A Specialized Concurrent Queue for Scheduling Irregular Workloads on GPUs

David Troendle
The University of Mississippi
Department of Computer and Information Science
University, Mississippi, USA
david@cs.olemiss.edu

Tuan Ta
Cornell University
School of Electrical and Computer Engineering
Ithaca, New York, USA
qtt2@cornell.edu

Byunghyun Jang
The University of Mississippi
Department of Computer and Information Science
University, Mississippi, USA
bjang@cs.olemiss.edu

ABSTRACT

The persistent thread model offers a viable solution for accelerating data-irregular workloads on Graphic Processing Units (GPUs). However, as the number of active threads increases, contention and retries on shared resources limit the efficiency of task scheduling among the persistent threads. To address this, we propose a highly scalable, non-blocking concurrent queue suitable for use as a GPU persistent thread task scheduler. The proposed concurrent queue has two novel properties: 1) The supporting enqueue/dequeue queue operations never suffer from retry overhead because the atomic operation does not fail and the queue empty exception has been refactored; and 2) The queue operates on an arbitrary number of queue entries for the same cost as a single entry. A proxy thread in each thread group performs all atomic operations on behalf of all threads in the group. These two novel properties substantially reduce thread contention caused by the GPU's lock-step Single Instruction Multiple Threads (SIMT) execution model.

To demonstrate the performance and scalability of the proposed queue, we implemented a top-down Breadth First Search (BFS) based on the persistent thread model using 1) the proposed concurrent queue, and 2) two traditional concurrent queues; and analyzed its performance and scalability characteristics under different input graph datasets and hardware configurations. Our experiments show that the BFS implementation based on our proposed queue outperforms not only ones based on traditional queues but also the state-of-the-art BFS implementations found in the literature by a minimum of 1.26× and maximum of 36.23×. We also observed the scalability of our proposed queue is within 10% of the ideal linear speedup for up to the maximum number of threads supported by high-end discrete GPUs (14K threads in our experiment).

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337837

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; *Concurrent algorithms*; • **Software and its engineering** → *Concurrent programming structures*.

KEYWORDS

Persistent threads, Task scheduling, Concurrent queue, Irregular workloads, GPU computing

ACM Reference Format:
David Troendle, Tuan Ta, and Byunghyun Jang. 2019. A Specialized Concurrent Queue for Scheduling Irregular Workloads on GPUs. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3337821.3337837

1 INTRODUCTION

GPUs are proven, powerful accelerators for data and compute intensive applications. They achieve high throughput by running a massive number of threads, each of which operates on a part of a dataset. Unlike traditional processors, GPU threads are created, scheduled, and destroyed by hardware, and the programmer has no control over the order of thread execution. GPUs have two levels of hardware thread scheduling: one that assigns a software thread group (i.e., workgroup in OpenCL and thread block in CUDA terminology) to GPU cores (i.e., Compute Units (CUs) in OpenCL and Streaming Multiprocessors (SMs) in CUDA), and another that schedules hardware thread groups (i.e., wavefront in AMD and warp in NVIDIA terminology) on the Single Instruction Multiple Data (SIMD) engines (we use OpenCL terminology hereinafter to simplify the presentation). This GPU hardware thread scheduling/execution model imposes programming challenges for workloads that require a certain thread execution order. For example, in graph traversal algorithms, multiple threads traversing different parts of a graph may need to run in a specific order to satisfy dependencies among vertices. Such workloads cannot fully benefit from GPU acceleration without a special programming technique.

Data irregular workoads are those whose execution flow and parallelism change dynamically at runtime depending on data [1]. While there are other forms of irregularity (e.g, associated with control flow or memory access patterns [1]), efficiently dealing with data irregularity has been one of the most difficult challenges in GPU programming. A scheduling technique known as *persistent threads* is a viable method

for scheduling data irregular workloads[1] on GPUs [8, 17]. The persistent thread model creates just enough threads to saturate the GPU compute cores, where all threads stay alive until the end of a kernel. Tasks (e.g., data to process) are created dynamically by a kernel and scheduled to running threads under an algorithm-specific task dependency constraint. A queue data structure (or variant) plays a critical role in designing the task scheduler. The queue is shared by all threads and thus requires atomic serialization of its shared access variables.

Designing a persistent thread task scheduler that performs well under a GPU's SIMT execution model is a difficult task because of the significantly increased thread contention caused by the massive number of threads and SIMT's lock-step execution. Also, GPUs do not support dynamic memory management at runtime and limit the size of available memory. In this paper, we propose a non-blocking, bounded, array-based concurrent queue that scales well in a GPU's massively multi-threaded SIMT environment. Our proposed concurrent queue exhibits two novel properties that significantly reduce thread contention and other associated overhead. The first property is named "*retry-free*" where the enqueue/dequeue operations never suffer from retry overhead from queue-empty and queue-full exceptions[2]. The atomic operations that serialize accesses to shared resources never trigger a retry mechanism. The second property is named "*arbitrary-n*", which refers to queue operations that operate on an arbitrary number of queue entries for the same cost as a single entry. The arbitrary-n property allows a proxy thread to perform all atomic operations on behalf of all threads in the group.

To demonstrate the performance and scalability of the proposed queue, we used representative irregular workloads, a top-down Breadth First Search (BFS) and a persistent thread model to schedule tasks to threads using the proposed concurrent queue. To show the performance contribution of the proposed queue's novel properties (i.e., retry-free and arbitrary-n), we then compare that to the same BFS implementations using two other queue designs that lack these properties. Finally, we compare the performance of our BFS implementation with well-known, state-of-the-art top-down BFS implementations found in the literature. Our experiments show that due to the improved efficiency of the proposed concurrent queue, our implementation outperforms the closest competing top down BFS algorithm by up to 36.23×. Although we use the proposed queue in a persistent thread task scheduler, it can be used for other purposes on GPUs with little change and without losing its performance and scalability.

[1]We simply refer to *data irregular workloads* as *irregular workloads*.
[2]Queue-full exceptions abort the kernel because there is insufficient space to store ready tasks, but do not retry.

2 BACKGROUND AND RELATED WORK

2.1 Irregular workloads on GPUs

The challenges associated with processing irregular workloads on GPUs have been well-studied. Che et al. [14] developed a suite of OpenCL applications to study irregular graph workloads. Tzeng et al. [17] studied task scheduling for irregular GPU workloads, from a single monolithic task queue to distributed queuing with task stealing and donation. They also proposed static and dynamic dependency-aware scheduling schemes for irregular workloads, and studied them with H.264 intra prediction video compression and the N-Queens constraint satisfaction problem [1].

