2019

Scheduling Irregular Workloads on GPUs

David Arthur Troendle
University of Mississippi

Scheduling Irregular Workloads on GPUs

A Dissertation
presented in partial fulfillment of requirements
for the degree of Doctor of Philosophy
in the Department of Computer and Information Science
The University of Mississippi

by

David Troendle

December, 2018

ABSTRACT

Graphic Processing Units (GPUs) have emerged as proven, powerful accelerators for data and compute intensive applications. It creates a massive number of threads, each of which operates on a part of the problem. When all threads complete the problem is solved. Data irregular workloads which hold dynamic data dependencies and parallelism, however, pose unique programming/performance challenges because a thread cannot make progress until all dependencies are cleared. This requires a special scheduling mechanism shared by all threads and atomic access to the shared scheduler. The scheduler is usually implemented with a queue data structure. The use of atomic operations to synchronize access to a data structures, however, comes with overhead that severely limits scalability. As the number of threads increases, overhead disproportionally increases. This results in diminished thread effectiveness because more of a thread's computing potential is lost to overhead.

This doctoral research aims at understanding the nature of the overhead for data irregular GPU workloads, proposing a solution, and examining the consequences of the result. We propose a novel, retry-free GPU workload scheduler for irregular workloads. When used in a Breadth First Search (BFS) algorithm, the queue scales to within 10% of ideal scalability on a Fiji GPU with 14,336 active threads. The proposed scheduler is based on a simple, monolithic concurrent queue[1]. The dissertation presents research that shows the retry overhead associated with Compare and Swap (CAS) operations[2] is the principle reason why concurrent queues do not scale well as the number of clients increase in a massively multi-threaded environment.

The proposed concurrent queue is based solely on non-failing atomic operations such

[1]The queue works like a traditional queue except that enqueue/dequeue operations return the queue index where data is stored (enqueue) or will appear (dequeue).

[2]Traditional state-of-the-art concurrent queues are based on CAS operations. CAS operations can fail. Failures force retries.

as `atomic_add`, `atomic_inc` and `atomic_min`. Since these operations never fail, there is no retry overhead. The non-failing atomic operations also allow an arbitrary number of elements on each enqueue or dequeue operation for the same cost as operating on a single element. Limited dynamic parallelism can result in situations where there are more threads than available tasks. When this happens, atomic dequeues result in retries caused by queue empty failures. The dissertation presents a novel solution that transforms the atomic queue empty failure to a non-atomic data arrival problem. The result is a wait-free concurrent queue that has no retires – neither from the atomic operations used to manage queue access nor from the dequeue queue empty exception.

The consequences of this research suggest a refocus of research from atomic overhead reduction to thread saturation. The techniques used in this dissertation effectively reduce overhead, leaving idle threads as the most significant remaining scalability-limiting factor. To address this problem, the dissertation also proposes a speculate and correct single-source, shortest path (SSSP) algorithm that effectively saturates the GPU.

DEDICATION

This research is dedicated to my wife, Jean, and my children Michael and Michelle.

I also dedicate this work to my teachers who, throughout my life, have believed in me and always took the time to show me the way. Foremost among my teachers is my advisor, Dr. Byunghyun Jang. Very few would have given an older student a chance to earn a PhD. Throughout my graduate studies, he has believed in me, saw me through all the failures that are part of research, encouraged me to keep trying and celebrated my successes. In no small part, my success is his success.

ACKNOWLEDGEMENTS

The candidate acknowleges the help and support he received from past and current members of the HEteROgEneous Systems research (HEROES) lab. These include past members Dr. Kyoshin "Joel" Choo; Md. Mainul Hassan, MS; Esraa Gad; Xiaoqi "Chelsea" Hu, MS and Ajay Sharma, MS. Current members include Mengshen "Mason" Zhao and Hossein Pourmeidani, MS.

A special thanks goes to past member Tuan Ta, who pointed out the importance of concurrent data structures, which is the foundation of much of this research.

Contents

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGEMENTS v

INTRODUCTION 1

1.1 Overview . . . 1

1.2 Foundation . . . 2

1.3 Porting to a GPU . . . 4

1.4 Dealing with dynamic parallelism . . . 5

1.5 Contributions . . . 6

1.6 Organization of dissertation . . . 7

BACKGROUND 8

2.1 Thread synchronization on GPUs . . . 8

2.2 Irregular workloads on GPUs . . . 9

2.3 Concurrent data structures (CDS) . . . 10

2.4 Persistent Thread Model . . . 11

2.5 Excess Persistent Threads . . . 12

2.6 Bread-First Search on GPUs . . . 12

2.7 Single-Source Shortest Path . . . 13

PERSISTENT TASK SCHEDULING DESIGN CHALLENGES **15**

3.1 Persistent task scheduling design challenges 15

3.2 CAS failure 17

3.3 Empty queue checking 18

3.4 Lock-step execution 19

CONCURRENT QUEUE FOR PERSISTENT GPU THREADS **21**

4.1 Concurrent queue for persistent GPU threads 21

4.2 Example Queue Operation 22

4.3 Wait-Free, Retry-Free, Arbitrary-n Dequeue 24

4.4 Data Arrival Details 24

4.5 Wait-free, retry-free, arbitrary-n enqueue 26

KERNEL DESIGN **28**

5.1 Kernel design 28

5.2 Persistent thread considerations 28

5.3 Porting considerations 29

5.4 The chunked persistent thread model 30

5.5 Queue operation considerations 31

EXPERIMENTAL SETUP **33**

6.1 BFS driver application and its data dependency 33

6.2 Input graph datasets 34

6.3 Confidence interval 36

6.4 Programming language and test hardware 38

ANALYSIS OF PROPOSED QUEUE **40**

7.1 Optimal queuing method and chunk size 40

7.2 Effects of the arbitrary-n and retry-free properties on performance. 55

7.3 Scalability 56
7.4 BFS performance comparison 58

SSSP GPU SPECULATE AND CORRECT ALGORITHM **61**
8.1 Motivation 61
8.2 Bellman-Ford SSSP algorithm 62
8.3 Proposed GPU speculate and correct SSSP algorithm 63
8.4 SSSP benchmark datasets 70
8.5 Speculate and correct SSSP kernel performance details 71
8.6 SSSP benchmark comparison 72

RELATED WORK **75**
9.1 Concurrent data structures 75
9.2 GPU persistent thread scheduling 78
9.3 GPU graph algorithms 79

SUMMARY AND CONCLUSION **80**

BIBLIOGRAPHY **82**

Appendices **88**

Kernel Support Variables **89**

Code Listing: The Direct Dequeue / Enqueue Kernel **91**

Code Listing: The Direct Dequeue / Proxy Enqueue Kernel **95**

Code Listing: The Proxy Dequeue / Direct Enqueue Kernel **100**

Code Listing: The Proxy Dequeue / Enqueue Kernel **105**

Code Listing: SSSP Kernel **110**

List of Figures

3.1 CAS retry overhead for BFS on various datasets and hardware. 16
3.2 Queue empty retry overhead for BFS on various datasets and hardware. . . . 17
3.3 Wavefront lock-step execution. 19
4.1 Proposed queue structure and operation. 22
6.1 BFS traversal strategy. 33
6.2 Area under Z PDF for p=0.95 and p=0.99 36
7.1 Synthetic graph dependency clearance by depth level. 41
7.2 BFS kernel execution time by device/queuing algorithm/chunk size (synthetic data). 42
7.3 gplus_combined analysis. 43
7.4 soc-LiveJournal1 analysis. 45
7.5 USA-road-d.NY analysis. 47
7.6 USA-road-d.LKS analysis. 49
7.7 USA-road-d.USA analysis. 51
7.8 Execution time and speedup. 54
7.9 Traditional speedup curves. 56
7.10 Scalability. 56
8.1 Vertex/Edge terminology. 64
8.2 Speculate and correct SSSP performance on selected datasets/GPUs. 71

List of Tables

6.1 Selected SNAP social media graph datasets statistics. 34

6.2 The 9^{th} DIMACS implementation challenge dataset statistics 34

7.1 Performance comparison with CHAI BFS (ms). 58

7.2 Performance comparison with Rodinia BFS (ms). 59

8.1 Selected ninth DIMACS implementation challenged dataset statistics. 69

8.2 SSSP performance (average kernel time in ms) summary. 73

CHAPTER 1

INTRODUCTION

1.1 Overview

GPUs accelerate applications by mapping a massive number of threads to data and computation tasks to solve a problem. Early GPU designs required a streaming-style thread mapping with no or little dependencies among threads. For applications amenable to that requirement, GPUs offer an effective, power efficient solution with impressive scalability and acceleration. However, for data irregular applications holding dynamic dependencies and parallelism, results were disappointing. This dissertation aims at identifying the overhead sources causing the poor performance in data irregular workloads and proposing novel solutions. Two main sources of inefficiency we identified are:

1. Retries caused by failed atomic operations.
2. Retries caused by empty conditions in dequeue operations when no task is available for an idle thread,

This dissertation describes the research that identified the above overhead, and proposes the following novel solutions:

1. A wait-free, retry-free queue that uses only non-failing atomic operations,
2. Refactoring the atomic dequeue queue empty failure into a non-atomic data arrival problem.

In general, lack of available work remains an intractable acceleration limiting problem for data dependent applications. However, in some cases, it is possible to speculate rather

than wait for dependencies to clear. If incorrect speculations can be detected and corrected, then GPU threads can be kept saturated with either speculation or correction tasks. The technique is effective if the number of incorrect speculations is small enough. This dissertation demonstrates the effectiveness of the speculate and correct approach on the SSSP problem.

1.2 Foundation

GPUs have emerged as powerful application accelerators. Acceleration is achieved by applying a massive number of threads to a problem. The more effectively an algorithm exploits the parallelism, the better the acceleration. From the earliest Multi-Core Multi-Threaded (MCMT) implementations, exploiting thread parallelism has proven a challenging design problem. The two most common issues are workload balancing (keeping all threads busy) and thread interference caused by atomic operations.

In a GPU environment, the problem is further complicated by the GPU's architecture. On a MCMT processor, thread instruction paths are always independent of each other, while on a GPU they are not always independent[1]. Early GPU implementations allowed only workloads with full thread independence (i.e., no thread depends on the work done by any other thread). This allowed the hardware scheduler to schedule work in any order, usually with the primary objective of hiding global memory latency.

