discusses the commonly-used CUDA synchronization primitives that we investigate.

1) Syncs: CUDA includes barriers at multiple granularities. For example, `__syncthreads()` synchronizes all threads in a block, and `__syncwarp()` synchronizes all threads in a warp.

2) Atomics: CUDA provides many atomic operations (add, sub, max, min, etc.). If an operation is not provided, the programmer can often emulate it using an atomic compare-and-swap (`atomicCAS()`), which compares the current value at an address with the expected value, and exchanges it with a new value if the comparison passes. `atomicExch()` is similar but skips the comparison.

3) Thread Fences: In CUDA, a memory fence is implemented as a `__threadfence()` call, which ensures that all memory accesses before the fence occur before any memory accesses after the fence, across the entire device. Variants such as `__threadfence_block()` and `__threadfence_system()` also exist, which change the scope to just a thread block and the entire system (CPU and GPU), respectively.

4) Warp-Level Functions: CUDA includes warp shuffle functions, which synchronize a defined subset of threads in the warp and have them exchange values (without accessing memory) according to the function's exchange pattern. For example, `__shfl_sync()` can broadcast a value from one thread to all other participating warp threads.

CUDA also includes warp voting functions, which take a value from each thread in the warp and compare it with zero. The results of the comparison are then reduced and broadcasted to each participating thread.

C. CUDA Example

We now provide a small example illustrating how the CUDA API evolved and how the performance of different synchronization primitives can be non-intuitive. Listing 1 presents five distinct ways of implementing a maximum reduction in CUDA. Many other solutions exist. We show these five to make several points.

```
const int lane = threadIdx.x % warpSize;
const int i = threadIdx.x + blockIdx.x * blockDim.x;
__shared__ int block_result;

// Reduction 1: since compute capability 1.3
if (i < size) atomicMax(&result, data[i]);

// Reduction 2: since compute capability 3.0
if (__any_sync(~0, i < size)) {
  int val = (i < size) ? data[i] : INT_MIN;
  for (int j = warpSize / 2; j > 0; j /= 2)
    val = max(val, __shfl_xor_sync(~0, val, j));
  if (lane == 0) atomicMax(&result, val);
}

// Reduction 3: since compute capability 6.0
if (threadIdx.x == 0) block_result = INT_MIN;
__syncthreads();
if (i < size) atomicMax_block(&block_result, data[i]);
__syncthreads();
if (threadIdx.x == 0) atomicMax(&result, block_result);

// Reduction 4: since compute capability 8.0
if (threadIdx.x == 0) block_result = INT_MIN;
__syncthreads();
if (__any_sync(~0, i < size)) {
  int val = (i < size) ? data[i] : INT_MIN;
  val = __reduce_max_sync(~0, val);
  if (lane == 0) atomicMax_block(&block_result, val);
}
__syncthreads();
if (threadIdx.x == 0) atomicMax(&result, block_result);

// Reduction 5
int thread_result = INT_MIN;
if (threadIdx.x == 0) block_result = INT_MIN;
__syncthreads();
for (int j = i; j < size; j += blockDim.x * gridDim.x)
  thread_result = max(thread_result, data[j]);
atomicMax_block(&block_result, thread_result);
__syncthreads();
if (threadIdx.x == 0) atomicMax(&result, block_result);
```

Listing 1: Five implementations of a reduction in CUDA

Reduction 1 is the most general. It works on all but the very first GPUs that supported CUDA. Each thread processes one data element and performs an atomic maximum operation on a global variable.

Reduction 2 employs warp primitives that only became available in later hardware generations (with higher compute capabilities). These primitives complicate the code but lower the number of atomic operations substantially.

Reduction 3 uses block-scoped atomics and barriers to further minimize the number of global atomics.

Reduction 4 exploits a warp-based reduction, a recent hardware addition, to lower the number of block-scoped atomics.

Reduction 5 is a variation of Reduction 3 that first computes thread-local results using a persistent-thread approach.

Of the first four versions, Reduction 3 is the fastest, followed by Reduction 4, then Reduction 1, and Reduction 2 is the slowest. This is non-intuitive. After all, minimizing the number of atomics does not yield the best performance, nor does utilizing the latest hardware capabilities. In fact, Reduction 5, a persistent-thread [9] variant of Reduction 3 where each thread processes multiple data elements, outperforms all four shown versions and is about 2.5× faster than Reduction 2 on our test input and GPU. It requires two block-scoped barriers, a block-scoped atomic operation, and a global atomic operation to be correctly synchronized.

These examples highlight the need for better understanding of the performance of synchronization primitives for the following reasons. 1) Even basic parallel primitives like reductions have complex synchronization requirements. 2) Many correct solutions exist, each with its own unique performance implications. 3) It is non-obvious which solution will be the fastest. Hence, even when targeting a single device, the programmer may want to implement and test several alternatives. 4) There is a tradeoff between portability and speed. To obtain high performance on different devices, programmers may have to maintain multiple versions of their code.

III. Approach

Measuring the runtime of synchronization primitives in parallel code requires careful design to avoid accidentally timing irrelevant parts of the testing framework. Specifically, we must avoid timing any overhead created by function calls, loops, or variable initialization. Additionally, we must ensure that the synchronization primitive we are timing is compiled into actual machine code and runs as expected. This usually requires that the tests perform some computation or use the output so that the compiler does not mark the timed primitive as dead code and remove it.

For each synchronization primitive, we define two functions—a *baseline* and a *test* function—each of which times how long it takes to perform many iterations of the primitive. The functions are nearly identical except the test function performs the measured synchronization at least one more time in each iteration. For example, the loop bodies for measuring the execution time of an OpenMP barrier would be a single `#pragma omp barrier` in the baseline function and two such pragmas in the test function. Thus, when we subtract the runtime of the baseline from the test, we are only left with the runtime of the measured synchronization primitive—without any of the testing overhead. This methodology has been inspired by Bialas and Strzelecki's work on benchmarking thread divergence in CUDA [10].

Listing 2 shows the template pseudocode for OpenMP. We run a warmup loop of `N_WARMUP` iterations before the timed section to eliminate any overhead associated with, for example, fetching data from main memory for the first time. We include a barrier after the warmup but before the timed code section to ensure all threads are ready to begin the measurement. Note that the inner loops at lines 10 and 22 are unrolled. If the compiler respects this unrolling, it removes any overhead from initializing and iterating the inner loop, which should improve the timing accuracy. In all our tests, we record the runtime of each thread separately.