In irregular workloads, a task may depend on the completion of other task(s) before it can be scheduled for execution. As a task progresses, it can clear dependencies in other tasks. When all dependencies for a task clear, that task can be scheduled for execution. Processing an irregular workload requires a mechanism for scheduling the dynamic subset of independent tasks. A common approach is to use a software scheduler that is shared by all threads. This requires atomic serialization of the shared access variables, which in turn causes significant atomic contention in a GPU's massively parallel environment. An effective irregular workload scheduler must be aware of the GPU's unique thread execution model, and mitigate its adverse effects.

2.2 Persistent thread model

The persistent thread model launches just enough independent threads [8] to saturate the hardware. Those persistent threads remain alive until all tasks complete. A task scheduler assigns ready tasks to hungry persistent threads. The scheduler holds unique tokens that identify the ready tasks. When a persistent thread needs new work, it requests a task token from the scheduler. As a thread executes a task, it may produce new tasks by clearing dependencies. When this happens the thread stores the unique token(s) of the newly discovered independent task(s) in the scheduler.

Algorithm 1 Persistent thread model.

```
1: while WorkRemains() do
2:     if GetWorkToken() then
3:         DoWorkUnit()
4:         ScheduleNewlyDiscoveredWorkTokens()
5:     end if
6: end while
```

Algorithm 1 shows the basic structure of persistent thread model. Each pass through lines 1–6 is called a *work cycle* that all threads execute as long as any task remains. Line 2 requests a task token from the scheduler. If it gets work, line 3 works on the task associated with the task token, and line 4 schedules any work tokens whose dependencies were cleared by the work done on line 3. If the thread fails to get work at line 2 it simply loops until all work is done or it gets work.

2.3 Concurrent Data Structures (CDS)

In multithreaded shared memory systems, threads communicate and synchronize with each other through data structures in logically shared memory. Concurrent Data Structures (CDSs) play a crucial role in achieving good performance on such systems. CDS research evolved as an alternative to mutual exclusion serialization strategies such as critical sections. Traditionally, CDSs are implemented using two techniques: blocking and non-blocking, and their characteristics are classified as follows [11, 12]:

- Obstruction-free: A competing thread makes progress only after the interference from other threads ceases.
- Lock-free: *At least one* competing thread makes progress after finite time.
- Wait-free: *All* competing threads make progress after finite time.

Non-blocking CDSs guarantee that if one or more active threads try to perform operations on a shared data structure, some operations will complete in finite time. Cederman et al. [2] showed that non-blocking CDSs perform better than blocking ones in most cases. Most state-of-the-art CDSs are lock-free and implemented using *Compare and Set/Swap (CAS)* operations to manage shared variable access.

A concurrent stack (Treiber et al. [16]), queue (Valois et al. [18]) or deque (Michael et al. [10], Valois [19]) are potential candidates for the persistent thread task scheduler. While a deque is a more general form of a queue, the scheduler requires only the features of a simpler single ended queue. A stack's push and pop operations compete for a single shared access location, the top of stack pointer, which increases contention. For these reasons, a concurrent queue is most commonly selected as the underlying data structure for the persistent thread task scheduler.

3 CHALLENGES IN CONCURRENT QUEUE DESIGN AND OUR APPROACH

The massively threaded GPU environment imposes unique design challenges on a persistent thread task scheduler. The queue data structure plays a critical role in the efficiency and scalability of the scheduler. This section summarizes the major challenges in designing an efficient persistent thread task scheduler.

3.1 Limited memory management

Dynamic parallelism in irregular workloads is highly data dependent and difficult to bound. This strongly suggests basing the queue on a dynamically allocated, unbounded linked-list. However, the GPU run-time does not support dynamic memory allocation, and its programming APIs restrict total memory allocation in the device's memory space. All application data, including supporting data structures such as the scheduler queue, must be allocated statically before kernel launch and fit in the space available.

Due to the constraints on dynamic allocation and available memory size, a GPU design should contemplate the reuse of queue entries. A queue implemented with CAS-based lock-free linked-list usually requires versioning overhead to deal with the pointer recycling problem, which is also known as the ABA problem [19]. A linked-list also adds the overhead of linking neighbor nodes. That overhead typically triples the space required to store a simple task token.

An alternative to an unbounded, linked-list based queue is a bounded, circular, array-based queue. The queue payload data is typically the integer index of a task ready for thread assignment. Thus each queue entry contains only the task token payload. There is no other space overhead.

3.2 Managing accesses to shared data

All threads must access the shared queue, which requires atomic serialized access on the queue's head and tail to avoid race conditions. There are two suitable widely used atomic operations available on modern GPUs: CAS (Compare and Set/Swap) and Atomic Fetch-Add (AFA). Each has advantages and disadvantages on GPUs as discussed below:

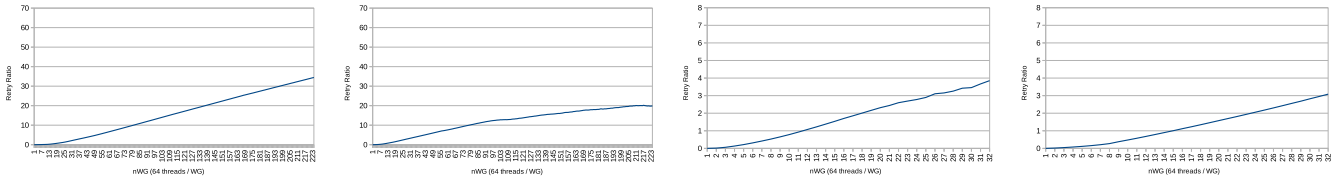
- **CAS** takes three parameters – a target, expected value and new value. If the target equals the expected value the target is set to the new value. While there are some implementation variations, all CAS operations return a value that allows the programmer to know if the set succeeded. Because a CAS can fail, atomic contention can be mitigated. In lock-free CAS-based CDS implementations, often the program simply retries on CAS failure. This introduces the overhead of retrying an unsuccessful operation.
- **AFA** adds an arbitrary value to its atomic target, and returns the old value of the target. It does not return until the operation successfully completes. This can result in long latency and contended hot spots [13]. However, the program does not need to retry as it never fails.

AFA offers an advantage over CAS on GPUs: *While the latency of both AFA and CAS atomic operations can be hidden by a GPU, the overhead of retrying an unsuccessful CAS cannot be hidden.* Since an AFA never fails, there is no retry cost, and the cost associated with atomic serialization can be effectively hidden with zero-cost thread switching on GPUs. Our experiments demonstrate that an AFA-based queue scales significantly better than CAS-based ones.