As GPUs matured, they eventually added the hardware support that enabled processing workloads with data dependencies. Work with outstanding dependencies cannot be scheduled for execution until all its dependencies are cleared. While data dependent workloads were allowed, GPUs provided no scheduling support. It was left to the programmer to detect when all dependencies have cleared and schedule that work for execution. Thus, an efficient scheduler is essential to the successful GPU acceleration of data dependent workloads

[1]GPUs cluster threads into thread groups that execute in lock-step. This creates an artificial relationship between threads in a group that does not exist in a MCMT environment, and complicates GPU algorithm design.

and is an important research problem.

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)[2] on the SIMD engines. The GPU hardware thread 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 benefit from GPU acceleration without a special programming technique.

Data irregular workloads [5] are those whose execution flow and parallelism change dynamically at runtime depending on data. While there are other forms of irregularity (e.g, associated with control flow or memory access patterns – see Burtscher et al. [5]), efficiently dealing with data irregularity has been one of the most difficult design challenges in GPU programming. A programming technique known as *persistent threads* [31] has emerged as a compelling solution for accelerating such irregular workloads[3] [31, 54]. The persistent thread model creates enough threads to saturate the GPU provided there are available tasks. All threads stay alive until the end of a GPU kernel. Computing tasks can be formed dynamically throughout the kernel and scheduled to running threads under an algorithm-specific task dependency constraint. When a persistent thread needs a task to execute, it obtains a unique token identifying a task from the task scheduler. When a running task completes, it may make other tasks ready for execution by passing the unique tokens for the newly runnable tasks to the scheduler. A queue data structure (or variant) plays a critical role in designing the task scheduler. The queue is shared by all threads and requires

[2]We use OpenCL terminology hereinafter to simplify the presentation.
[3]We simply refer to *data irregular workloads* as *irregular workloads*.

atomic access to the shared queue access variables. A CAS operation is typically used to manage access, but a CAS succeeds for only one competing thread, forcing the unsuccessful competitors to retry until they succeed.

Designing a persistent thread task scheduler that performs well under a GPU's Single Instruction Multiple Data [40] (SIMD) thread model is a difficult task arising from two factors: 1) CAS failure retry overhead, and 2) atomic contention issues arising from the lock-step execution nature of a GPU's SIMD thread groups. CAS failure retries disproportionally increase as the number of competing threads increases, and thus limit efficiency. In a lock-step execution environment, no thread can make progress until all CAS operations in the lock-step execution thread group succeed. This dissertation proposes a wait-free array-based concurrent queue data structure to address the design issues of a persistent thread task scheduler. The proposed concurrent queue has the following novel properties:

- **Retry-free:** This property ensures the atomic operations managing access to the shared queue variables never retry. They successfully complete first time, every time. Further, dequeue failures due to an empty queue condition have been refactored and do not cause dequeue retries. Dequeue failures are handled outside the queue mechanism, and require no atomic operations.

- **Arbitrary-n:** Each queue operation can operate on an arbitrary number of entries for the cost of a single entry. Lock-step execution of atomic operations in a GPU wavefront can cause intense contention. The arbitrary-n property allows a single proxy thread to perform atomic operations on behalf of all threads in a lock-step execution thread group. This property allows a simple monolithic queue to serve as an efficient persistent thread task scheduler, and avoids the overhead of multi-level queues.

1.3 Porting to a GPU

Often porting a data dependent algorithm begins with a multi-threaded CPU version. Sometimes the algorithm is an established CPU algorithm that needs to be ported to a GPU.

Sometimes it is a new algorithm that is easier to develop and debug in a multi-threaded CPU environment. Rendering an algorithm in a GPU environment has proven a difficult challenge that has been mastered by few. The proposed queue structure was designed to ease this task. This dissertation describes how a typical MCMT thread can be refactored into a persistent thread using the proposed queue. It identifies how each major section of a MCMT threaded algorithm can be refactored into a GPU kernel using the proposed queue.

1.4 Dealing with dynamic parallelism

An application is an algorithm applied to a dataset. When there are no dependencies in the dataset, tasks can be performed in any order. Data dependencies force tasks to be performed in a specific order. Only the subset of tasks with no dependencies can be scheduled to a thread. Typically, as a task runs it clears dependencies held by other tasks, allowing them to be scheduled. *Dynamic parallelism* is the number of tasks eligible for processing at any instant in time. Often, the number of threads exceeds dynamic parallelism. When this happens, the threads with no tasks assigned do not accelerate, which limits the overall ability of the GPU to accelerate the application.

It is possible for an algorithm to speculate if it can detect and correct when it has speculated incorrectly. *Speculate and correct* is a form of dynamic programming that speculates rather than waiting when it encounters a for a data dependency to clear, and corrects when it detects an incorrect speculation. If speculations are sufficiently correct, this can allow the algorithm to ignore data dependencies and saturate all threads by speculating. When an incorrect speculation is detected, tasks dependent on the incorrect speculation are rescheduled for correction.

This involves two queues: One queue handles speculation and the other queue handles corrections. The correction queue has a higher priority. In this manner, corrections are performed before speculation can propagate the effects of an earlier incorrect speculation. One important example of this approach is SSSP. SSSP finds the short weighted path

from a single source vertex to all descendants. The Bellman-Ford algorithm [2, 21] can be viewed as a speculate and correct algorithm. The algorithm makes at most $|V| - 1$ passes on the graph. Each pass detects and corrects path errors made in a previous pass. If a pass makes no corrections, no further passes are required. The algorithm is inefficient in a GPU environment, because there are sufficient threads to concurrently correct any errors and immediately queue their descendants for correction. The net result is that the SSSP problem can be done in a single pass, while keeping all threads busy. For the DIMACS [17] roadmap datasets, the speculate and correct algorithm improved performance by two orders of magnitude for small GPUs such as the AMD Spectre, and one order of magnitude for large GPUs such as the AMD Fiji. This dissertation details the SSSP GPU speculate and correct algorithm.

1.5 Contributions

The contributions of dissertation are summarized as follows.

1. A highly scalable concurrent queue data structure is proposed and implemented for massively multi-threaded GPU architectures.

2. BFS is implemented to demonstrate and analyze the performance characteristics of the proposed queue.

3. A refactoring process is presented to aid in the rapid migration of CPU thread-safe algorithms to an equivalent GPU algorithm.

4. BFS performance is compared with state-of-the-art competitors found in the literature.

5. A novel speculate and correct SSSP algorithm is presented that outperforms state-of-the-art algorithms found in the literature by two orders of magnitude for small GPUs such as the AMD Spectre and one order of magnitude for larger GPUs such as the AMD Fiji.

1.6 Organization of dissertation

The remaining chapters of this dissertation are organized as follows: Chapter 2 gives background information; Chapter 3 outlines persistent thread scheduling design challenges; Chapter 4 details the proposed queue; Chapter 5 discusses the kernel design; Chapter 6 details the experimental setup; Chapter 7 analyzes the proposed queue; chapter 8 presents a speculate and correct SSSP algorithm; Chapter 9 gives related work; and chapter 10 summarizes results and gives concluding remarks.

CHAPTER 2

BACKGROUND

2.1 Thread synchronization on GPUs

The GPU thread execution model clusters threads into groups called *wavefronts*. From the programmer's perspective, all threads in a wavefront appear to execute in lock-step[1]. To hide memory latency, GPUs use a zero-cost wavefront switching mechanism. If a wavefront stalls on a long latency operation (e.g., memory read or write), the GPU attempts to switch to another ready wavefront. The programmer does not have control over this hardware thread (wavefront) scheduling. This requires all threads within a wavefront to be independent in order to avoid deadlock situations, and imposes significant limitations on thread synchronization and communication.

Algorithm 1 Critical section.

```
1: while !lock(flag) do
2: end while
3: ...
4: unlock(flag)
```

The lock-step execution nature of a wavefront has unexpected consequences when atomic operations are used to serialize access to shared resources. For example, the simple mutex-based critical section shown in Algorithm 1 causes a deadlock on a GPU. The problem is the mutex unlock on line 4 of Algorithm 1 may never execute. This is because all the threads in a wavefront simultaneously compete for the flag on line 1. The hardware picks a

[1] A wavefront is formed from several hardware SIMD thread groups. The threads in a SIMD thread group are actually executed in parallel. The wavefront is a logical grouping that appears to execute in lockstep.

winner, and all other competing threads fail. The failing threads spin (lines 1-2) on the lock until they obtain it. This never happens because when the failing threads spin, the lock-step execution of a wavefront also forces the winning thread into a No Operation (NOP) spin. This prevents the successful thread from ever executing line 4, which clears the flag. Blocking techniques from a multi-threaded CPU environment must be adjusted to work in a GPU's lock-step thread environment.

2.2 Irregular workloads on GPUs

The challenges associated with processing irregular workloads on GPUs have been well studied. Che et al. [7] developed a suite of OpenCL applications to study irregular graph workloads. Tzeng et al. [54] 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 [5].

In irregular workloads, a task may depend on the completion of some other task(s) before it can be processed. As a task is completed, it can clear dependencies for other dependent tasks so that they can be scheduled to execute. At any instant there is a dynamic subset of tasks with no active dependency that are ready to execute.

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 massively parallel GPU environment. Further, the lock-step execution of a wavefront increases simultaneous atomic access and consequently exacerbates contention issues. A practical irregular workload scheduler must be aware of the GPU's unique thread execution model.

2.3 Concurrent data structures (CDS)

In multi-threaded shared memory systems, threads perform tasks concurrently and synchronize with one another through data structures in logically shared memory. Data structures play a crucial role in achieving good performance on such systems. Concurrent Data Structure (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 [46, 45]:

- **Obstruction-free**: A thread competing for data structure access makes progress only after the interference from other threads ceases.
- **Lock-free**: *At least one* thread competing for data structure access makes progress after finite time.
- **Wait-free**: *All* threads competing for data structure access 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. [6] 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 *CAS* operations to manage shared variable access.

In CAS-based implementations, when multiple threads attempt to update a shared variable at the same time, only one succeeds while all other competitors fail and must retry. The highly threaded nature of a GPU environment tends to increase competition. A CAS implementation ensures only one competitor at a time succeeds. So, at best CAS implementations are lock-free. A wait-free CDS is better suited to highly competitive GPU environments, because all competitors succeed in finite time. However, wait-free CDSs have proven difficult to achieve [15].

2.4 Persistent Thread Model

A GPU's thread execution model clearly imposes significant disadvantages on irregular workloads. A common solution is to launch enough independent *persistent threads* [19, 31] to saturate the hardware and use a task scheduler to assign tasks to the persistent threads. The scheduler holds unique tokens that identify the independent tasks. When a persistent thread is ready for new work, it requests a task token from the scheduler. As a thread performs a task, it may produce new tasks with cleared dependencies. When this happens the thread stores the unique tokens of the newly discovered independent tasks in the scheduler.