Listing 3 shows the template pseudocode for the CUDA codes. The CUDA test framework is nearly identical to the OpenMP counterpart, except it is a CUDA kernel and uses CUDA's `clock64()` function to read the clock-cycle counter at the start and end of the measured code section.

```
void openmp_template(/* ... */) {
  /* define shared variables here */

  #pragma omp parallel num_threads(n_threads) ...
  {
    /* define private variables here */

    for (int i = 0; i < N_WARMUP; i++) {
      #pragma unroll
      for (int j = 0; j < N_UNROLL; j++) {
        /* measured synchronization(s) here */
      }
    }

    #pragma omp barrier
    double local_runtime;
    timeval start, end;
    gettimeofday(&start, NULL);

    for (int i = 0; i < n_iter; i++) {
      #pragma unroll
      for (int j = 0; j < N_UNROLL; j++) {
        /* measured sychronization(s) here */
      }
    }

    gettimeofday(&end, NULL);
    /* save elapsed time for each thread */
  }
}
```

Listing 2: OpenMP Test Template Pseudocode

```
__global__ void cuda_template(/* ... */) {
  /* define shared and private variables here */

  for (int i = 0; i < N_WARMUP; i++) {
    #pragma unroll
    for (int j = 0; j < N_UNROLL; j++) {
      /* measured synchronization(s) here */
    }
  }

  __syncthreads();
  long long start = clock64();

  for (int i = 0; i < n_iter; i++) {
    #pragma unroll
    for (int j = 0; j < N_UNROLL; j++) {
      /* measured synchronization(s) here */
    }
  }

  long long stop = clock64();
  /* save elapsed clock cycles for each thread */
}
```

Listing 3: CUDA Test Template Pseudocode

IV. Experimental Methodology

We use the following parameters in our experiments. All tests vary the number of threads. For the CUDA codes, we also vary the number of thread blocks. For tests that deal with arithmetic or memory operations, we run the code with the `int`, `unsigned long long` (`ull`), `float`, and `double` data types. Since repeated type conversions can incur a large performance cost, we ensure each variable that takes part in an operation is of the tested type. In certain cases, we omit data types that are not natively supported by the tested synchronization primitive. Some of our codes operate across arrays, with each thread reading/writing a private element. For these codes, we vary the stride, which indicates the distance between accessed elements. Lastly, some OpenMP tests vary the thread affinity between “spread” and “close” to explore the impact of thread placement. If the thread affinity is not mentioned for a test, we did not specify the affinity and let the system choose the thread placement.

We use the following procedure to measure the speed of a synchronization primitive. For each combination of parameters, we perform a total of nine runs. Each run attempts to gather a valid measurement seven times. Each attempt runs the baseline and test function, recording the maximum runtime across the running threads. If the maximum runtime of the test function was less than the baseline kernel (suggesting a faulty measurement due to random fluctuations in system performance [11]), we reattempt. After all runs are complete, we determine the median runtime of the seven test runs, the median runtime of the seven baseline runs, and compute the difference. To find the runtime of a single primitive, we divide the result by the number of loop iterations (`n_iter = 1000`) and by the unroll factor (`N_UNROLL = 100`). Section V presents our results in terms of throughput, or operations per second per thread, which is `1 / runtime` for the OpenMP tests and `1 / num_cycles / clock_freq` for the CUDA tests.

We set the number of runs, loop iterations, etc. after experimenting with a wide range of parameter values and finding that the results stabilize above a certain threshold. We chose values that are well within the stable region but not too high to keep the overall experiment runtimes reasonable.

Table I describes the CPUs and GPUs in our three test systems. Additionally, we list the `g++` and `nvcc` versions on each system as well as the compute capability of each GPU. For the GPUs, we list the clock frequency reported by the `cudaDeviceProp` struct included in CUDA’s API [12]. We chose these systems for their diversity in architecture generation, specifications, and manufacturer.

We compiled the OpenMP codes using `g++` with the `-O3` optimization flag and the CUDA codes using `nvcc` with the same flag and the target GPU’s compute capability. We ran all test codes on the three systems.

Note that each measured CUDA primitive is directly supported by the hardware and compiles into a single machine instruction. Hence, we expect other compilers and GPU programming APIs (e.g., OpenCL [13]) to yield similar results. On the CPU, OpenMP primitives like atomics and memory flushes are also compiler independent as they are supported in hardware. Other primitives like barriers and critical sections are implemented in the OpenMP library, so different compilers using the same OpenMP library should produce similar results.

To minimize fluctuations, we ensure we are the only user on the machine during tests. On System 3’s CPU, across the nine runs, the standard deviation of a single primitive’s runtime is typically about 7.8 nanoseconds, which is negligible. On the GPU, there are no background processes or OS, and we directly read the cycle counter. Thus, many of the GPU tests yield the exact same runtime for all nine runs.

TABLE I: System Specifications

(a) System 1

Intel Xeon E5-2687 v3	
Base Clock Frequency	3.10 GHz
Sockets	2
Cores Per Socket	10
Threads Per Core	2
NUMA nodes	2
Main memory	128 GB
NVIDIA GeForce RTX 2070 SUPER	
Compute Capability	7.5
Clock Frequency	1.80 GHz
SMs	40
Max Threads per SM	1024
CUDA Cores per SM	64
Memory	8 GB
`g++` Version	12.3.1
`nvcc` Version	12.0
GPU Driver	550.67

(b) System 2

Intel Xeon Gold 6226R	
Base Clock Frequency	2.80 GHz
Sockets	2
Cores Per Socket	16
Threads Per Core	2
NUMA nodes	2
Main memory	64 GB
NVIDIA A100 40GB	
Compute Capability	8.0
Clock Frequency	1.41 GHz
SMs	108
Max Threads per SM	2048
CUDA Cores per SM	64
Memory	40 GB
`g++` Version	12.3.1
`nvcc` Version	12.0
GPU Driver	535.113.01

(c) System 3

AMD Ryzen Threadripper 2950X	
Base Clock Frequency	3.50 GHz
Sockets	1
Cores Per Socket	16
Threads Per Core	2
NUMA nodes	2
Main memory	48 GB
NVIDIA GeForce RTX 4090	
Compute Capability	8.9
Clock Frequency	2.625 GHz
SMs	128
Max Threads per SM	1536
CUDA Cores per SM	128
Memory	24 GB
`g++` Version	12.2.1
`nvcc` Version	12.0
GPU Driver	525.85.05

V. Results

This section presents and analyzes our results. Unless otherwise noted, all figures display measurements from System 3, as it is the system with the latest CPU and GPU. We only provide results for the other systems when their behavior differs significantly from System 3. The result charts show the throughput (y-axis) for varying thread counts (x-axis). Throughput is a higher-is-better metric. Note that some aspects of OpenMP, CUDA, and the CPU and GPU architectures are unknown to us. Thus, this section focuses on providing recommendations for developers.

A. OpenMP Results

On each system, we ran the OpenMP codes for thread counts from 2 to the maximum number of supported hyperthreads. The dashed vertical line indicates the point where the thread count matches the core count, that is, hyperthreading is used to the right of this point. We omit a thread count of 1 since synchronization serves no purpose in a serial execution.

1) Barrier: Fig. 1 shows the throughput of the OpenMP barrier for various thread counts. The thread affinity was set to "spread" for this figure but did not make a significant impact on this test.