Figure 1 reveals that CAS failures increase as the number of actively running threads increases. §6.3 (*Retry overhead*) analyzes the consequences of retry overhead.

3.3 Lock-step execution

SIMT lock-step execution affects queue design more than any other GPU architectural feature. The threads in a wavefront share a common Program Counter (PC) and execute in lock-step. Thus, all wavefront threads share the same code path. Threads not using a code path used by other threads idle through that code path, treating their instructions as if they were No Operations (NOPs). This is called *thread divergence*. No thread in a wavefront completes until the longest running



(a) LKS on Fiji. (b) USA on Fiji. (c) LKS on Spectre. (d) USA on Spectre.

Figure 1: Retries caused by CAS failure for a top-down BFS (Breadth First Search) application.

thread completes. Faster threads idle until the longest running thread completes.

In a persistent thread model, thread divergence can be a significant source of performance degradation that impacts the efficiency of persistent thread task scheduling if tasks do not have homogeneous complexity. Within a wavefront, work cycles with complex tasks delay the completion of less complex tasks. In many cases, an effective solution for this problem is to refactor tasks into sub-tasks with uniform complexity For instance, many graph algorithms are based on traversing the children of vertices. While the processing complexity of a vertex varies with the number of children, the processing complexity for each child is roughly uniform. Therefore, this issue can be addressed by processing a fixed number[3] of uniformly complex sub-tasks in each work cycle instead of traversing all children. Hungry threads (i.e., threads that need work) can request a task after each work cycle, which has been refactored to a fixed number of homogeneous complexity sub-tasks.

The GPU's SIMT lock-step execution exacerbates the latency of atomic operation, especially AFA operation because they wait their turn rather than fail. If all threads within a SIMT are hungry and each performs an AFA-based dequeue operation, then the latency of the operation could be increased by the factor of the number of SIMT threads. This could easily become a performance bottleneck and must be avoided. Our approach to avoiding such intensified atomic contention and overhead is to have a representative thread (named as proxy thread) perform all atomic operations on behalf of all threads in the SIMT group.

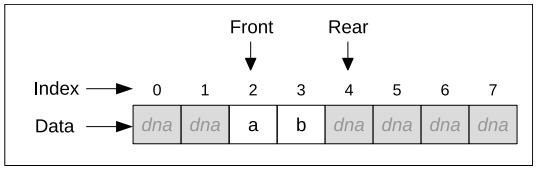


Figure 2: Proposed queue structure and sentinels. The circular property of the array is implemented by manipulating *Front* and *Rear* indices (i.e., modulus operation).

[3] Empirically we found work cycles of 4 sub-tasks works well.

4 DESIGN AND IMPLEMENTATION OF CONCURRENT QUEUE

A concurrent queue specialized for use as a GPU persistent thread task scheduler should avoid the adverse effects of lock-step execution while accessing shared data. Our proposed queue is implemented using a bounded array and AFA operations to manage access to the shared queue resources. It also uses a sentinel named *data-not-arrived* to remove queue operation exceptions (e.g., queue empty), which, when combined with AFA, enables the use of *proxy threads*. We store a data-not-arrived sentinel in all queue slots where valid data has not yet arrived. Figure 2 depicts the proposed queue structure. The data-not-arrived sentinels are labled as "*dna*" in shaded entries. Rather than actually storing or retrieving task tokens, the queue operations return the slot index where the task tokens will be enqueued or dequeued.

4.1 Atomic operations by proxy thread

We arbitrarily chose the first thread in each wavefront as the proxy thread. The proxy thread performs all global atomic operations for all threads (including itself) using a single atomic operation. It functions as a normal thread except for atomic operations. Suppose three threads are hungry in Figure 2, then the proxy thread performs:

$$StartSlotIndex = \texttt{atomic\_fetch\_add}\left(Front, 3\right)$$

$StartSlotIndex$ is set to 2, and $Front$ atomically advances to 5. The first, second, and third hungry threads are assigned slot index 2, 3, and 4 respectively. All slot assignments are done in parallel. The first two threads have data and start processing it immediately. The third thread sees its data has not yet arrived (*dna* sentinel) and checks again in each subsequent work cycle until data arrives. Since only the proxy thread performs an atomic operation, contention for the queue access variable is significantly reduced and completes faster.

4.2 Dequeue operation

Dequeue operations feed each hungry thread a unique slot index to monitor. The atomic operation occurs once when a thread becomes hungry. Since the slot index is unique, only a simple global memory operation is required to check for data arrival. When a thread sees data has arrived (no sentinel), it picks up its work token from its slot and replaces it with the sentinel. This has the effect of refactoring an atomic queue

Listing 1: Dequeue kernel code snippet.

```
1    // Get base index of the slots for hungry threads.
2    if (IsProxyThread)  {
3      lnQueueSlotsNeeded = 0u;
4    }
5
6    if (ThreadNeedsWork)  {
7      // Count all threads and assign each thread its relative slot index;
8      DequeueThreadSlotIndex = atomic_inc(&lnQueueSlotsNeeded);
9    }
10
11   // Get base index of the slots for hungry threads.
12   if (IsProxyThread && lnQueueSlotsNeeded)  {
13     lQueueSlotBaseIndex = atomic_add(&Parms->WorkQueueFront, lnQueueSlotsNeeded);
14   }
15
16   if (ThreadNeedsWork)  {
17     DequeueThreadSlotIndex += lQueueSlotBaseIndex;
18     ThreadNeedsWork = false;
19     QueueDataAvailable = false;
20    }
```

Listing 2: Data arrival kernel code snippet.

```
1  if (!QueueDataAvailable)  {
2    // Check to see if data has arrived.
3    if ((DequeueThreadSlotIndex < QueueSize) &&
4        (QueueDataAvailable = (WorkQueue[DequeueThreadSlotIndex] !=
5         Missing)))  {
6
7      // Work has arrived. Setup to process this node.
8      // No atomics are needed because this is the only thread accessing
9      // the slot or node.
10
11     // Get work token (index of node to process).
12     CurrentNodeIndex = WorkQueue[DequeueThreadSlotIndex];
13
14     // Get assigned node.
15     CurrentNode = Nodes[CurrentNodeIndex];
16
17     // Get starting edge for this node.
18     CurrentEdge = Edges + CurrentNode.StartingEdgeIndex;
19
20     // Get current node cost;
21     CurrentNodeCost = Costs[CurrentNodeIndex];
22   }
23 }
```

empty exception to a non-atomic memory check for data arrival.