Algorithm 2 Persistent thread model.

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

Algorithm 2 shows the basic persistent thread model. Each pass through lines 1–6 is called a *work cycle* and *all* persistent threads remain active as long as *any* task remains. Line 2 requests a work 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.

Choosing a data structure to implement the task scheduler is an important design decision. In the massively threaded GPU environment, scalability is a primary design consideration. A CDS with a wait-free property is ideal because all competing threads make progress in finite time. A concurrent stack (Treiber et al. [51]), queue (Valois et al. [57], Harris [25]) or deque (Michael et al. [44], Valois [58], Sundell et al. [50]) are potential candidates for the persistent thread task scheduler. While a deque [50] is a more general form

of a queue, the scheduler requires only the features of a much simpler single ended queue. A stack's push and pop operations compete for a single shared access location, the top of stack pointer, which causes high contention. For these reasons, we chose and developed a wait-free concurrent queue for the task scheduler. It minimizes the overhead associated with the dequeue operation used to obtain task tokens (Algorithm 2, Line 2) and the enqueue operation used to schedule task tokens (Algorithm 2, Line 4).

2.5 Excess Persistent Threads

Dynamic parallelism is a form of parallelism in which the number of independent tasks available for execution varies over time. In the persistent thread model, the number of persistent threads can exceed available dynamic parallelism. This dissertation refers to the persistent threads in excess of dynamic parallelism as *excess persistent threads*.

Even though the excess threads have no available work, they nonetheless futilely attempt to dequeue a task token each work cycle. Each futile attempt results in a queue empty exception, and increased atomic retry contention with no opportunity for benefit. The excess threads cannot be destroyed because they may be needed in the future. Since there is no software-level system (e.g., operating system) on GPUs, those threads cannot be put to sleep.

The proposed solution is to ensure an excess thread dequeues only once. This prevents excess persistent threads from retrying multiple times when there are no available tasks in the task queue. This mechanism is detailed in §4.1 (*Concurrent queue for persistent GPU threads*).

2.6 Bread-First Search on GPUs

BFS is an important, fundamental graph algorithm that finds numerous applications in many diverse fields. It has been extensively researched and optimized for CPU [28, 56, 59, 9] and recently GPU [28, 36, 37, 42] environments. Because it exhibits irregularity and

dynamic parallelism, it has been considered as a representative irregular GPU workload.

Harish et al. [24] pioneered the acceleration of BFS on GPUs, but achieved limited acceleration over CPU implementations. Deng et al. [14] introduced a Sparse-Matrix Vector Product-based formulation for BFS and achieved a 10× acceleration over CPU implementations. Luo et al. [39] presented a GPU BFS implementation using a hierarchical queue and a three-layer CUDA kernel that was intended for applications with near-regular graphs typically found in Electronic Design Automation (EDA) applications. It achieved up to 10× speedup over CPU implementations.

The Rodinia benchmark suite [7] includes a BFS implementation using course-grain atomic operations. The CHAI benchmark suite [29] includes a true heterogeneous BFS implementation. Liu et al. [37] achieved the best-performing BFS implementation known to the author. It implements a CUDA hybrid top-down/bottom-up algorithm. It's top-down performance is similar to other implementations on low-fanout deep graphs such as roadmap datasets, but the bottom-up algorithm achieves exceptional performance on high-fanout shallow graphs such as social media datasets.

Top-down BFS is a simple algorithm with an easily understood data dependency whose performance has been extensively studied on CPUs and GPUs. For these reasons, a top-down persistent thread BFS implementation was chosen to drive the queue and demonstrate its performance characteristics.

2.7 Single-Source Shortest Path

SSSP (Cormen et al. [10, pp 643–683]) finds the shortest weighted path between graph vertices starting from a single source. The weight is an abstract metric suitable for the problem. For instance, if the shortest distance is desired, then an appropriate weight would be the distance between adjacent nodes. Alternatively, if the shortest time is desired, then the travel time between adjacent nodes would be an appropriate weight. The problem can become complicated if there are cycles in the graph with negative weights.

There are several algorithms for this problem. The most notable are due to Dijkstra [16], and Bellman [2] and Ford et al. [21]. Dijkstra's algorithm does not allow cycles, while the Bellman-Ford algorithm does allow non-negative cycles. Typically, Bellman-Ford runs slower than Dijkstra.

The Bellman-Ford algorithm [2, 21] performs at most $|V|$ passes to develop the paths and detect negative cycles. Each pass detects and corrects errors made in the previous path. Each pass *speculates* the best path and *corrects* errors made in prior passes. The algorithm is inefficient in a GPU environment, because there are sufficient threads to concurrently correct any path errors as they are detected. Queuing the descendants of a corrected path for immediate correction avoids multiple passes. The correction queue has higher priority than the speculation queue. The helps minimize propagation of incorrect speculations. The net result is that the SSSP problem can be done in a single pass, and the GPU threads are kept busy either speculating or correcting. Further, the speculation frontier expands exponentially. The exceptional scaling characteristics of the proposed queue are well-suited to the speculate and correct technique. However, the technique introduces correction overhead.

CHAPTER 3

PERSISTENT TASK SCHEDULING DESIGN CHALLENGES

3.1 Persistent task scheduling design challenges

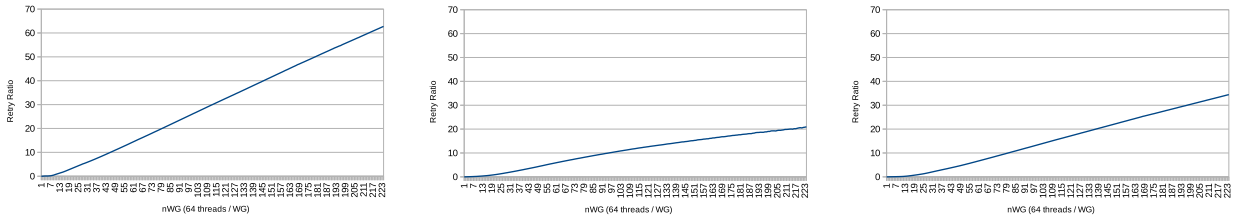
The queue data structure plays a critical role in persistent thread task scheduling. A persistent thread requests a task token from the queue when it needs work and stores a new task token in the queue when it discovers a newly independent task. The massively threaded GPU environment generates unique design challenges because a large number of threads atomically compete for queue access. Traditional CAS-based concurrent queues exhibit three major challenges when used for persistent task scheduling on GPUs.

1. **CAS failure:** CAS operations can fail if there is more than one competing thread: one succeeds while all other competitors must retry until they succeed.

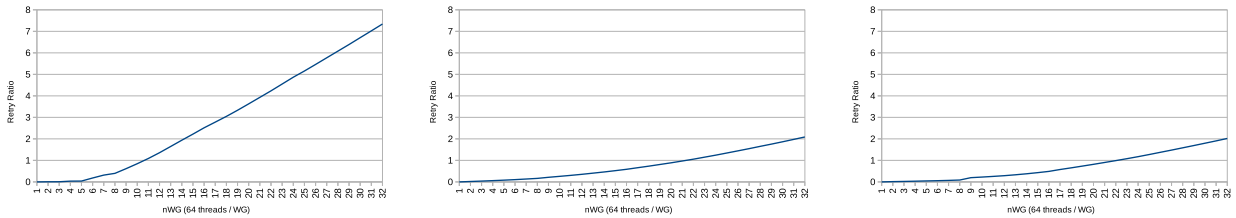
2. **Dequeue "queue empty" checking:** When there are excess persistent threads due to lack of data parallelism, the kernel must keep retrying dequeues until data is available.

3. **Lock-step execution:** Lock-step execution increases retries and delays progress until all competing threads in a wavefront succeed.

The following subsections describe each challenge in more depth.

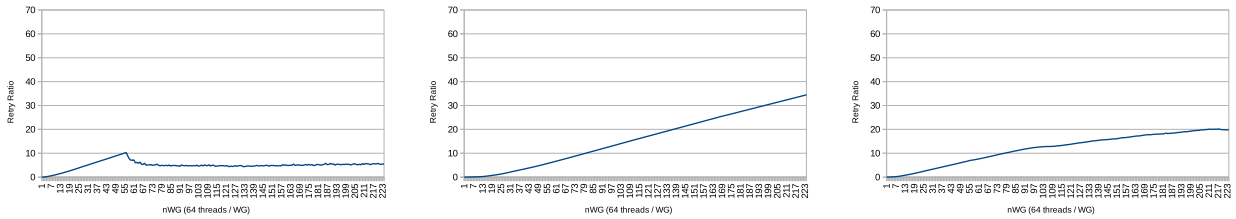


(a) Synthetic on Fiji. (b) gplus_combined on Fiji. (c) soc-LiveJournal1 on Fiji.

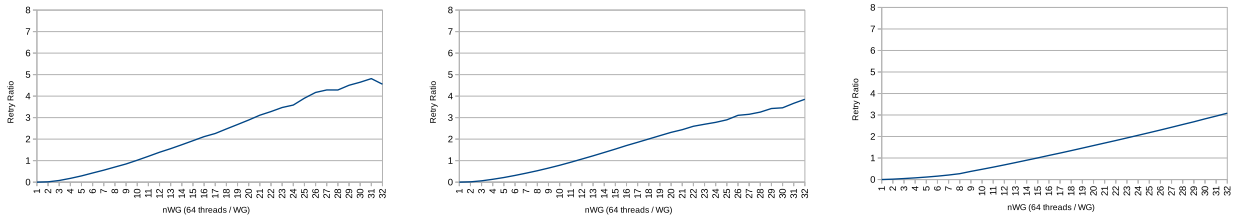


(d) Synthetic on Spectre. (e) gplus_combined on Spectre. (f) soc-LiveJournal1 on Spectre.

. . . . . .