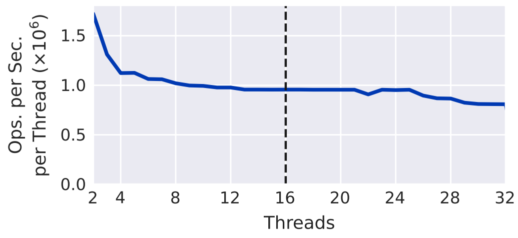


Fig. 1: Throughput of OpenMP Barrier

The throughput per thread initially decreases as more threads participate. This is expected since the threads spend, on average, more time waiting for the other threads to reach the barrier. Surprisingly, beyond about 8 threads, the per-thread throughput remains largely stable. Since OpenMP barriers are implemented in a library, we cannot say what causes this behavior. The throughput also does not decrease much when hyperthreading is used. Since every thread participates in the barrier, this implies that the scheduling of hyperthreads is quite balanced on the tested CPUs. The following OpenMP tests further suggest this to be the case.

2) Atomics: We tested several of OpenMP's atomics to compare and contrast their performance. Fig. 2 shows the throughput of each thread atomically adding a value to a shared variable using `#pragma omp atomic update`.

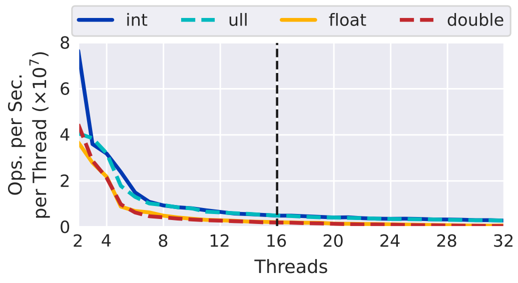
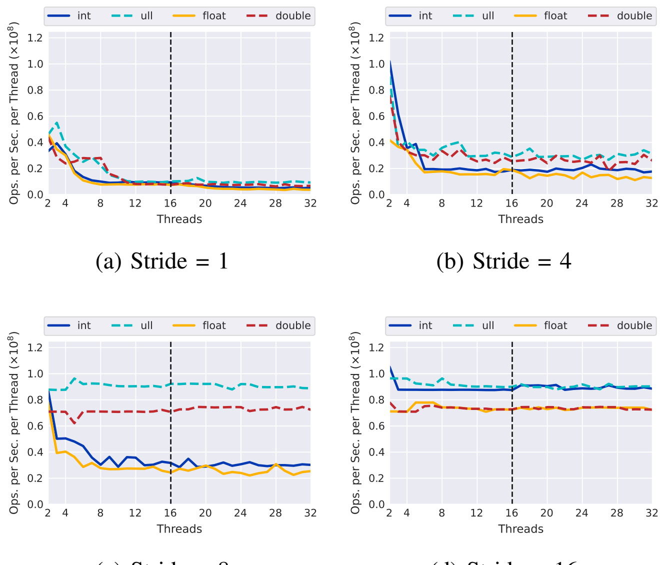


Fig. 2: Throughput of OpenMP atomic update on a single shared variable

Overall, we see the same performance trend as for the barrier (see Fig. 1), indicating that the barrier implementation is likely based on atomic operations on shared variables. For the atomic increment, the integer types (`int` and `ull`) are generally faster than the floating-point types (`float` and `double`). The word size (32 or 64 bits) does not affect the performance on our 64-bit CPUs.

We performed a similar experiment with OpenMP's atomic capture. The resulting behavior and throughputs are nearly identical to the atomic update and not shown.

Fig. 3 displays the throughput of each thread atomically adding a value to its own private element in a shared array. Of course, atomically protecting a private element is unnecessary as there is no risk of a data race. However, we perform this experiment to emulate a scenario where the possibility of contention exists but does not occur. We repeated this test with several strides (i.e., distances in elements between accessed array elements) and show results for a stride of 1, 4, 8, and 16. Since each thread is accessing data from an array, the stride dictates how much false sharing [14] occurs when threads running on separate cores (with separate L1 data caches) update elements that, while disjoint, belong to the same cache line. We also tested thread affinities of "spread" and "close" but saw no notable difference. To better highlight the progression between the strides, the 4 y-axes are zero-based and use the same scale.



(a) Stride = 1 (b) Stride = 4

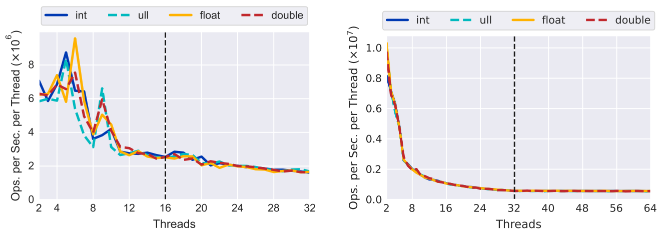
(c) Stride = 8 (d) Stride = 16

Fig. 3: Throughput of OpenMP atomic update on private elements in a shared array for different strides

Here, we see not only the effect of the data type but also symptoms of the coherent caches. Note that the L1 cache line size for System 3's CPU is 64 bytes. Fig. 3a with a stride of 1 (adjacent elements) shows the case with maximum false sharing, where the 4-byte types (`int` and `float`) generally perform slightly worse than the 8-byte types because the 4-byte types have twice as many words in a cache line. As we increase the stride, false sharing becomes less of a factor. For example, Fig. 3b shows that the throughput of all four data types is higher with a stride of 4. At a stride of 8, something interesting happens as illustrated in Fig. 3c: the throughput of the two 32-bit types further increases a little, but the throughput of the two 64-bit types shoots up drastically as there is now enough padding between accessed elements for each thread's element to reside in a separate cache line, meaning there is no more false sharing. Fig. 3d shows that the same happens to the 32-bit types when going to a stride of 16. In this last stride configuration, all types no longer suffer from false sharing, and the bottleneck is instead the atomic arithmetic on the data type (like in Fig. 2), which is why the integer types are faster than the floating-point types, regardless of word size. At this point, the throughput is largely constant since the operations are embarrassingly parallel (they access the private L1 caches) and there is no coherence traffic.

Note that, for any stride, hyperthreads running on the same core cannot suffer from false sharing as they access the same cache. However, as made evident by many of our OpenMP tests, hyperthreading yields more variability in thread timing, and, therefore, more chances of recording jitter in the measurements.

Fig. 4 shows the throughput of OpenMP's atomic write on two systems. The baseline function performs an atomic write to one shared memory location, and the test function performs an atomic write to two shared memory locations on separate cache lines. We present results from two systems due to the notable jitter in System 3's results (Fig. 4a), which we attribute to architectural qualities of the AMD chip. System 1's results are similar to System 2's.