Listing 1 shows the kernel code snippet for the dequeue operation. In Lines 2-4, `lnQueueSlotsNeeded` which will contain a count of the number of hungry threads in the wavefront for a given work cycle is zeroed by the proxy thread. In Lines 6–9, the hungry threads (`ThreadNeedsWork`) are assigned a slot. Line 8 increments the number of hungry threads (`lnQueueSlotsNeeded`) and assigns the private variable `DequeueThreadSlotIndex` a slot index relative to the wavefront. Later, this will be converted to an actual queue slot index. The local atomic operations are executed by all hungry wavefront threads in lock-step. Like their global atomic counterparts, they do not fail and the GPU can hide the latency associated with these operations.

In Lines 12–14, the proxy thread reserves queue slots for all the hungry threads in the wavefront. To avoid unnecessary atomic contention, this is done only if there is at least one hungry thread. Line 13 performs the actual allocation. The base index of the reserved area is stored in `lQueueSlotBaseIndex`, and `WorkQueueFront` is atomically incremented by the number of slots allocated. Line 17 converts the thread's wavefront relative slot index to an actual queue index unique for this thread. Lines 18-19 mark that the thread is no longer hungry and needs to check for data arrival.

Listing 3: Enqueue kernel code snippet.

```
1    // Initialize
2    if (IsProxyThread)  {
3    lnQueueSlotsNeeded = 0u;
4    }
5
6    // Count all newly discovered work in this cycle and assign slot index
7    // for each thread.
8    if (nNewlyDiscoveredWork)  {
9    EnqueueThreadSlotIndex =
10   atomic_add(&lnQueueSlotsNeeded, nNewlyDiscoveredWork);
11   }
12
13   // Reserve space in queue, and get base index.
14   if (IsProxyThread && lnQueueSlotsNeeded)  {
15   lQueueSlotBaseIndex = atomic_add(&Parms->WorkQueueRear, lnQueueSlotsNeeded);
16   }
17
18   if (nNewlyDiscoveredWork)  {
19          // Convert slot index to base index within queue.
20          EnqueueThreadSlotIndex += lQueueSlotBaseIndex;
21
22          // Copy newly discovered work to the queue slot reserved for this
23          // work token.
24          for (uint32_t i = 0u; i < nNewlyDiscoveredWork; ++i)  {
25            if (WorkQueue[EnqueueThreadSlotIndex] != Missing) QueueFullAbort();
26                WorkQueue[EnqueueThreadSlotIndex++] = NewlyDiscoveredWork[i];
27          }
28   }
```

4.3 Data arrival

Listing 2 is the kernel snippet that checks data arrival, which occurs when the thread's unique slot index no longer has the *dna* sentinel. Lines 1-23 are executed only if data has not yet arrived, which is signaled by `QueueDataAvailable`. Lines 3-5 perform the actual data arrival check. No atomic operations are required. They ensure the assigned slot index is within queue bounds and the data at the slot index is no longer the *dna* sentinel. If data has arrived, it sets `QueueDataAvailable` to true. As the kernel concludes, no new tasks are discovered. Threads are assigned slots to monitor where no data will ever arrive. The slot may, in fact, be outside the queue bounds and cannot be accessed. Lines 3-5 handle these details. The enqueue operation ensures data is never stored out of bounds. Lines 6-22 are executed once just before node enumeration occurs. They form the enumeration prolog and setup for child enumeration.

4.4 Enqueue operation

Enqueue operations proceed in a complementary manner to dequeue operations. The proxy thread reserves indices for all newly independent tasks discovered this work cycle using a single AFA operation. All threads in the wavefront then copy their data to the queue in parallel. The threads monitoring that slot see the newly arrived data and begin processing it. Each thread ensures the slot contains a sentinel before writing its data. If a sentinel is not there, then a queue full exception has occurred. When a queue full exception occurs the problem is too large for the allocated queue size. It indicates there are more available tasks ready for execution than can be stored in the queue. Buffer space, including the queue, is allocated by the host prior to kernel execution. If more space can be allocated, the user can retry the kernel with a larger queue.

Listing 3 shows the kernel snippet for the enqueue operation. In Lines 8-11, if a thread has enqueued new tasks, the number of new task tokens is counted. Each thread will be

assigned the number of slots needed. The base of that area relative to the wavefront is stored in `EnqueueThreadSlotIndex`. Later it will be converted to an actual queue index. In Lines 14-16, the proxy thread reserves queue slots for all newly discovered task tokens in the wavefront. To avoid unnecessary atomic contention, this is done only if there is at least one newly discovered task token. Line 15 performs the actual allocation. The base index of the reserved area is stored in `lQueueSlotBaseIndex`, and `WorkQueueRear` is incremented by the number of slots allocated. to an actual queue index. Lines 24-27 copy each newly discovered task token index to its queue slot in lock-step. This overwrites the *dna* sentinel. The thread monitoring this slot sees the arrival in its next work cycle when it executes lines 3-5 of Listing 2. In Line 25, tokens can only be enqueued to slots where data has not yet arrived, which is signaled by the presence of a sentinel value. If the sentinel is not present, a queue full exception has occurred, which aborts the kernel.

5 EXPERIMENTAL SETUP

The performance of an ideal queue must scale as threads are added. This is especially important in a massively threaded GPU environment, but difficult to achieve due to increased thread contention on shared resources and lock-step execution. The objective of our experimental setup is to reveal the performance and scalability characteristics of the proposed queue.

5.1 BFS as a driver application

We chose Breadth First Search (BFS) to test the proposed concurrent queue for use in a persistent task scheduler. BFS is an important, fundamental graph algorithm that finds numerous applications in many different fields. It is often considered as a representative irregular workload on GPUs. While faster BFS algorithms exist [9], we chose a classic top-down BFS algorithm [4, 15] because it is well-known and well-suited for driving the proposed queue. It traverses a graph in a width-first manner starting from a source vertex. In a multi-threaded environment, all threads at any given level enumerate their children, and must complete their enumerations before next level processing can begin. This is the source of data irregularity and dynamic parallelism.