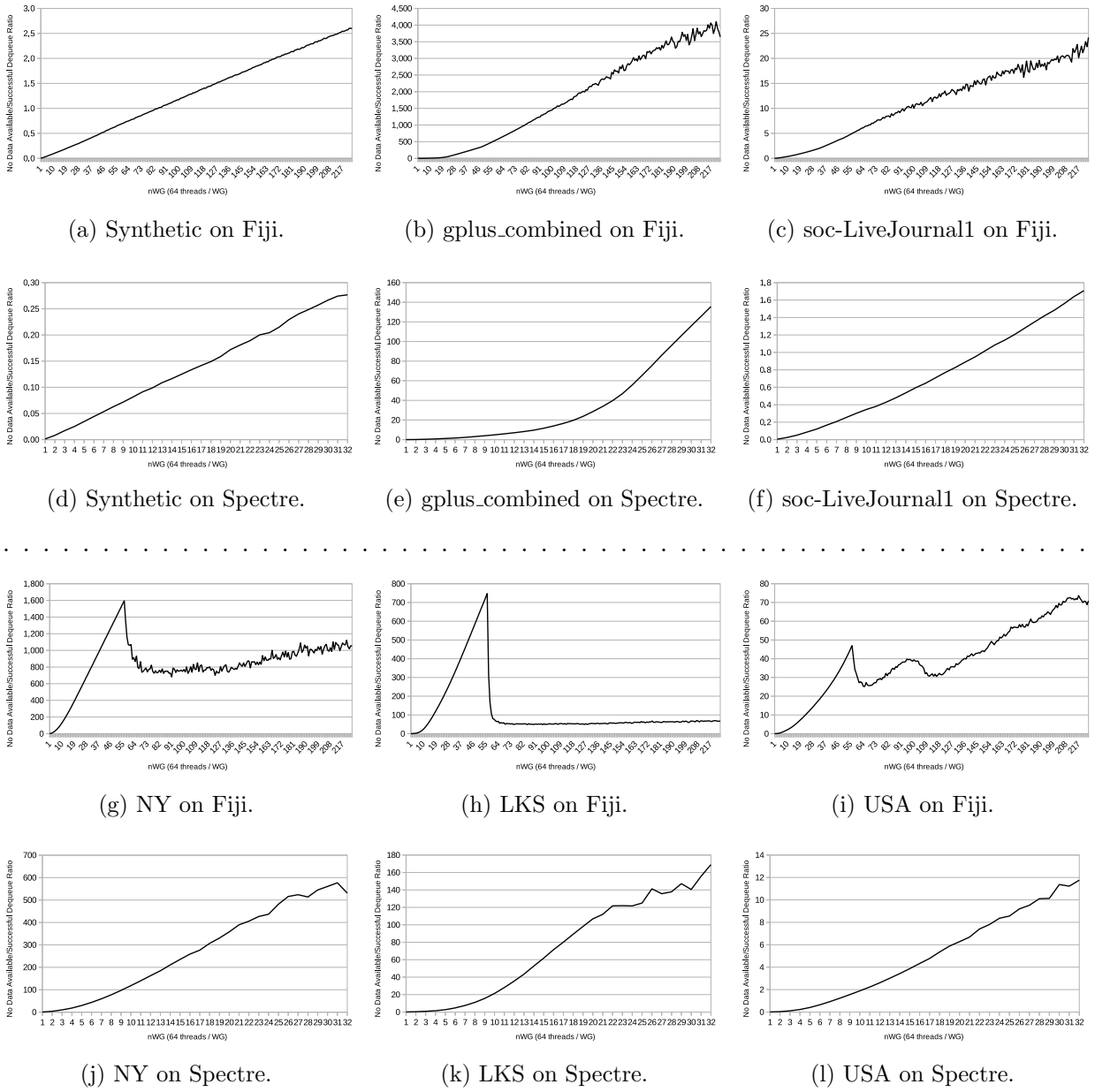


(g) NY on Fiji. (h) LKS on Fiji. (i) USA on Fiji.



(j) NY on Spectre. (k) LKS on Spectre. (l) USA on Spectre.

Figure 3.1. CAS retry overhead for BFS on various datasets and hardware.



(a) Synthetic on Fiji. (b) gplus_combined on Fiji. (c) soc-LiveJournal1 on Fiji.

(d) Synthetic on Spectre. (e) gplus_combined on Spectre. (f) soc-LiveJournal1 on Spectre.

(g) NY on Fiji. (h) LKS on Fiji. (i) USA on Fiji.

(j) NY on Spectre. (k) LKS on Spectre. (l) USA on Spectre.

Figure 3.2. Queue empty retry overhead for BFS on various datasets and hardware.

3.2 CAS failure

CAS operations can either succeed or fail. Each failure forces a retry until the operation eventually succeeds. In the best case all accesses succeed on their first attempt. In the worst case, $\mathcal{O}(n^2)$ tries are required for n operations. As the number of threads increases, so does atomic competition and thus the number of retries. Experiments show

that the actual number of retires required for n competitors lies between n (best case) and $\frac{n(n+1)}{2} = 1 + 2 + \cdots + n$ (worst case). In other words, one thread succeeds on its first attempt; another thread requires two attempts before it succeeds; etc. Figure 3.1 shows the CAS retry overhead for BFS (input data and experimental configuration are detailed in §6.2 (*Input graph datasets*)). Note that these experiments used a persistent thread model with a traditional lock-free CAS-based queue and a proxy thread. In general, contention increases as the number of threads increase (shown in terms of 64-thread workgroups on the x-axis). The y-axis shows the overhead ratio which is computed as:

$$\text{CAS } retry\ ratio = \frac{\#\text{CAS } retries}{\#Successful\ \text{CAS } ops}$$

The retry ratio varies with dynamic parallelism and the number of persistent threads, which directly correlates to the number of CUs tested (e.g., the larger Fiji GPU has more CUs and thus requires more persistent threads to saturate the hardware than the smaller Spectre GPU). Generally the retry ratio increases as the number of threads increase. However, this is not always true. For example, in Figure 3.1g, retry overhead increases up to about 50 workgroups, then declines until about 70 workgroups and remains about level thereafter. The effects of CAS retries are also dataset dependent, with larger datasets tending to have higher overhead. When there are sufficient threads to saturate the CUs, CAS retries typically increase the number of queue operations by 4- to 60-fold. This strongly motivates a queue design that minimizes CAS failure retries. The proposed queue is retry-free for both enqueue and dequeue operations and always gives best case retry performance.

3.3 Empty queue checking

The second source of retries is caused by dequeue operations that experience a queue empty exception. This is an intrinsic characteristic of all traditional queues. For example, when a thread experiences a queue empty exception on a dequeue it must retry until data is enqueued. Figure 3.2 shows the dequeue queue empty retry overhead for BFS (input data configurations are detailed in §6.2 (*Input graph datasets*)). The overhead is expressed as:

$$Dequeue\ overhead = \frac{\#Queue\ empty\ retries}{\#Successful\ dequeue\ ops}$$

The effects of the dequeue queue empty exception are highly dependent on a dataset's parallelism, and the number of persistent threads required to saturate the GPU. Retries present an intractable problem. When data parallelism is limited, the larger number of persistent threads required to saturate the GPU generate retries due to queue empty exceptions. On the other hand, when data parallelism is good, there are more threads competing for the queue and thus more CAS retries. The source of the retries does not matter. Either limits GPU acceleration.

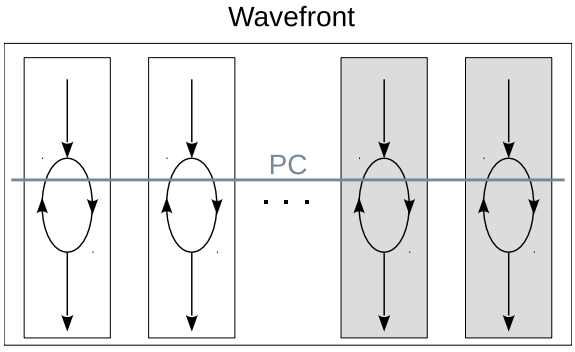


Figure 3.3. Wavefront lock-step execution.

3.4 Lock-step execution

A GPU's lock-step execution model causes two design challenges. The first is "increased retries." If there are k hungry threads in a wavefront, all k threads will simultaneously attempt a dequeue forcing a worst case of $\frac{k(k+1)}{2}$ retries. In a CPU environment this can be mitigated with a backoff technique [41], but in a GPU the programmer has no control over the GPU hardware schedulers. The second is "delayed progress." Figure 3.3 depicts a wavefront with a retry loop. The rectangular boxes represent its threads. The ellipse represents the retry loop (either due to CAS failure or a queue empty condition). The common Program Counter (PC) for all threads has progressed somewhere inside the retry loop. The shading represents competing threads that have not yet succeeded. The unshaded

threads are not competing either because they have succeeded or never needed to compete. The threads still competing force the non-competing threads into a NOP spin even though the non-competing threads are ready to proceed. The net effect is that no thread in the wavefront can progress until all competing threads succeed.

These problems are typically solved using a multi-level queue structure. The intra-workgroup level implements a queue for each workgroup and uses a proxy thread and local variables. Only the proxy thread accesses the queue, thus eliminating the need for atomic operations. The inter-workgroup level uses a shared global queue that brokers communication between workgroups. The brokering mechanism is typically either task stealing or task donation. In either case, a multi-level queue increases code length because it must detect when and where work needs to be stolen or donated.

The proposed queue design works differently. It is a monolithic queue. The intra-workgroup queue operations are handled by a proxy thread that access multiple queue entries in a single non-failing operation. Because all workgroups share the same queue, no inter-workgroup brokering is required. The operations never fail, avoiding the adverse effects of kernel retry loops.

CHAPTER 4

CONCURRENT QUEUE FOR PERSISTENT GPU THREADS

4.1 Concurrent queue for persistent GPU threads

A concurrent queue specialized for use as a GPU persistent thread scheduler should avoid the adverse effects of retries and lock-step execution. The proposed design avoids CAS retries by using a non-failing *atomic fetch-add*, and queue empty retries by using a *data-not-arrived* sentinel that is used to signal no data has been enqueued to a queue slot.

Each queue slot is initialized with either the data-not-arrived sentinel or initial task token(s) appropriate for the problem. For example, the BFS application initializes the first queue slot with the source vertex task token (i.e., vertex ID), and the remaining slots with the data-not-arrived sentinel. When a thread needing a task dequeues, it receives a unique queue slot index rather than an actual task token. This avoids detecting and handling a queue empty occurrence because a unique slot index is always available even though a task token may not yet have arrived.

Once in each work cycle, a thread checks its slot index for the arrival of a task token until a task token arrives. Since each thread has a unique slot index, no atomic operations are needed. When a task token arrives, the thread begins processing the task associated with the token. In the course of processing the task, newly independent task tokens can be enqueued. This overwrites the data-not-arrived sentinels, and the threads monitoring those slot indices detect task token arrival and begin their processing.