(a) System 3 (b) System 2

Fig. 4: Throughput of OpenMP atomic write on 2 systems

In spite of the jitter, the exponentially decreasing trend we have seen before is again evident. The results echo those of the atomic update, except the integer types (`int` and `ull`) have no performance advantage since no arithmetic is involved. Moreover, the size of the data type has no observable effect on the throughput of the write. This is because all the tested CPUs have 64-bit architectures, meaning they can perform a store/load instruction on up to 64 bits (8 bytes) of data in a single transaction.

We also performed experiments to measure OpenMP's atomic read, where the baseline function performs a non-atomic read and the test function performs the same read atomically. The intention was to measure the overhead of performing the read atomically. We found that the difference between the runtimes of the baseline and test functions were extremely small and within the timer's accuracy. Hence, we conclude that the throughputs are actually the same, and there is no performance penalty in using an atomic read.

3) Critical Section: It is well-accepted that critical sections are slower than atomics when performing a logically-equivalent operation (since the critical section's locking mechanism is implemented using atomics). Yet, it is important to analyze the behavior of critical sections to guide their usage in situations where no alternative implementation is available. Fig. 5 shows the throughput of a critical section used to adding a value to a single shared variable. The thread affinity was set to “spread”, but varying the affinity did not make a noticeable impact.

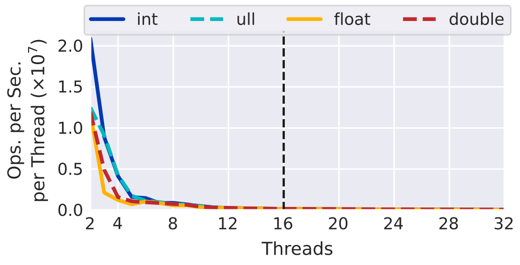
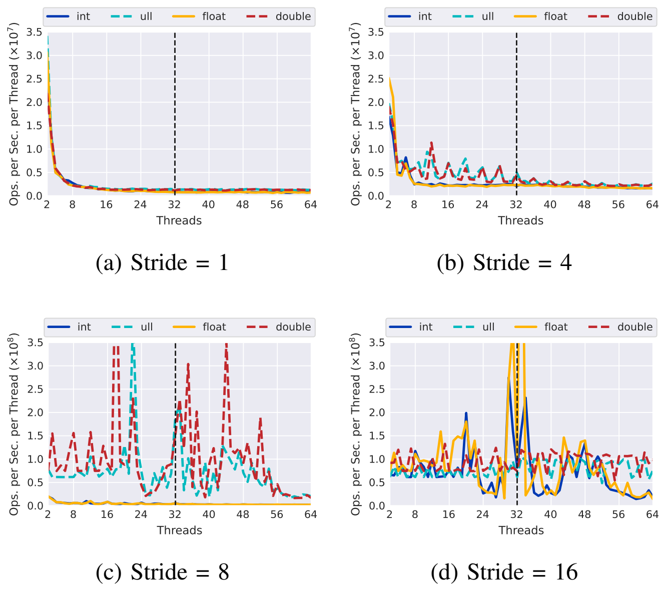


Fig. 5: Throughput of an addition on a single variable protected by an OpenMP critical section

The trend for this figure is similar to the atomic counterpart in Fig. 2, but the throughput drops more quickly and is lower. We recommend developers only use critical sections when no alternative exists.

4) Memory Flush: To measure a flush, we declare two arrays with a size equal to the number of active threads, padded by the test’s specified array stride. In the baseline function, each thread increments its private element of each array by a value (see Listing 2). The test function does the same except with a flush in between the two operations.

Fig. 6 shows the throughput of a flush with the “close” thread affinity at several strides for System 2. The “spread” thread affinity yielded similar results. Note that the y-axis scale is $\times 10^7$ in Fig. 6a and 6b and $\times 10^8$ in Fig. 6c and 6d. We show System 2’s results because they are less noisy, but the trends are the same on the other systems.



(a) Stride = 1

(b) Stride = 4

(c) Stride = 8

(d) Stride = 16

Fig. 6: Throughput of OpenMP flush for different strides

As we increase the stride, we decrease the impact of false sharing. At a stride of 1, the throughput of the flush exponentially decreases and plateaus at around half the number of physical cores (16 for System 2). At a stride of 4, all types start to exhibit oscillations—moreso for the 64-bit types. At a stride of 8, where the 64-bit types are no longer affected by false sharing, their throughput increases substantially. However, so does the oscillation’s amplitude. At a stride of 16, the padding is enough for all accessed elements to belong to their own cache line, and the 32-bit types begin to exhibit the same behavior as the 64-bit types. It seems that, in general, when there is little to no false sharing, the impact of enforcing the order of reads and writes with a flush is minimal in our test code. In summary, an OpenMP flush has little per-thread performance impact in situations where it is not really necessary for enforcing consistency. Otherwise, it exhibits similar throughput behavior as an atomic update or barrier.

5) Recommendations: Our OpenMP results lead to the following recommendations. 1) Barriers are not much cheaper when only a few threads are used, thus, they are not a concern with larger thread counts. 2) Atomic updates or writes by multiple threads to the same memory location should be avoided, as they are quite slow. 3) Atomic operations by multiple threads to different locations are much faster if the different locations do not reside in the same cache line. Thus, we recommend that programmers avoid false sharing by assigning work to threads that leads to mostly non-overlapping accesses between threads if possible. 4) Atomic reads appear to not incur any extra latency and can be used wherever prudent. 5) Critical sections should be avoided unless there is no other option. 6) Flushes do not have a major performance impact and can be used as needed. 7) In general, using hyperthreads is fine as they do not significantly slow down synchronizations.

B. CUDA Results

On each GPU, we ran the CUDA codes with block counts of 1, 2, half the number of SMs, the number of SMs, and twice the number of SMs. Each block count was tested with thread counts of powers of 2 through 1024. Note that the x-axis of the result charts uses a logarithmic scale.

1) Syncs: Fig. 7 shows the throughput of `__syncthreads()`, which is a block-wide barrier.

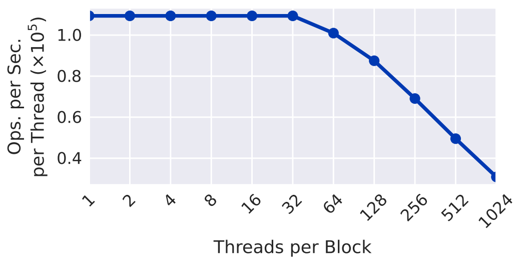
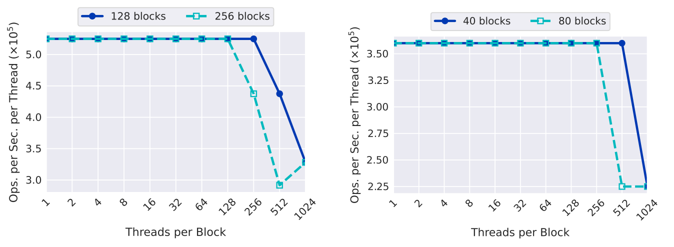


Fig. 7: `__syncthreads()` throughput at any block count

The throughput is constant until we surpass the warp size of 32 threads. The constant throughput up to the warp size is expected because, at smaller sizes, the GPU still runs a whole warp with some of the lanes disabled. Beyond the warp size, throughput drops because warps are required to wait for each other. This mirrors the behavior of increasing active threads in the OpenMP barrier (see Fig. 1), except on the GPU more active *warps* consistently result in longer average wait times. Note that, unlike in OpenMP, barriers are implemented in hardware on GPUs. The results are identical for all block counts since `__syncthreads()` is a block-wide barrier that has no dependencies across blocks.