5.2 Input graph datasets

For BFS, the number of vertices available for processing at any given instant depends on the input dataset. Therefore, input datasets must be carefully chosen to thoroughly evaluate the performance and scalability of the proposed queue. This ranges from a synthetic dataset that massively saturate threads to road map datasets that do not saturate the hardware. We selected six diverse graph datasets in three categories as test input data. The three categories are:

- **Synthetic:** To analyze the scalability of the proposed persistent scheduler without the influence of other factors, we constructed a synthetic dataset designed to keep all persistent threads busy. This ensures kernel performance differences are due only to thread contention and not due to idle threads (lack of parallelism). Figure 3a shows the number of vertices available for thread assignment at each level for synthetic dataset, which has 10,485,760 vertices, with a fanout of 4 edges per vertex. After the first 8 levels, both the Spectre and Fiji GPUs are fully saturated. This effectively removes lack of work as a source of poor acceleration.
- **Social media:** Social media graphs and their processing speed are becoming increasingly important as Social Networking Service (SNS) gets popular. We selected two representative social media datasets [7] as detailed in Table 1. Typically social media graphs have a large edge fanout[4], but are not very deep. The two datasets cover small- and medium-sized social media graphs. Figures 3b and 3c show this characteristic graphically as well as their dynamic parallelism.
- **Roadmap:** Roadmap graphs typically have a fanout of between 2 and 3, but are deep. Table 2 details the three selected roadmap datasets from the 9^{th} DIMACS implementation challenge [5]. Each selected dataset has about 10× more vertices and edges than the next smaller dataset. This covers a broad spectrum of roadmap graphs. Because roadmap graphs are so deep, the number of vertices available at any given level is smaller than in social media graphs. Figures 3d, 3e and 3f show this characteristic graphically. Only the USA dataset saturates the Spectre with a small number of CUs to any significant degree. Thus, insufficient data parallelism is a limiting factor in this category.

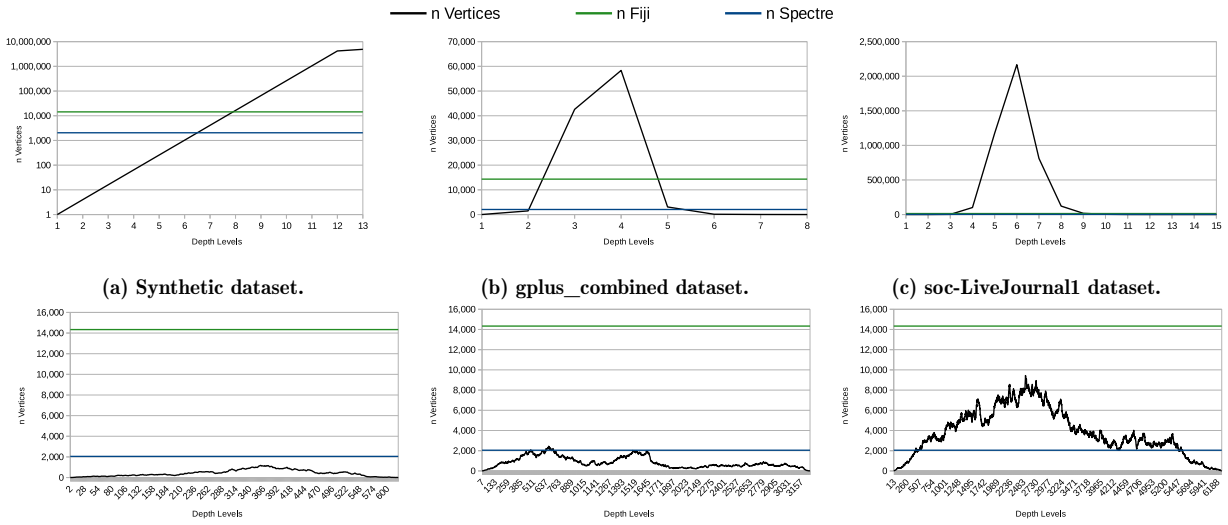
5.3 Dissecting the effects of queue properties

We designed and implemented three concurrent queue variations to expose the effects of the retry-free and arbitrary-n properties respectively:

- **BASE**: This is a traditional lock-free concurrent queue using CAS atomic operations. This version has neither the retry-free nor arbitrary-n properties.
- **AN**: This queue variant adds the arbitrary-n property to BASE. This version retries on CAS failures.
- **RF/AN**: This is the proposed retry-free, arbitrary-n concurrent queue. It uses non-failing AFA atomics.

The difference between the AN and RF/AN queue variations exposes the effect of the retry-free property on performance, while the difference between the BASE and AN queue variations exposes the effect of the arbitrary-n property on performance.

[4] The large fanout of social media graphs present a design challenge. As edges are discovered, they must be stored in local or private memory before being queued. Private and local memory are scarce resources that limit the number of edges that can be processed. Our proposed queue and the Rodinia benchmark avoid this issue, but is an issue for the CHAI BFS benchmark.



(a) Synthetic dataset. (b) gplus_combined dataset. (c) soc-LiveJournal1 dataset.

(d) NY dataset. (e) LKS dataset. (f) USA dataset.

Figure 3: Dynamic data parallelism in our selected input graph datasets.

Dataset	n Vertices	n Edges	Edges Per Vertex			
			Min	Max	Avg	Std
gplus_combined	107,614	30,494,866	0	49,041	283.4	1,245.18
soc-LiveJournal1	4,847,571	68,993,773	0	20,293	14.2	36.08

Table 1: Selected SNAP social media graph datasets statistics.

Dataset	Description	n Vertices	n Edges	Edges Per Vertex			
				Min	Max	Avg	Std
USA-road-d.NY	New York City	264,346	733,846	1	8	2.8	0.98
USA-road-d.LKS	Great Lakes	2,758,12	6,885,658	1	8	2.5	0.95
USA-road-d.USA	Full USA	23,947,347	58,333,344	1	9	2.4	0.95

Table 2: Selected 9th DIMACS implementation challenge roadmap dataset statistics.

5.4 Programming language and test hardware

We chose an OpenCL 2.0 programming environment because it is an established non-proprietary cross platform industry standard. However, porting to CUDA should not lose any intellectual merit.

All experiments were performed on two hardware platforms: a powerful high-end discrete GPU (AMD Radeon R9 Fury codenamed *Fiji*), and a low-end integrated GPU with shared CPU-GPU memory (AMD Radeon R7 APU codenamed *Spectre*). The Spectre GPU has 8 CUs and shares memory with the CPU. The Fiji GPU has 56 CUs and separate device memory.