The use of an atomic fetch-add has another advantage. It can atomically advance the front (dequeue) and rear (enqueue) access variables by an arbitrary value. Thus, for the cost of a single atomic operation, an arbitrary number of slots can be reserved. This

allows one thread in a wavefront (typically the first) to be designated as a *proxy thread* and to perform all enqueues and dequeues on behalf of all threads in the wavefront, including the proxy thread. This avoids the adverse effects of a wavefront's lock-step execution and the need for a multi-level queue design. The combination of a data-not-arrived sentinel and atomic fetch-add brings these features to the queue design:

1. **Wait-free**: An atomic fetch-add completes in bounded time. Thus, all threads make progress.

2. **Retry-free**: An atomic fetch-add never fails and, for dequeue operations, a slot index is always available even if a task token has not yet arrived at that slot. The only two causes of retries are removed, thus making both atomic and queue operations retry-free.

3. **Arbitrary-n**: An atomic fetch-add can reserve an arbitrary number of slots in each operation. This enables a proxy thread in each wavefront to provide queue services to all threads in the wavefront. The lock-step execution of the wavefront distributes queue slots to all affected threads in parallel.

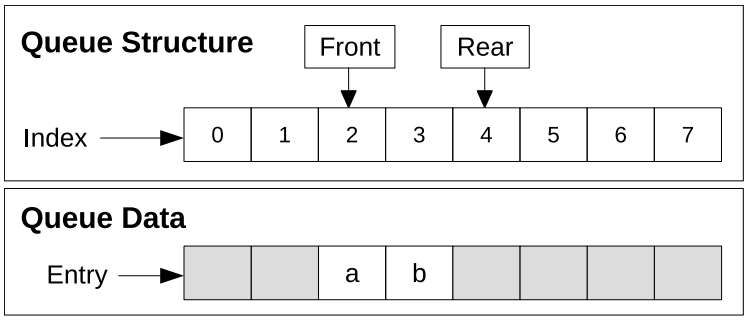


Figure 4.1. Proposed queue structure and operation.

4.2 Example Queue Operation

Figure 4.1 depicts the proposed queue structure. The data-not-arrived sentinels are shown as shaded entries. Rather than actually storing or retrieving task tokens, the enqueue and dequeue operations return the slot index where the task token will be stored (enqueue)

or is to be retrieved (dequeue). Suppose three threads are hungry (i.e., need work). The proxy thread performs:

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

$StartSlotIndex$ is set to 2, and $Front$ atomically advances to 5. The first hungry thread is assigned slot index 2; the second hungry thread is assigned slot index 3; and the third hungry thread is assigned slot index 4. All slot assignments are done in parallel. The first two threads have data and start processing it immediately. The third client thread sees its data has not yet arrived and non-atomically checks again in each subsequent work cycle until data arrives. The following subsections detail each queue operation with a code snippet from the actual kernel.

Listing 4.1. Wait-free, retry-free, arbitrary-n dequeue.

```
1  // Get base index of the slots for hungry threads.
2  if (IsProxyThread)  {
3    lnQueueSlotsNeeded = 0u;
4    lnThreadsEndingThisCycle = 0u;
5  }
6
7  if (ThreadNeedsWork)  {
8    // Count all threads and assign each thread
9    // it's relative slot index
10   DequeueThreadSlotIndex=atomic_inc(&lnQueueSlotsNeeded);
11 }
12
13 // Get base index of the slots for hungry threads.
14 if (IsProxyThread && lnQueueSlotsNeeded)  {
15   lQueueSlotBaseIndex=atomic_add(&Parms->WorkQueueFront,
16   lnQueueSlotsNeeded);
17 }
18
19 if (ThreadNeedsWork)  {
20   DequeueThreadSlotIndex += lQueueSlotBaseIndex;
21   ThreadNeedsWork = false;
22   QueueDataAvailable = false;
23 }
```

4.3 Wait-Free, Retry-Free, Arbitrary-n Dequeue

Listing 4.1 is the kernel code snippet of the wait-free, retry-free, arbitrary-n dequeue. It details how hungry threads are counted, queue slots are reserved, and how queue slots are distributed to the threads in parallel.

- **Lines 2-5:** `lnQueueSlotsNeeded` is zeroed by the proxy thread. It will contain a count of the number of hungry threads in the wavefront in a given work cycle.
- **Lines 7–11:** Each thread in the wavefront executes these lines in lock-step. If `ThreadNeedsWork` is true, the thread is hungry and will be assigned a slot. For each hungry thread, line 10 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.
- **Lines 14–17:** 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 16 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.
- **Lines 19–23:** Each hungry thread in the wavefront executes these lines in lock-step. Line 20 converts the thread's wavefront relative slot index to an actual queue index unique for this thread. Line 21 flags the thread as no longer hungry. Line 22 flags that the thread needs to check for data arrival.

4.4 Data Arrival Details

Listing 4.2 is the kernel snippet that checks data arrival, which occurs when the thread's unique slot index no longer has the data-not-arrived sentinel. It details how a thread ensures its slot index is in bounds and how the thread checks for data arrival.

Listing 4.2. Data arrival.

```
1  if (!QueueDataAvailable)  {
2    // Check to see if data has arrived.
3    if ((DequeueThreadSlotIndex < QueueSize) &&
4        (QueueDataAvailable = (WorkQueue[DequeueThreadSlotIndex] !=
5         Missing))) {
6
7      // Work as arrived. Setup to process this node.
8      // No atomics are needed because this is the only
9      // thread accessing 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 }
```

- **Lines 1-23:** This is executed only if data has not yet arrived, which is signaled by `QueueDataAvailable`.
- **Lines 3-5:** These lines 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 data-not-arrived sentinel. If data has arrived, it sets `QueueDataAvailable` to true.
- **Lines 6-22:** These lines are executed once just before node enumeration occurs. They form the enumeration prolog and setup for child enumeration.

Listing 4.3. Wait-free, retry-free, arbitrary-n enqueue.

```
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     WorkQueue[EnqueueThreadSlotIndex++] = NewlyDiscoveredWork[i];
26   }
27 }
```

4.5 Wait-free, retry-free, arbitrary-n enqueue

Listing 4.3 is the kernel snippet of the wait-free, retry-free, arbitrary-n enqueue. It details how the number of newly discovered task tokens are counted, how the slots are reserved, and how the task tokens are inserted into the queue in parallel.

- **Lines 2-4:** `lnQueueSlotsNeeded` is zeroed by the proxy thread. It will contain a count of the number of entries that need to be enqueued in this work cycle.
- **Lines 8-11:** Each thread in the wavefront executes these lines in lock-step. 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.

- **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.

- **Lines 18-27:** Each thread with newly discovered tasks executes these lines in lock-step. Line 20 converts the wavefront relative start index to an actual queue index. Lines 24-26 copy each newly discovered task token index to its queue slot in lock-step. This overwrites the data-not-arrived sentinel. The thread monitoring this slot sees the arrival in its next work cycle when it executes lines 3-5 of Listing 4.2.

CHAPTER 5

KERNEL DESIGN

5.1 Kernel design

The chapter discusses the design issues of a data irregular kernel implementing an algorithm using the persistent thread model. This dissertation studies implementations using the persistent thread model and concurrent queue discussed and developed in the previous chapters. The kernel design must be sensitive to the architecture of the hosting GPU.

5.2 Persistent thread considerations

Algorithm 2 on page 11 gives the persistent thread model. *DoWorkUnit*() performs the assigned task. However, there are no guarantees that wavefront threads are assigned tasks with homogeneous complexity. Thus, within a wavefront, longer running, more complex tasks delay faster running, less complex tasks because all threads within a wavefront run in lock-step. The effect is that the longest running *DoWorkUnit*() controls the execution time of the work cycles within wavefront. The slower running tasks NOP and do not accelerate the application.

In some cases tasks can be divided into subtasks of nearly uniform complexity. These subtasks are referred to as *chunks*. For instance, in BFS, a task processes a vertex by enumerating its children. Thus the complexity of a BFS task depends on the number of children, which can vary significantly. However, processing each child has roughly uniform complexity, and is a good candidate for a chunk. This dissertation studies the maximum number of chunks that should be processed in each work cycle.

Algorithm 3 Chunked MCMT thread model.

```
1: Prolog()
2: for up to ChunkMax chunks do
3:     DoChunk()
4: end for
5: Epilog()
```

5.3 Porting considerations

In many cases, a kernel begins life as a MCMT thread. Algorithm 3 gives a simplified view of the work performed by each thread. This corresponds to *DoWorkUnit*() on line 3 of Algorithm 2 on page 11. *Prolog()* sets up for chunk processing and may be trivially empty; *DoChunk()* processes the nearly uniformly complex chunks; and *Epilog()* handles any post-processing details.

Algorithm 4 Chunked persistent thread model.

```
1: DataArrived ← false
2: NeedsToken ← true
3: while WorkRemains() do
4:     if NeedsToken then
5:         QueueSlot ← GetWorkToken()
6:         NeedsToken ← false
7:         DataArrived ← false
8:     end if
9:     if !DataArrived then
10:        Token ← Queue[QueueSlot]
11:        if Token != DataNotArrivedSentinel then
12:            DataArrived ← true
13:            Prolog()
14:        end if
15:    end if
16:    if DataArrived then
17:        for up to ChunkMax chunks do
18:            DoChunk()
19:        end for
20:        if last chunk then
21:            NeedsToken ← true
22:            Epilog()
23:        end if
24:        ScheduleNewlyDiscoveredTokens()
25:    end if
26: end while
```

5.4 The chunked persistent thread model

Algorithm 4 shows how chunking affects the persistent thread model. It shows how the components of Algorithm 3 are ported to the chunked persistent thread model. This dissertation studies optimal number of chunks to process in each work cycle. The following details the algorithm:

1. **Lines 1-2:** Initializes flags.

2. **Lines 3-26:** Defines a work cycle.

3. **Lines 4-8:** Dequeues a queue slot if work is needed. GetWorkToken() atomically dequeues work. This operation never fails, but the slot will have the data not arrived sentinel until data arrives (is enqueued).

4. **Lines 9-15:** This executes only if data has not yet arrived. It checks if its slot contains a valid token (i.e., does not contain the "data not arrived sentinel"). When arrival is detected, it sets the data arrival flag and executes the *Prolog* to setup for chunk processing. (See Algorithm 3 line 4.)

5. **Line 13:** An example of porting a BFS *Prolog()* function can be found in Appendix E lines 141-154.

6. **Lines 16-25:** This processes at most ChunkMax chunks, and helps keep the complexity of a work cycle roughly homogeneous. When the last chunk is processed, the task is complete and it flags a new token is needed. It executes an Epilog to do any cleanup necessary.