Fig. 8 shows the throughput of a `__syncwarp()`, a warp-wide barrier [15], for Systems 1 and 3 at full and double block configurations. We compare these GPUs because of their difference in behavior, which stems from the differing number of maximum threads per SM. The behavior of System 2 is the same as System 3 and not shown.



(a) System 3 (RTX 4090) (b) System 1 (RTX 2070 SUPER)

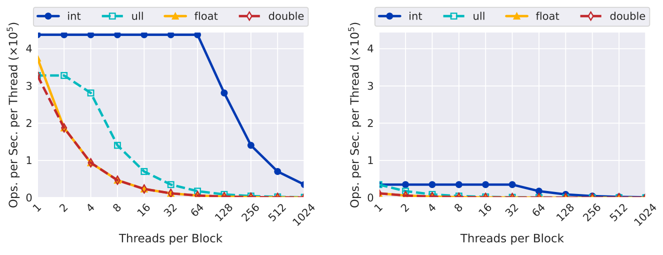
Fig. 8: Throughput of `__syncwarp()` on two systems

On both systems at both block counts, the throughput is constant up to a point. For both systems, the double block configuration drops in performance one step earlier than the full block configuration. This can be attributed to the fact that, except at 1024 threads, the double block experiments allocates 2 blocks to each SM. So, while the per-thread throughput decreases, we are running twice the number of blocks at a time and, therefore, twice as many threads per SM. Hence, the `__syncwarp()` throughput depends on the number of warps running on an SM rather than the number of warps per thread block.

Next, let us focus on the full-block configuration. With 1024 threads per block, both systems must run one block to completion and then the other. It seems like the RTX 4090 can handle up to 256 threads per SM, and the RTX 2070 SUPER can handle up to 512 threads per SM at full speed. Beyond those thread counts, the throughput drops somewhat (the y-axis does not start at zero).

2) Atomics: We tested several of CUDA's atomics to analyze their behavior. Fig. 9 shows the throughput of each thread atomically adding a value to a single shared variable using `atomicAdd()` at two notable block counts: 2 and 64 (half the number of SMs on the RTX 4090). The 1-block configuration behaves like the 2-block configuration, and the full and double configurations behave like the half configuration, though at lower absolute throughputs.

For the `int`, the throughput is constant until we surpass the size of a warp. Interestingly, the 2-block configuration remains at a constant throughput up to 64 threads (2 warps).



(a) 2 blocks (b) 64 blocks

Fig. 9: Throughput of `atomicAdd()` on 1 shared variable

It seems that CUDA is able to automatically aggregate the individual atomics into a warp-aggregated atomic [16], which is implemented as a reduction-and-broadcast within the warp. Our examination of the `nvcc`-generated PTX [17] assembly found no evidence of this optimization, which suggests that this is a just-in-time compiler optimization performed in the driver when the code is loaded onto the GPU.

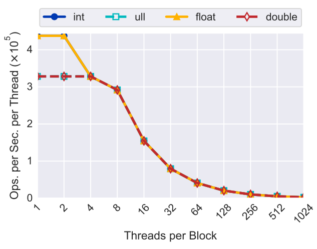
In either configuration, there is a gap in performance between the `int` and the other three data types. Old CUDA documentation mentions *atomic units*—GPU hardware elements that handle atomic operations [18]. Unfortunately, there is a lack of detailed description of these units in recent documents. Nevertheless, this performance gap implies that there are more integer than floating-point atomic units or that the integer atomic unit's add operation is much faster. Whereas `ull` is faster than the floating-point types, it is slower than `int` presumably because the tested GPUs have 32-bit architectures (unlike the tested CPU architectures).

Fig. 10 summarizes the behavior of `atomicAdd()` on private elements in a shared array. We show results for a stride of 1 and 32 and for block counts of 1 and 128.

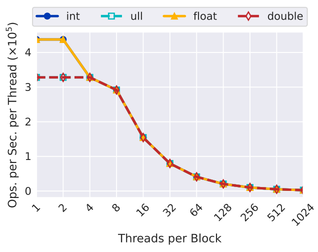
Since each thread writes to a different element, we do not see the benefit of the warp-aggregated atomics like with the previous single-variable test. In general, as we increase the block count, the throughput per thread decreases. Interestingly, for the block count of 1, the throughput trend is the same regardless of stride. In contrast, stride fundamentally changes the throughput trend for all types for the higher block counts. At 128 blocks, the throughput is lower than at 1 block since more SMs are sharing the L2 cache bandwidth. It seems that the primary reason for the downward trend is the fixed number of atomics that the hardware can perform per time unit.

Fig. 11 shows our results for `atomicCAS()` on a single shared variable. Since this function contains an implied conditional operation (the swap depends on whether the comparison passes), we created two test versions: one that always passes the comparison and another that always fails. We saw no difference in performance and present the version that always passes. Note that `atomicCAS()` does not natively support floating-point types.

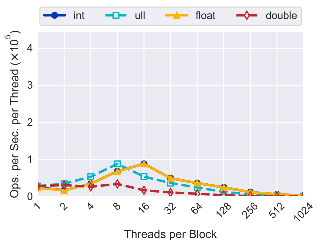
Interestingly, the 1-block configuration has a constant throughput up to 4 threads. Similarly, while not shown, the 2-block configuration has a constant throughput up to 2 threads.