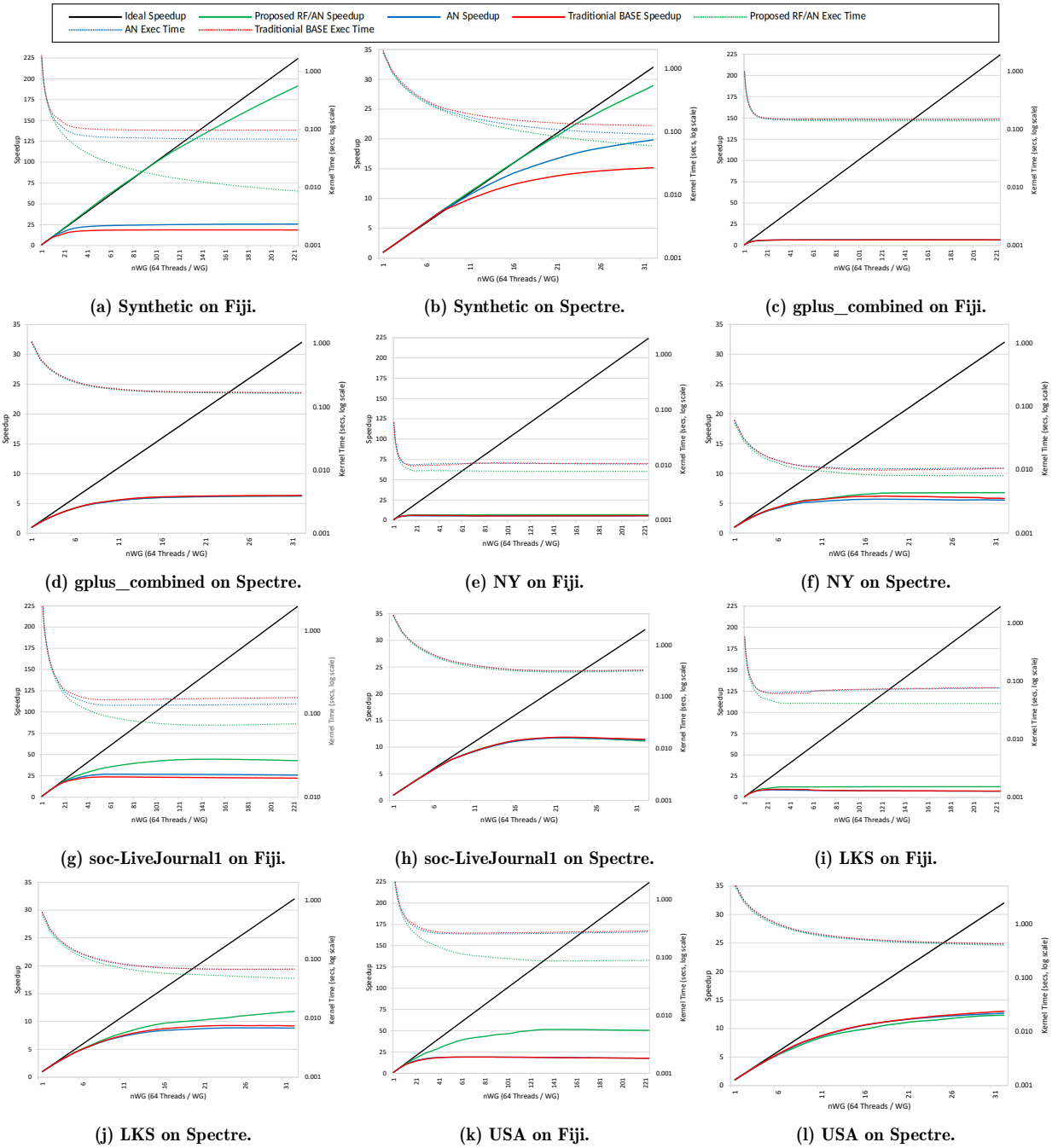
We used a workgroup size of one wavefront (64 threads) to avoid barriers, and launched 4 workgroups on each CU to facilitate zero-cost thread switching. This resulted in 2,048 persistent threads (32 workgroups of 64 threads) on the integrated Spectre GPU, and 14,336 persistent threads (224 workgroups of 64 threads) on the discrete Fiji GPU.

6 EXPERIMENTAL RESULTS AND ANALYSIS

The performance and scalability of the proposed queue are investigated under a variety of loads using a persistent thread implementation of top down BFS. We also compare the performance of our BFS implementation to the ones found in the well-known benchmark suites, CHAI [6] and Rodinia [3].

6.1 Kernel execution time

Kernel execution time combines all factors affecting performance into a single, absolute metric. Table 3 shows the execution times for each queue variant, dataset and GPU. The proposed retry-free/arbitrary-n queue (RF/AN) is the fastest in all cases. Table 4 shows the kernel execution time improvement of the AN and RF/AN kernels relative to the BASE kernel. Figure 4 illustrates execution time and speedup for the three queue implementations across all datasets and hardware as the number of threads (workgroups) increases.



(a) Synthetic on Fiji.

(b) Synthetic on Spectre.

(c) gplus_combined on Fiji.

(d) gplus_combined on Spectre.

(e) NY on Fiji.

(f) NY on Spectre.

(g) soc-LiveJournal1 on Fiji.

(h) soc-LiveJournal1 on Spectre.

(i) LKS on Fiji.

(j) LKS on Spectre.

(k) USA on Fiji.

(l) USA on Spectre.

Figure 4: Execution time and speedup across different input graph datasets and hardware.

The Fiji discrete GPU uses 14,336 threads (224 workgroups of 64 threads), and the Spectre integrated GPU uses 2,048 threads (32 workgroups of 64 threads).

Figures 4a and 4b show the synthetic dataset speedup and execution time for the discrete and integrated GPUs respectively. The synthetic dataset removes lack of work as a factor limiting acceleration, and thus reveals the performance limiting effects of atomic contention and retries. The speedups of the proposed retry-free/arbitrary-n queue are within 10% of the ideal. When the atomic operations are replaced by a CAS-based lock-free queue, AN, the increased atomic contention and retry overhead clearly limits speedup. When the CAS-based lock-free version of the proposed queue

GPU	nWG	Dataset	BASE	AN	RF/AN
Fiji	224	Synthetic	0.09760	0.06777	0.00865
		gplus_combined	0.15066	0.15066	0.14229
		soc-LiveJournal1	0.15778	0.13217	0.07642
		USA-road-d.NY	0.01056	0.01038	0.00767
		USA-road-d.LKS	0.07808	0.07706	0.04172
		USA-road-d.USA	0.28393	0.27274	0.08829
Spectre	32	Synthetic	0.12501	0.09125	0.05957
		gplus_combined	0.16799	0.16736	0.16343
		soc-LiveJournal1	0.32705	0.32428	0.31613
		USA-road-d.NY	0.01055	0.01064	0.00808
		USA-road-d.LKS	0.06764	0.06789	0.04722
		USA-road-d.USA	0.42379	0.41971	0.40307

Table 3: Execution times (in seconds) of queue variants across different datasets and hardware.