7. **Line 18:** An example of porting a BFS *DoChunk()* function can be found in Appendix E lines 162-201.

8. **Line 22:** An example of porting a BFS *Epilog()* function can be found in Appendix E lines 241-266.

9. **Line 24:** Enqueues any newly discovered independent tasks.

5.5 Queue operation considerations

The arbitrary-n property of the proposed concurrent queue permits the use of a proxy thread to perform queue operations on behalf of all threads in the wavefront. When chunking is taken into consideration, not all threads in a wavefront will be hungry at the beginning of each work cycle. (Threads with unprocessed chunks will not become hungry until all chunks are processed.)

Thus, for each queue operation (dequeue or enqueue) there are two options – use a proxy thread or have each thread directly dequeue or enqueue. This dissertation studies the optimal queue operation configuration.

CHAPTER 6

EXPERIMENTAL SETUP

6.1 BFS driver application and its data dependency

Breadth First Search (BFS) was chosen to test the proposed concurrent queue for use as a persistent task scheduler. The classic top-down BFS algorithm (see Cormen et al. [10, pp 594–601] and Sedgewick [48, pp 395–398]) 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 BFS data dependency and dynamic parallelism. The BFS data dependency can be stated as: *enumeration of child vertices depends upon (i.e., must wait for) the completion of all vertex enumeration at the parent level.* Enumerated children must be queued until all parent level processing completes. At that point the queued children can begin enumerating their children. This processing pattern continues until no more children are enumerated.

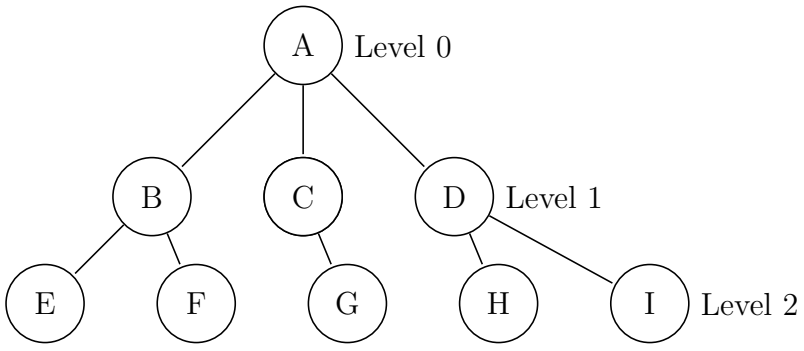


Figure 6.1. BFS traversal strategy.

For example, refer to Figure 6.1. Traversal starts at level 0 by enumerating node A's children (nodes B-D, which are at level 1). Nodes B-D cannot begin enumerating their

children until node A completes its enumeration. Level 1 processing of nodes B-D discovers nodes E-I. Enumeration of nodes E-I (at level 2) begins only after nodes B-D (at level 1) complete their enumeration. This process continues until enumeration at a level yields no new children.

The number of vertices available for processing at any given instant depends on the input dataset. Carefully chosen datasets allow for evaluation under a variety of data parallelism conditions. This ranges from a synthetic dataset that massively saturate threads to a small road map datasets that do not saturate the hardware.

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 6.1. Selected SNAP social media graph datasets statistics.

Dataset	Description	n Vertices	n Edges	Edges Per Vertex			
				Min	Max	Avg	Std
USA-road-d.BAY	San Francisco Bay Area	321,270	800,172	1	7	2.4907	0.9916
USA-road-d.CAL	California and Nevada	1,890,815	4,657,742	1	8	2.4634	0.9464
USA-road-d.COL	Colorado	435,666	1,057,066	1	8	2.4263	0.9424
USA-road-d.CTR	Central USA	14,081,816	34,292,496	1	9	2.4352	0.9525
USA-road-d.E	Eastern USA	3,598,623	8,778,114	1	9	2.4393	0.9487
USA-road-d.FLA	Florida	1,070,376	2,712,798	1	8	2.5344	0.9627
USA-road-d.LKS	Great Lakes	2,758,119	6,885,658	1	8	2.4965	0.9531
USA-road-d.NE	Northeast USA	1,524,453	3,897,636	1	9	2.5567	0.9551
USA-road-d.NW	Northwest USA	1,207,945	2,840,208	1	9	2.3513	0.9463
USA-road-d.NY	New York City	264,346	733,846	1	8	2.7761	0.9814
USA-road-d.USA	Full USA	23,947,347	58,333,344	1	9	2.4359	0.9467
USA-road-d.W	Western USA	6,262,104	15,248,146	1	9	2.4350	0.9324

Table 6.2. The 9^{th} DIMACS implementation challenge dataset statistics

6.2 Input graph datasets

For BFS, the degree of irregularity changes depending on the input graph. 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 simply idle threads. Figure 7.1 on page 41 shows the number of vertices available for thread assignment at each level. The test synthetic dataset 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 source of poor acceleration. It exposes how performance and scalability is affected by the various algorithms, hardware and thread counts.

- **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 [30] as detailed in Table 6.1. Typically social media graphs have a large edge fanout[1], but are not very deep. The two datasets cover small- and medium-sized social media graphs. Figure 7.3a on page 43 and Figure 7.4a on page 45 show this property graphically as well as its available dynamic parallelism.

- **Roadmap:** Roadmap graphs typically have a fanout of between 2 and 3 but are deep. Table 6.2 shows the roadmap datasets available in the 9^{th} DIMACS implementation challenge [17]. The datasets selected for analysis are shaded in gray. They were selected so that they cover 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. Figure 7.5a on page 47, Figure 7.6a on page 49 and Figure 7.7a on page 51 show this characteristic graphically. Only the USA dataset saturates the low-end GPUs with a small number of CUs (e.g., AMD's Spectre GPU) to any significant degree. Thus, the lack of tasks (i.e., insufficient data parallelism) is a limiting factor in this category.

[1]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. The proposed queue and the Rodinia benchmark avoid this issue, but it is an issue for the CHAI BFS benchmark.

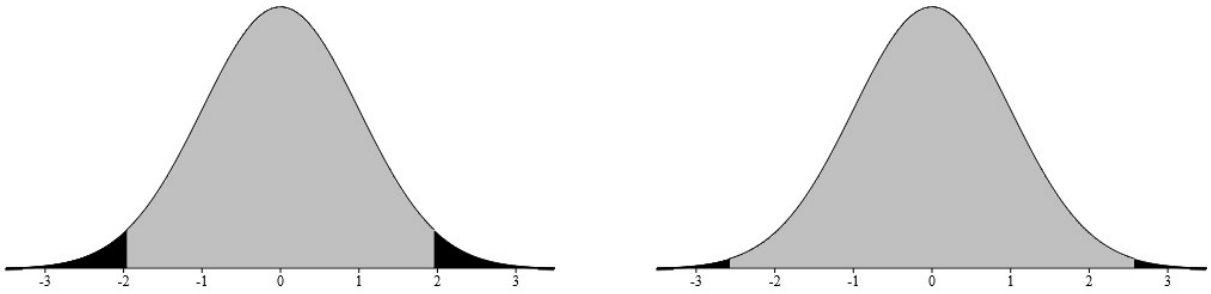
6.3 Confidence interval

From a statistical perspective, the average, $\overline{X}$, described above is actually an estimate of the true population mean, μ. Some expression of confidence that μ lies within a margin of error about $\overline{X}$ is required. The margin of error about $\overline{X}$ can be expressed as $\overline{X} \pm E$, where E is the confidence interval. The question becomes what confidence interval assures μ lies within $\overline{X} \pm E$ with probability p. This is a well-known problem with solution:

$$\overline{X} \pm \frac{Z_p \sigma}{\sqrt{n}}$$

,

Where Z_p is the Z-value corresponding to probability p, σ is the population standard deviation, and n is the sample size of $\overline{X}$. When σ is not known, the sample standard deviation, s, is used.



(a) p=0.95 (b) p=0.99

Figure 6.2. Area under Z PDF for p=0.95 and p=0.99

Two common Z_p are $Z_{0.95} = 1.96$ and $Z_{0.99} = 2.576$. Figure 6.2 shows the excluded areas under the Probability Distribution Function (PDF) in black. The areas in gray are the normalized 0.95% and 0.99% intervals. The term $\frac{\sigma}{\sqrt{n}}$ can be viewed as converting the normalized interval to the observed distribution's interval with sample size n and standard deviation s.

This dissertation uses $Z_{0.95}$ and estimates σ with s. $\overline{X}$ is computed using 100 samples.

Thus, the confidence interval for $\overline{X}$ becomes:

$$E = \frac{1.96s}{\sqrt{100}} = \frac{1.96s}{10} = 0.196s$$

The concise expression of the confidence interval is: 95% of the time, the population mean μ is within the interval:

$$\overline{X} \pm \frac{0.196s}{10} \tag{6.1}$$

It is clear from Equation 6.1 that reducing s and/or increasing n tightens the confidence interval. Increasing n increases run times, which can be expensive for large datasets. Using one iteration to warm-up the GPU can reduce s at very little cost. Thus, 1 iteration is dedicated to warm-up before doing the 100 samples.

For our computations, the confidence interval in Equation 6.1 is very small. Graphically they amount to little more than a thin line immediately above and below the computed means. Because they contribute so little to the graphical presentation, they are shown only for Figure 7.2 on page 42 and dropped thereafter.

While not shown graphically, the confidence intervals have an effect on the conduct of the experiments as detailed in the next section.

6.3.1 Confidence interval considerations

The tighter the confidence interval about $\overline{X}$, the better the probability that $\overline{X}$ is closer to the true mean μ. Examining Equation 6.1 yields the three ways the confidence interval can be tightened:

1. **Use a less stringent Z_p**: For example the interval about the mean in Figure is 6.2a is tighter about the mean than for Figure 6.2b. However, changing Z_p is ultimately unproductive from the perspective of achieving a tighter confidence interval where μ is in that interval with some fixed probability. (A reduced Z_p simply trades off a narrower

confidence interval for an increase in the probability of an error.)

2. **Increasing n**: As n increases, so does its square root, making the confidence interval smaller. Because the square root diminishes the effect of an increasing n, it is not a good first choice. The additional samples increase benchmark runtime.

3. **Reducing s**: There are several ways to control s. One easy trick is to ensure the GPU is not working on any other problem (e.g., the Graphical User Interface (GUI)). Another technique is to avoid code constructions that intrinsically vary in execution time. For instance, if threads compete for an atomic variable using a CAS, only one thread at a time will succeed and force the others to retry. There is no accurate way to predict how long it will take to succeed.