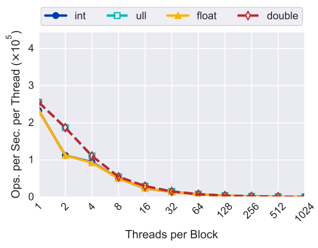
(a) 1 Block, Stride = 1



(b) 1 Block, Stride = 32

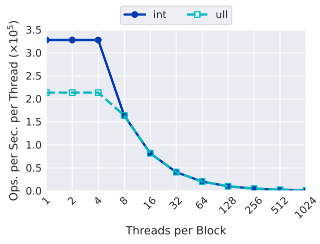


(c) 128 Blocks, Stride = 1

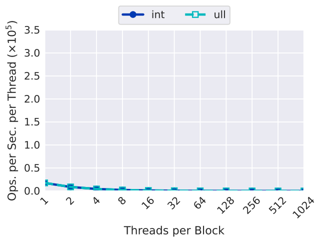


(d) 128 Blocks, Stride = 32

Fig. 10: Throughput of `atomicAdd()` on private elements in a shared array for different block counts and strides

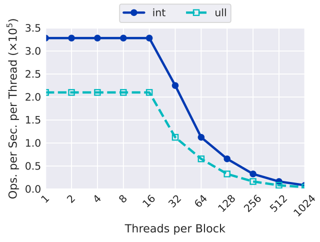


(a) 1 Block

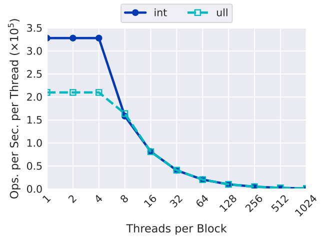


(b) 128 Blocks

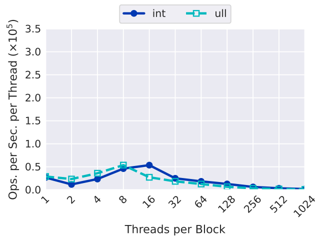
Fig. 11: Throughput of `atomicCAS()` on 1 shared variable



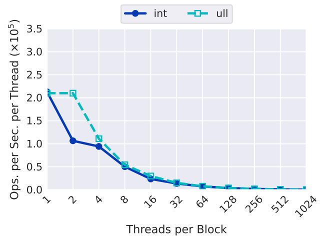
(a) 1 Block, Stride = 1



(b) 1 Block, Stride = 32

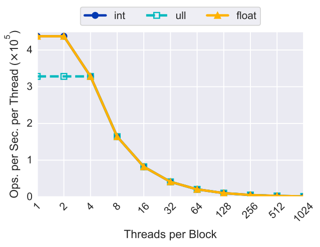


(c) 128 Blocks, Stride = 1

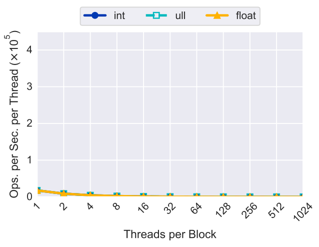


(d) 128 Blocks, Stride = 32

Fig. 12: Throughput of `atomicCAS()` on private elements in a shared array for different block counts and strides



(a) 1 Block



(b) 128 Blocks

Fig. 13: Throughput of `atomicExch()` at 2 block counts

An `atomicCAS()` on its own cannot benefit from the same warp-aggregation optimization as `atomicAdd()`, since the result of the comparison may change the outcome for other threads within the same warp. Thus, the constant throughput drops earlier than with the `atomicAdd()` but otherwise follows the same trend.

Fig. 12 summarizes the behavior of each thread performing an `atomicCAS()` on its private element of a shared array. We show results for a stride of 1 and 32, for block counts of 1 and 128.

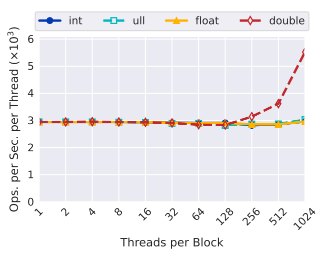
These results resemble the trends of the `atomicAdd()` (see Fig. 10), albeit with a different drop-off point for the block count of 1 (Fig. 12a and 12b). Again, it seems that there is a fixed maximum number of atomic CAS operations that the hardware can handle concurrently.

Fig. 13 shows the results for `atomicExch()`. In the baseline function, each thread repeatedly swaps the current value in a single shared address with the value of its global thread ID. The test function does so twice as many times.

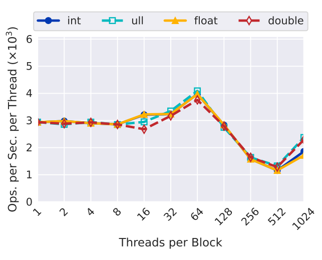
These results are similar to the `atomicCAS()` (see Fig. 11). As there is no arithmetic, the per-thread performance is simply memory-bound. Since each thread is reading from and writing to the same location, more active threads means each thread has to wait longer for the other threads to finish their atomic operation.

3) Thread Fences: Fig. 14 summarizes the behavior of a device-wide `__threadfence()`. This test has the same setup as the OpenMP flush; each thread updates its private element in two distinct arrays, with the test function's version introducing a `__threadfence()` in between the updates. We show the results for block counts of 1 and 128 and strides of 1 and 32.

Interestingly, the overall pattern has little similarity to the OpenMP flush counterpart, presumably due to memory hierarchy differences between CPU and GPU. (Our CPUs have coherent private L1 and L2 caches and a shared L3 cache whereas our GPUs have incoherent L1 caches that are not used for shared data and a shared L2 cache.) The throughput of the CUDA fence is fairly constant regardless of thread count, block count, or stride. It seems that the latency stems primarily



(a) 1 Block, Stride = 1



(b) 128 Blocks, Stride = 32

Fig. 14: Throughput of __threadfence() for different block counts and strides

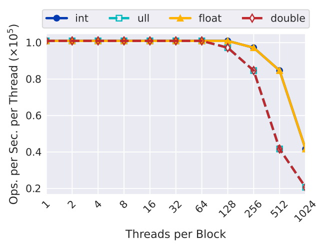
from waiting for the load/store buffers to be drained and from preventing the memory controller from reordering accesses to improve performance.

The __threadfence_system() test is close in behavior to its device-wide counterpart, so we do not show the results. However, the behavior is more erratic since it involves communication with the CPU across the PCIe bus.

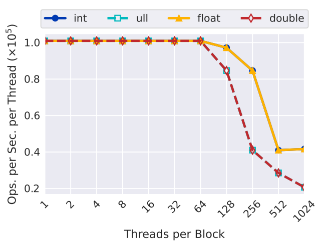
We also tested __threadfence_block() using the same test setup as __threadfence(). The throughput for thread counts within the warp size is constant. However, at thread counts above the warp size and at strides above 2, we observe many runtimes at or near zero. This suggests that our test function's measured section is taking nearly the same amount of time as the baseline function. We attribute this to the fact that for this code (without fences), the memory accesses within a thread block are not reordered relative to the code with the fence.

4) Warp-Level Functions: We studied several types of warp-level functions, all of which implicitly synchronize the participating threads. For functions that require a mask to specify the participating threads, we pass a mask that includes all threads in the warp.

Fig. 15 shows the throughput of a __shfl_sync() at two block configurations (full and double). We also tested the up, down, and xor variants but observed no performance difference, indicating that the implementations are identical aside from the data movement pattern.



(a) 128 blocks



(b) 256 blocks

Fig. 15: Throughput of CUDA __shfl_sync()

The behavior of the __shfl_sync() is the same as a __syncwarp() because the former implies the latter. Since the __shfl_sync() involves a data movement, the size of the data type matters. The GPU hardware only supports 32-bit shuffles, so the compiler generates two such instructions for 64-bit words. Thus, we see that the 64-bit types drop at half the thread count compared to the 32-bit types.