	Performance improvement over BASE			
	Fiji		Spectre	
Dataset	AN	RF/AN	AN	RF/AN
Synthetic	144.03%	1128.12%	137.00%	209.86%
gplus_combined	100.00%	105.88%	100.37%	102.79%
soc-LiveJournal1	119.38%	206.46%	100.85%	103.45%
USA-road-d.NY	101.70%	137.57%	99.18%	130.58%
USA-road-d.LKS	101.33%	187.14%	99.63%	143.24%
USA-road-d.USA	104.10%	321.60%	100.97%	105.14%

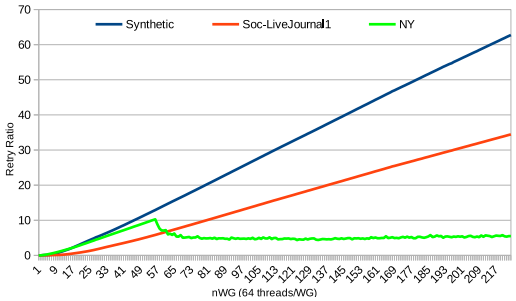
Table 4: Performance improvement of RF/AN and AN over BASE.

is replaced by a traditional lock-free queue, BASE, speedup is further limited by the way a traditional queue handles a dequeue attempt on an empty queue.

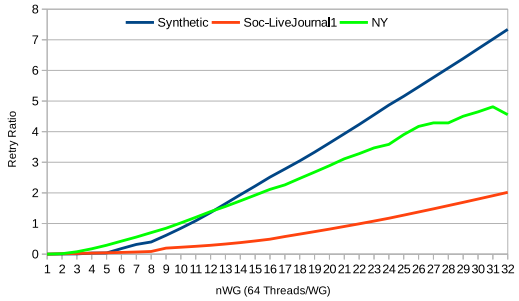
Figures 4c thru 4h show the execution time and speedup for the selected social media graphs. The gplus_combined dataset (Figures 4c and 4d) only briefly saturates the persistent threads, which limits speedup. The faster speed of the more capable discrete GPU (Fiji) does reduce execution time, but has no effect on speedup. The soc-Live-Journal1 dataset (Figures 4g and 4h) saturates more threads, but only the Fiji GPU shows modest improved scalability. Figures 4e thru 4l show the execution time and speedup for the selected roadmap graphs. None of these graphs saturate the Fiji GPU, and only the USA graph partially saturates the Spectre GPU. Except for the USA dataset on the Fiji GPU, all have similar scaling characteristics. The above suggests that speedup for the proposed queue is very sensitive to thread saturation (dynamic parallelism) because idle threads do not contribute to acceleration.

6.2 Scalability

Scalability is measured by comparing the speedup of the three studied queue implementations as threads (workgroups) are added. All speedups are computed relative to using one workgroup. Ideally, increasing the number of threads should proportionally decrease the execution time. The ideal speedups are shown as the black straight lines in Figure 4 for comparison purposes. In practice, actual speedup falls off due to increased atomic contention and other overhead as workgroups are added. Notice the left side of all graphs (smaller number of WGs) shows little difference in either speedup or execution time. In this region there are relatively few persistent threads, and most of those threads are saturated by available work. Because there are relatively few threads, atomic contention and retries have little effect. As the number of threads increase, the adverse effects of overhead become more pronounced.



(a) Fiji.



(b) Spectre.

Figure 5: Retry ratios (BASE over PROPOSED) for selected datasets.

6.3 Retry overhead

Minimizing retry overhead is the single-most important factor for our design. Its effects are evident in Figure 4 by comparing the green (a design with no retry overhead) and red (a design with retry overhead) lines for any dataset or GPU. There are two major sources of retry overhead – retries due to atomic operation failure and retries due to queue operation exceptions. We dealt with atomic operation failure by choosing AFA, and queue operation exceptions by operating on slot indicies rather than queue entries. This allowed the use of proxy threads, which further reduced atomic overhead.

Retry overhead is not always easily seen. In datasets with good parallelism such as Figure 4a, however, the gap between the green and red lines is pronounced. Good parallelism helps saturate the cores, which increases atomic competition and overhead. In datasets with sparse parallelism, such as Figure 4e, the gap between the green and red lines is virtually nonexistent. Lack of parallelism causes many idle threads, which reduces atomic competition and overhead.

Figure 5 emphasizes this effect for three selected datasets (synthetic, soc_LiveJournal1, and NY) on the Fiji (Figure 5a) and Spectre (Figure 5b) GPUs. The retry ratio is the ratio of total atomic operations used by a kernel over the number of operations required by our design.

The synthetic dataset was designed to saturate the persistent threads. As the number of threads increases, so do retries for the BASE kernel. Figure 5a shows the BASE queue requires over 60× more atomic operations than the proposed queue when the largest number of threads is used on the discrete Fiji GPU. The soc_LiveJournal1 and NY datasets have progressively less parallelism. The result is corresponding less retry overhead due to reduced thread saturation and atomic competition.

The soc_LiveJournal1 dataset has a high fanout with a large standard deviation. This is characteristic of social media graphs. The result is relatively good, consistent data parallelism, and thus a higher retry ratio when the number of threads is high. The NY dataset has a small but consistent fanout (as do many roadmap graphs). This results in poor thread saturation.

Initially, the NY dataset has good parallelism, but falls off as threads are added. This is clearly visible in the green line on the left side of Figure 5a. The Spectre GPU follows this trend. The retry overhead falloff just becomes apparent on the right side of Figure 5b.

6.4 BFS performance comparison

To evaluate the overall performance of BFS implemented using the proposed concurrent queue, we compare it with two other BFS implementations found in the literature.

(1) **CHAI [6]**: CHAI is a benchmark suite for tightly integrated heterogeneous platforms. There are implementations for programming languages such as OpenCL 2.0, CUDA 8.0, and C++ AMP, and true heterogeneous implementations that exploit productive collaboration between CPU and GPU threads. The BFS included in the benchmark suite uses a top-down algorithm and persistent threads. Unlike our implementation, however, it uses a heterogeneous CPU/GPU model, while ours uses only the GPU.

(2) **Rodinia [3]**: Rodinia is a benchmark suite for heterogeneous computing to help architects study emerging platforms. Rodinia includes applications and kernels that target multi-core CPU and GPU platforms. The BFS implementation in this benchmark suite uses a top-down algorithm with course grain buffers. It exits after each level and allocates 1 thread per node. Only nodes with no dependencies process at each level. If the number of levels is significant, this approach can have significant overhead.