   The experiments performed in this dissertation quiesce the GUI, and only one CAS is performed, but it is used in a way that does not involving any retrying.

6.4 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's Fiji), and a low-end integrated GPU with shared CPU-GPU memory (AMD's 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. This resulted in 2,048 persistent threads (32 workgroups of 64 threads) on the Spectre, and either

1. 14,336 persistent threads (224 workgroups of 64 threads) on the Fiji for scaling experiments, or

2. 8,192 persistent threads (128 workgroups of 64 threads) for optimal queue method experiments.

.

CHAPTER 7

ANALYSIS OF PROPOSED QUEUE

This chapter analyzes the proposed queue from three perspectives:

1. Optimal queuing method (direct or proxy) and chunk size,
2. Scalability, and
3. Effect of the arbitrary-n and retry-free properties on performance.

7.1 Optimal queuing method and chunk size

For this analysis, one experiment was performed for each of the six selected datasets. Each experiment varies the chunk size from 1 to 8 for each of the 4 possible queuing methods. To obtain a data point, for each configuration BFS was run 100 times and the results averaged. The smaller Spectre GPU was configured for 32 workgroups of 64 threads, for a total of 2,048 persistent threads. The larger Fiji GPU was configured for 128 workgroups of 64 threads, for a total of 8,192 threads.

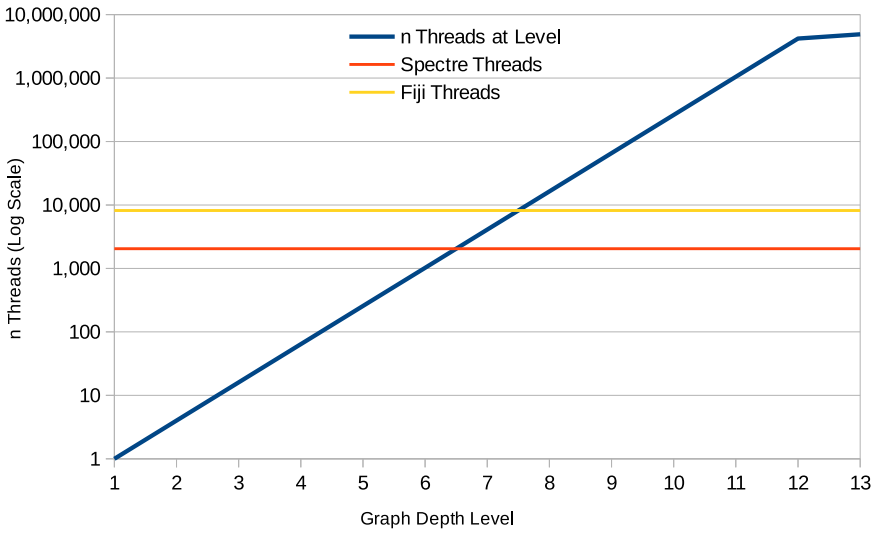


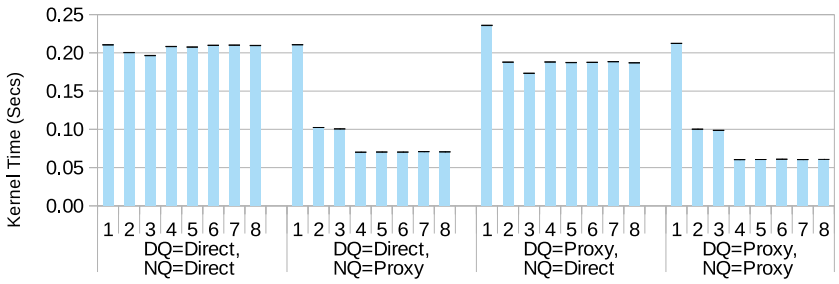
Figure 7.1. Synthetic graph dependency clearance by depth level.

7.1.1 Synthetic dataset optimal queuing method and chunk size

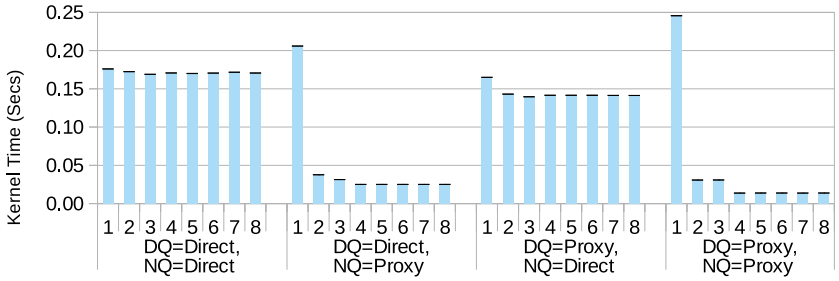
Figure 7.1 shows the number of available nodes at the start of each depth level. The number of nodes available is the direct result of the dependencies cleared at each level. The choice of 4 edges per node causes a rapid expansion in available nodes and heavily favors a chunk size of of 4, which is experimentally verified. The first 7 levels each finish in one work cycle. By level 8, there are more nodes available than there are persistent threads for both the Spectre and Fiji GPUs. After level 8, the number of nodes continues to exponentially increase resulting in massively more available work than persistent threads.

Figure 7.2 shows the performance of queuing algorithm by chunk size for the Spectre (Figure 7.2a) and the Fiji (Figure 7.2b) devices. For the test dataset, the proxy enqueue/dequeue queuing algorithm is the best for both devices. Our observation is as follows:

- The confidence intervals are shown at the top of each bar. The intervals were very tight and appear only as a dark mark at the top of each bar.
- Generally the Fiji outperformed the Spectre except for the proxy enqueue/dequeue queuing algorithm with a chunk size of 1.



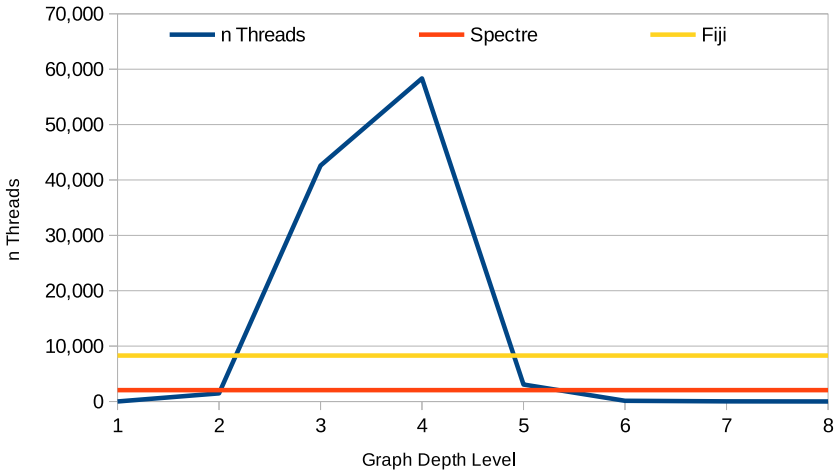
(a) Spectre.



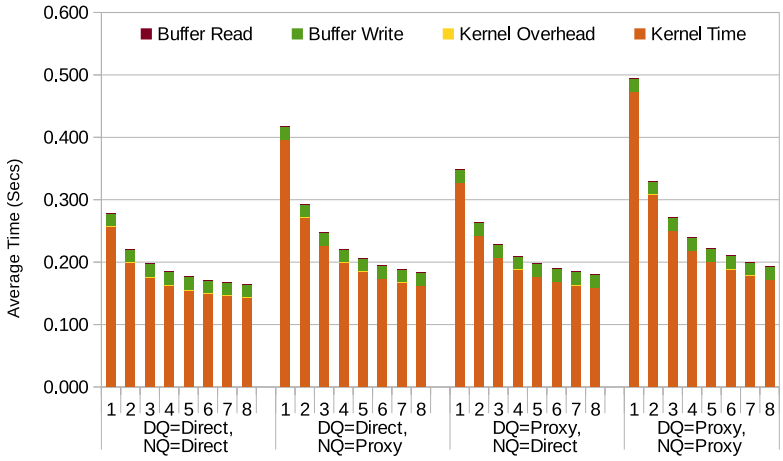
(b) Fiji.

Figure 7.2. BFS kernel execution time by device/queuing algorithm/chunk size (synthetic data).

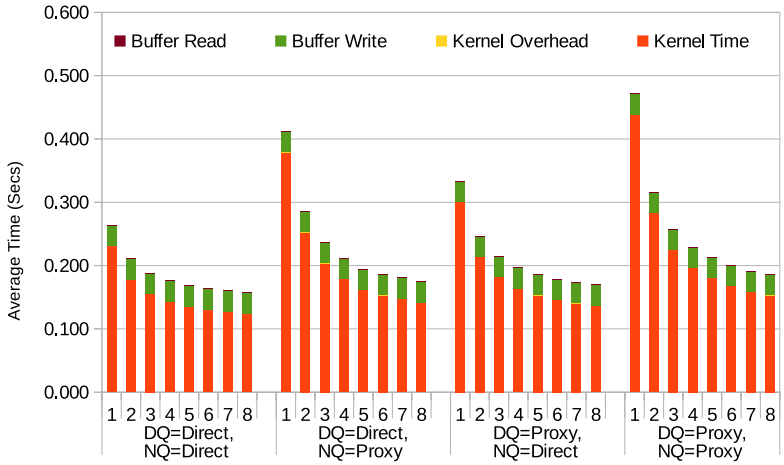
- Once the chunk size reaches 4, execution time does not change. This is an artifact of choosing 4 edges per node for the test data and is not generally true for all workloads.
- For both devices, the proxy enqueue/dequeue queuing algorithm with a chunk size of 4 is best.



(a) Dependency clearance by level.



(b) Spectre execution times.



(c) Fiji execution times.

Figure 7.3. gplus_combined analysis.

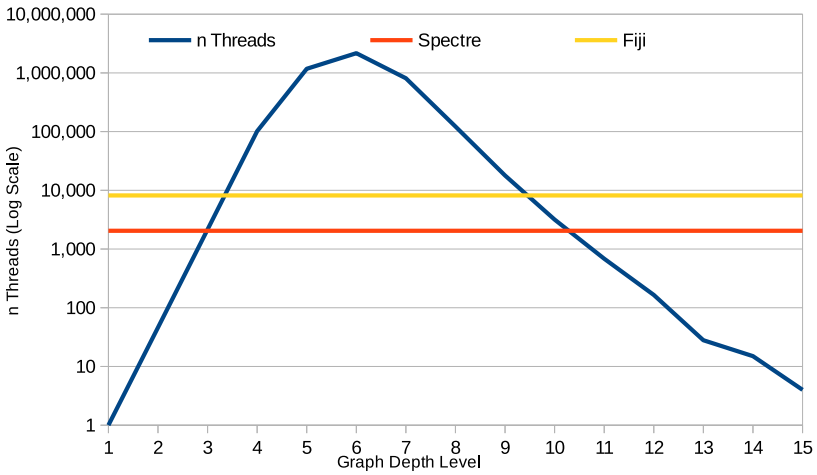
7.1.2 gplus_combined dataset optimal queuing method and chunk size

This dataset consists of "circles" from Google+. The Google+ data was collected from users who had manually shared their circles using the "share circle" feature. Figure 7.3 gives the analysis for this dataset.