We also measured the three warp voting functions __ballot_sync(), __all_sync() and __any_sync(). We were unable to reliably record __ballot_sync(), likely due to some optimization preventing it from being properly generated/executed. In a real situation, we expect it to behave similarly to the other voting functions, which behave identically to __syncwarp() (see Fig. 8), but with a slightly lower absolute throughput.

5) Recommendations: Our CUDA results lead to the following recommendations. 1) __syncthread() performance decreases with increasing warp counts, so smaller block sizes might help in a barrier-heavy code. 2) __syncwarp() throughput is largely constant and can be used without consideration for block or thread count. 3) int atomic adds and CAS are preferred over other data types. 4) Running multiple atomic adds/CASs on the same memory location slows performance, so overlap should be avoided. 5) Running too many simultaneous atomics should be avoided, as we observed limits on the number of atomics per unit of time. 6) Thread fences incur largely constant overhead and can be used as necessary without regard for thread count. 7) Warp shuffles are fast but decrease throughput when the SM is nearly fully loaded—moreso for 8-byte types. Still, they should be used when possible to avoid memory traffic. 8) In general, codes should always use full warps to maximize performance except for atomic operations, where partial warps can give higher performance. That is, "turning off" threads in a warp that do not need to execute an atomic with an if-statement may yield performance benefits.

VI. Related Work

Several prior works benchmark or otherwise evaluate the behavior of CPU [19] [20] or GPU [21] [22] [23] synchronization primitives in a variety of contexts. We highlight the closest to our work in the following.

Brovenetsky et al. [24] created CLOMP, a benchmark that characterizes the overhead introduced by OpenMP synchronizations. CLOMP's main goal is to identify the amount of work required to compensate for the introduced overhead. Our work differs in that we observe behaviors of synchronization primitives at per-thread granularity, and our experiments are on generalized access patterns rather than actual codes. Furthermore, we test different sets of primitives and include CUDA.

Bialas and Strzelecki [10] created a micro-benchmark to determine the cost of thread divergence in CUDA on several architectures. They found that the cost of a diverging branch is essentially constant, depending on the architecture. Our timing approach is heavily inspired by their work, albeit for measuring the behavior of synchronization primitives.

O'Neil and Burtscher [25] characterize the performance of several irregular GPU codes, identifying the particular bottlenecks (e.g., thread divergence, stalls for atomics) that occur and how much they affect the application runtime. Additionally, they perform simulations to understand the impact of cache sizes, cache and memory latencies, etc. Our work measures the throughput of synchronization primitives on physical hardware.

Lee et al. [26] performed a detailed study and analysis to debunk claims that GPUs outperform multi-core CPUs by a factor of 10 to 1,000. They found that the average performance gap was only 2.5×. Their work highlights that inherent differences between CPU and GPU architectures make them appropriate for different types of problems. Whereas our primary goal is not a direct comparison of GPU and CPU performance, our results do illustrate how effective the various synchronization primitives are on the two types of devices.

VII. Summary and Conclusions

This work presents a test framework to measure the throughput of individual OpenMP and CUDA synchronization primitives under a variety of parameters. The code and results are open-source and publicly available [6]. Through our experiments, we arrive at a number of recommendations for parallel-program developers. For example, we suggest that OpenMP programmers be mindful of false sharing and avoid the utilization of critical sections. Furthermore, we observe that the use of hyperthreading has little to no effect on per-thread throughput. Section V-A5 provides further recommendations. For CUDA, we recommend that programmers stay wary of overlapping memory accesses as well as the number of simultaneous atomic operations. We also generally recommend the use of whole warps except for certain atomics. Section V-B5 lists additional recommendations. We hope this information will help software engineers write more efficient parallel CPU and GPU codes and proves useful to hardware manufacturers and parallel-programming-library developers.

References

[1] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998. [Online]. Available: https://ieeexplore.ieee.org/document/658762

[2] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974. [Online]. Available: https://ieeexplore.ieee.org/document/1050511

[3] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4785534

[4] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 9, pp. 1277–1284, Sep. 1996. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/535411

[5] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*. Atlantic City, New Jersey: ACM Press, 1967, p. 483. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1465482.1465560

[6] B. A. Burtchell and M. Burtscher, "SyncPerformance," 2024. [Online]. Available: https://github.com/burtscher/SyncPerformance

[7] OpenMP Architecture Review Board, "OpenMP API Specification: Version 5.1 November 2020," 2020. [Online]. Available: https://www.openmp.org/spec-html/5.1/openmp.html

[8] NVIDIA, "CUDA C Programming Guide Version 12.5," 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[9] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of Persistent Threads style GPU programming for GPGPU workloads," in *2012 Innovative Parallel Computing (InPar)*. San Jose, CA, USA: IEEE, May 2012, pp. 1–14. [Online]. Available: http://ieeexplore.ieee.org/document/6339596

[10] P. Bialas and A. Strzelecki, "Benchmarking the cost of thread divergence in CUDA," Apr. 2015. [Online]. Available: http://arxiv.org/abs/1504.01650

[11] E. Vicente and R. Matias Jr., "Exploratory Study on the Linux OS Jitter," in *2012 Brazilian Symposium on Computing System Engineering*. Natal, Brazil: IEEE, Nov. 2012, pp. 19–24. [Online]. Available: http://ieeexplore.ieee.org/document/6473626/

[12] NVIDIA, "CUDA Runtime API Version 12.5," 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[13] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010. [Online]. Available: https://doi.org/10.1109/MCSE.2010.69

[14] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, 4th ed., ser. The Morgan Kaufmann series in computer architecture and design. Waltham, MA: Morgan Kaufmann, 2012.

[15] NVIDIA, "NVIDIA TESLA V100 GPU ARCHITECTURE," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[16] A. Adinets, "CUDA pro tip: Optimized filtering with warp-aggregated atomics," 2017. [Online]. Available: https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

[17] NVIDIA, "Parallel Thread Execution ISA Version 8.5," 2024. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[18] NVIDIA, "Whitepaper: NVIDIA's next generation CUDA compute architecture: Fermi," 2009. [Online]. Available: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[19] J. Torrellas, A. Gupta, and J. Hennessy, "Characterizing the caching and synchronization performance of a multiprocessor operating system," *ACM SIGPLAN Notices*, vol. 27, no. 9, pp. 162–174, Sep. 1992. [Online]. Available: https://dl.acm.org/doi/10.1145/143371.143506

[20] P. Wang, W. Gao, J. Fang, C. Huang, and Z. Wang, "Characterizing OpenMP Synchronization Implementations on ARMv8 Multi-Cores," in *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, Dec. 2021, pp. 669–676. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9780950

[21] B. Gurumurthy, D. Broneske, M. Schäler, T. Pionteck, and G. Saake, "Novel insights on atomic synchronization for sort-based group-by on GPUs," *Distributed and Parallel Databases*, vol. 41, no. 3, pp. 387–409, Sep. 2023. [Online]. Available: https://doi.org/10.1007/s10619-023-07424-2