For both BFS benchmarks, we use the test datasets and configuration parameters chosen by the original authors. This helps eliminate any bias in dataset choice, and ensures the benchmarks were run as the authors intended. All tests were run with the GUI off to eliminate the GUI overhead on the GPU.

6.4.1 Comparison to the CHAI BFS benchmark. The CHAI BFS benchmark provides two datasets to test the performance of their heterogeneous BFS kernel. The discrete Fiji GPU cannot run this heterogeneous kernel because it does not support cross cluster CPU/GPU atomic operations. Their test datasets are relatively small road map graphs with only modest dynamic parallelism. Table 5 details the kernel times for CHAI and the proposed queue (RF/AN). All times are in milliseconds. Our proposed algorithm outperforms CHAI BFS by at least 2.57 times.

Dataset	CHAI	RF/AN	Speedup
NYR_input.dat	20.8015	8.0811	2.574×
USA-road-d.BAY.gr.parboil	20.8998	4.9691	4.206×

Table 5: Performance comparison with CHAI BFS (ms).

Dataset	Device	Rodinia	RF/AN	Speedup
graph4096	Spectre	6.7436	0.2227	30.28×
	Fiji	5.9282	0.2048	28.95×
graph65536	Spectre	17.9806	1.6257	11.06×
	Fiji	13.6875	0.3778	36.23×
graph1MW_6	Spectre	111.758	32.7679	3.41×
	Fiji	4.4950	3.5640	1.26×

Table 6: Performance comparison with Rodinia BFS (ms).

6.4.2 Comparison to the Rodinia BFS benchmark. The Rodinia BFS benchmark provides three synthetic test datasets. The datasets have 4K, 64K and 1M vertices. None of the three datasets has more than 11 levels, and have good dynamic parallelism, especially for the largest dataset. The effect of these dataset choices is that the smaller datasets have relatively more overhead than the large dataset. This can be seen in Table 6. The Rodinia tests were run on both the Spectre and Fiji GPUs. Table 6 compares the kernel times for Rodinia and the proposed queue (RF/AN) run on the same datasets.

6.5 Analysis Summary

In practice, performance overhead arises from two issues – lack of dynamic parallelism and retrying failed operations.

There are two sources of retries: retries due to queue empty exceptions and retries due to atomic operation failure. The synthetic dataset artificially removes queue empty exceptions so that the effects of atomic overhead can be easily seen. Figures 4a and 4b exposes the limiting effects of atomic retries on scalability. Overhead increases as the thread count increases, thus limiting a GPU's ability to accelerate an application.

Retries due to queue empty exceptions occur when there are more threads than tasks available for those threads. This occurs when there is poor dynamic parallelism. The proposed queue mitigates the effects of queue empty exceptions by ensuring a hungry thread performs only one atomic operation to obtain a task, and by assigning a unique slot to monitor for task arrival. The check for task arrival is a non-atomic global memory read on the assigned slot.

Retries due to atomic operation failure are due to the choice of atomic operation used. The proposed queue uses only non-failing atomic operations. Non-failing atomic operations are often associated with long latency. However, GPUs can hide

long latency operations by using zero-cost thread switching to a ready thread. GPU cannot hide the cost of retrying a failed atomic operation.

Figures 4c thru 4l expose the effects of limited dynamic parallelism. The smallest dataset is the NY roadmap dataset (Figures 4e and 4f), and has the poorest dynamic parallelism. Even when dynamic parallelism is limited, the elimination of atomic retry overhead and the efficient task arrival strategy employed by the proposed queue results in a 2.57× speedup over CAS-based queue implementations such as those found in CHAI BFS.

7 CONCLUSION

The persistent thread model is an intuitive solution for irregular workloads on GPUs. However, a difficult performance dilemma arises: If there are sufficient tasks to saturate the persistent threads, atomic contention and retries limit acceleration. If there are not sufficient tasks to saturate the persistent threads, the resulting idle threads limit acceleration.

This paper presents a novel retry-free, arbitrary-n concurrent queue for suitable for use as a GPU persistent task scheduler. It eliminates retries and significantly reduces atomic contention. It significantly boosts acceleration when all persistent threads are saturated. Using BFS as a driver application, we demonstrate the performance and scalability characteristics of the proposed queue across various real-world input graph datasets and hardware configurations by comparing with state-of-the-art implementations found in the literature.

REFERENCES

[1] M. Burtscher, R. Nasre, and K. Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[2] Cederman, Daniel and Tsigas, Philippas. 2008. On Dynamic Load Balancing on Graphics Processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH '08)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 57–64. http://dl.acm.org.umiss.idm.oclc.org/citation.cfm?id=1413957.1413967

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[5] DIMACS. [n. d.]. DIMACS Challenge. http://dimacs.rutgers.edu/Challenges/ http://dimacs.rutgers.edu/Challenges/.

[6] J. Gómez-Luna and I. E. Hajj and L. W. Chang and V. García-Floreszx and S. G. de Gonzalo and T. B. Jablin and A. J. Peña and W. m. Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 43–54. https://doi.org/10.1109/ISPASS.2017.7975269

[7] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[8] K. Gupta and J. A. Stuart and J. D. Owens. 2012. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *2012 Innovative Parallel Computing (InPar)*. 1–14. https://doi.org/10.1109/InPar.2012.6339596

[9] H. Liu and H. H. Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807594

[10] Maged M Michael. 2003. CAS-based lock-free algorithm for shared deques. In *European Conference on Parallel Processing*. Springer, 651–660.

[11] Michael, Maged M. and Scott, Michael L. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. ACM, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[12] Moir, Mark and Shavit, Nir. 2004. Concurrent Data Structures.

[13] Morrison, Adam and Afek, Yehuda. 2013. Fast Concurrent Queues for x86 Processors. *SIGPLAN Not.* 48, 8 (Feb. 2013), 103–112. https://doi.org/10.1145/2517327.2442527

[14] S. Che and B. M. Beckmann and S. K. Reinhardt and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195. https://doi.org/10.1109/IISWC.2013.6704684

[15] Robert Sedgewick. 1984. *Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[16] R Kent Treiber. 1986. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center.

[17] Tzeng, Stanley and Patney, Anjul and Owens, John D. 2010. Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics (HPG '10)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 29–37. http://dl.acm.org.umiss.idm.oclc.org/citation.cfm?id=1921479.1921485

[18] John D Valois. 1994. Implementing Lock-Free Queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*. 64–69.

[19] John D. Valois. 1995. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 214–222. https://doi.org/10.1145/224964.224988