[22] M. Elteir, H. Lin, and W.-C. Feng, "Performance Characterization and Optimization of Atomic Operations on AMD GPUs," in *2011 IEEE International Conference on Cluster Computing*. Austin, TX, USA: IEEE, Sep. 2011, pp. 234–243. [Online]. Available: http://ieeexplore.ieee.org/document/6061141/

[23] L. Zhang, M. Wahib, H. Zhang, and S. Matsuoka, "A Study of Single and Multi-device Synchronization Methods in Nvidia GPUs," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 483–493. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9139854

[24] G. Bronevetsky, J. Gyllenhaal, and B. R. de Supinski, "CLOMP: Accurately Characterizing OpenMP Application Overheads," *International Journal of Parallel Programming*, vol. 37, no. 3, pp. 250–265, Jun. 2009. [Online]. Available: https://doi.org/10.1007/s10766-009-0096-7

[25] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. Raleigh, NC, USA: IEEE, Oct. 2014, pp. 130–139. [Online]. Available: http://ieeexplore.ieee.org/document/6983052

[26] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund,

R. Singhal, and P. Dubey, “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*, ser. ISCA ’10. New York, NY, USA: Association for Computing Machinery, Jun. 2010, pp. 451–460. [Online]. Available: https://dl.acm.org/doi/10.1145/1815961.1816021

Appendix

A. Abstract

This artifact description provides information about the workflow required to execute the testing framework for measuring the execution time of synchronization primitives on a supported system. We describe how the software can be obtained as well as the necessary steps to install and set up scripts to automate running the synchronization-primitive measurements across all parameters. Furthermore, the artifact contains the raw results and figures obtained from the three tested systems specified in the paper. Some of them were not presented in the paper for the sake of avoiding redundancy. Here they are provided in full.

B. Artifact check-list (meta-information)

- **Program:** SyncPerformance v1.0.0
- **Compilation:** GNU C++ (`g++`) and NVIDIA CUDA (`nvcc`) compiler
- **Hardware:** Our results are from the following systems, but the codes can be run on any supported hardware and should yield similar trends.
  - System 1: Intel Xeon E5-2687 v3, NVIDIA GeForce RTX 2070 SUPER
  - System 2: Intel Xeon Gold 6226R, NVIDIA A100 40GB
  - System 3: AMD Ryzen Threadripper 2950X, NVIDIA GeForce RTX 4090
- **Output:** For each synchronization primitive test code: raw runtime results and a figure summarizing the behavior in terms of throughput.
- **Time needed to complete experiments:** Approximately 72 hours for all codes on a single system. This can be more or less—primarily depending on the number of cores in the CPU and the number of SMs in the GPU, but also other factors.
- **Publicly available:** The code and results obtained from our test systems are publicly available.
- **Archived:** https://doi.org/10.5281/zenodo.13227900

C. Description

1) How to access: The software can be obtained from GitHub: https://github.com/burtscher/SyncPerformance.

```
$ git clone \
  https://github.com/burtscher/SyncPerformance.git
```

To set the repository to the paper version, run:

```
$ git checkout v1.0.0
```

2) Hardware dependencies: The experiments can be executed on any system that meets the following requirements:

- OpenMP tests require a multicore CPU
- CUDA tests require a CUDA-enabled NVIDIA GPU with a minimum compute capability of 7.5

3) Software dependencies: All code is intended to run in a Linux environment. The requirements for compiling and executing individual test codes on a CPU and GPU are:

- `g++` version 12.2 or higher
- `nvcc` version 12.0 or higher

We provide Python scripts to automate compiling and running each test code across parameters as well as generating subsequent tables and figures. The requirements to run these scripts are:

- Python 3.11
- Numpy (https://numpy.org/) version 1.25 or later
- Matplotlib (https://matplotlib.org/) version 3.8 or later
- Seaborn (https://seaborn.pydata.org/) version 0.13.2 or later

D. Installation

Install the necessary Python packages. It is recommended to install these with `pip`, i.e.:

```
$ pip3 install numpy matplotlib seaborn
```

1) CUDA setup: If testing CUDA codes, it may be necessary to specify the compute capability (see https://developer.nvidia.com/cuda-gpus) when compiling codes to match a system's GPU, especially if the system has multiple GPUs. To do so, create `config.py` in the root of the repository. Inside, specify the desired `nvcc` "`-arch=`" argument. For example, for an NVIDIA GeForce RTX 4090 GPU, which has a compute capability of 8.9, `config.py` should simply contain:

```
nvcc_arch = "sm_89"
```

A working example is provided in `./config.py.example` and can be copied. If the compute capability is not a concern (e.g., if only running OpenMP codes), this step can be safely ignored, and the code will default to `nvcc_arch = "native"`.

If the system has multiple GPUs, ensure the GPU of interest is selected before running any scripts, e.g.:

```
$ export CUDA_VISIBLE_DEVICES=1
```

E. Experiment workflow

Before launching any experiments, care should be taken to minimize running background processes. Otherwise, the yielded measurements could be inaccurate.

1) Whole experiment (all codes): To run all OpenMP and CUDA codes with a single command, run:

```
$ ./launch.py all
```

The script will list every code that will be compiled and run and prompt the user for confirmation to proceed.

2) Partial experiment (OpenMP or CUDA only): To run only the OpenMP or CUDA set of codes, specify which as an argument. For example, to only run the OpenMP codes:

```
$ ./launch.py openmp
```

3) Individual codes: To run individual codes across all parameters, list the paths to the desired codes. For example, to launch the code that measures an OpenMP atomic update on a single shared variable, run:

```
$ ./launch.py ./codes/omp/omp_atomicadd_scalar.cpp
```

F. Evaluation and expected results

Each test's results will be output to a corresponding directory in `./results/<hostname>/`. If the test uses different strides, there will be subdirectories for each stride. Inside is a log of the raw program output (`log.txt`), a binary file containing the runtime results in Python data structures (`runtimes.bin`), a CSV file listing the runtime of the single primitive across all parameters (`runtimes.csv`), and a figure summarizing the behavior of the primitive across parameters in terms of throughput (`<testname>.pdf`). For the CUDA tests, one figure will be generated for each block configuration.

The results may vary from those presented in the paper if run on a system with different specifications than ours. Nonetheless, we expect the same general trends to be evident on a majority of similar hardware.

G. Experiment customization

Global test parameters (e.g., `N_UNROLL`, `N_RUNS`) can be found and edited in `./include/config.h`. Other per-code parameters (e.g., `n_iter`, `thread_range`) can be found in `./run_tests.py` for both OpenMP and CUDA in the functions `execute_omp()` and `execute_cuda()`, respectively.

H. Notes

We provide the raw results and figures obtained from the three tested systems specified in the paper in `./results/system*/`.

I. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html