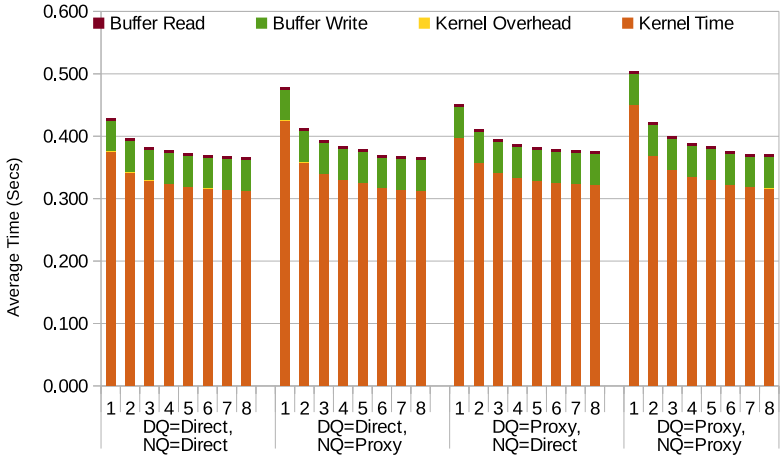
Figure 7.3a shows the number of threads cleared at each dependency level. It requires 8 levels to process. Figure 7.3b breaks down the execution times by queuing algorithm and chunk size for the Spectre GPU. Figure 7.3c breaks down the execution times by queuing algorithm and chunk size for the Fiji GPU.

Both the Spectre's 2,048 threads, and the Fiji's 8,192 threads are significantly saturated. Ordinarily this would favor proxy access. However, chunking reduces dequeue contention. Further, refer to Table 6.1 on page 34 and notice there are over 100 times more edges than nodes. Thus it is likely a node can be discovered via many edges. Only the first discovery results in an enqueue. The overall effect is reduced contention, marginally favoring direct enqueue/dequeue.

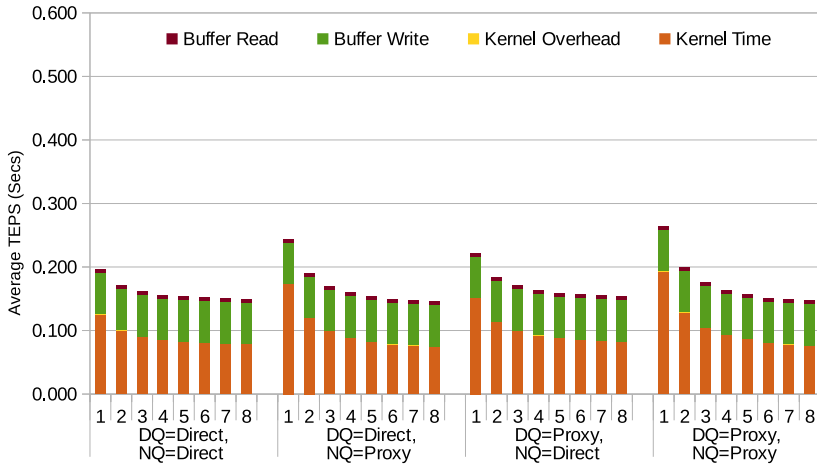
For this dataset, increasing the chunk size will further reduce dequeue contention, but it is a diminishing return. Since the number of idle threads increases with chunk size, this quickly overcomes the queue overhead savings of an increased chunk size. The large number of threads favors the Fiji, but it is only marginally faster than Spectre because there are so many non-productive edges (edges that don't discover a new work).



(a) Dependency clearance by level.



(b) Spectre execution times.



(c) Fiji execution times.

Figure 7.4. soc-LiveJournal1 analysis.

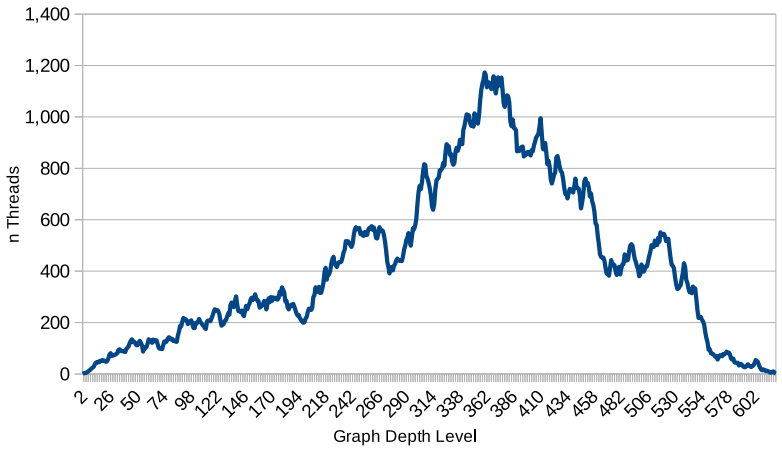
7.1.3 soc-LiveJournal1 dataset optimal queuing method and chunk size

LiveJournal is a free on-line community with almost 10 million members. A significant fraction of these members are highly active (For example, roughly 300,000 update their content in any given 24-hour period). LiveJournal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong. Figure 7.4 shows the analysis of this dataset.

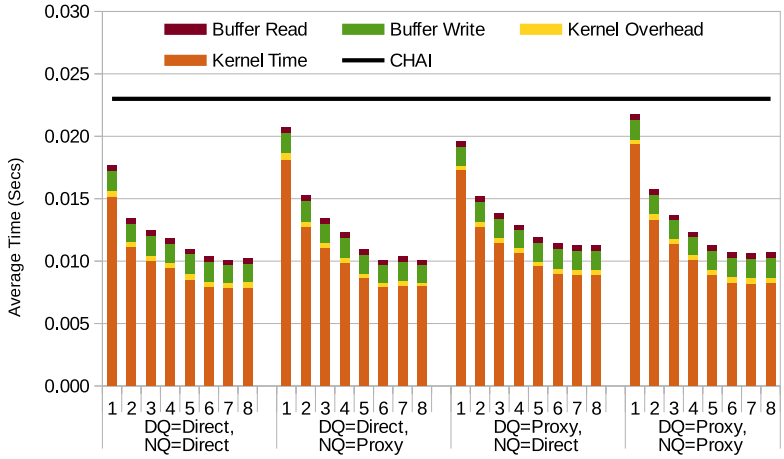
Figure 7.4a shows the number of threads cleared at each dependency level. This dataset requires 15 levels to process. Both GPUs are massively saturated, which strongly favors the Fiji GPU. This is clearly visible in the graphs. Figure 7.4b breaks down the execution times by queuing algorithm and chunk size for the Spectre GPU. Figure 7.4c breaks down the execution times by queuing algorithm and chunk size for the Fiji GPU.

Chunking reduces dequeue contention enough to allow direct dequeues. Refer to Table 6.1 on page 34 and notice there are many more nodes than the gplus_combined dataset but a smaller edge fan-out. The smaller fan-out does not sufficiently mitigate the enqueue contention, and requires proxy enqueue.

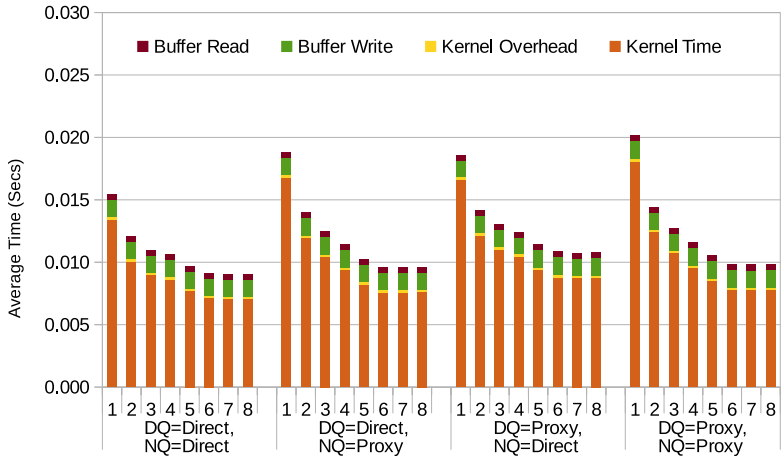
The direct dequeue and the proxy enqueue with a chunk size of 8 produced the best results. Note the proxy dequeue and enqueue method with a chunk size of 8 produced competitive results.



(a) Dependency clearance by level.



(b) Spectre execution times.



(c) Fiji execution times.

Figure 7.5. USA-road-d.NY analysis.

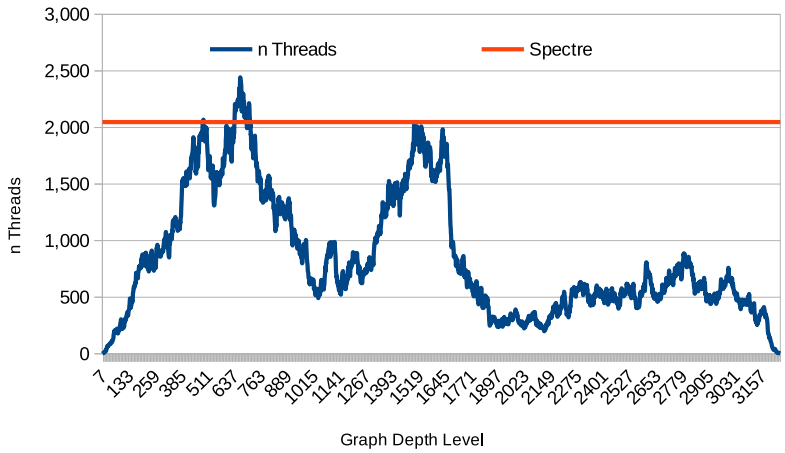
7.1.4 USA-road-d.NY dataset optimal queuing method and chunk size

This dataset is a representation of the New York City road grid. Figure 7.5 gives the analysis of that dataset.

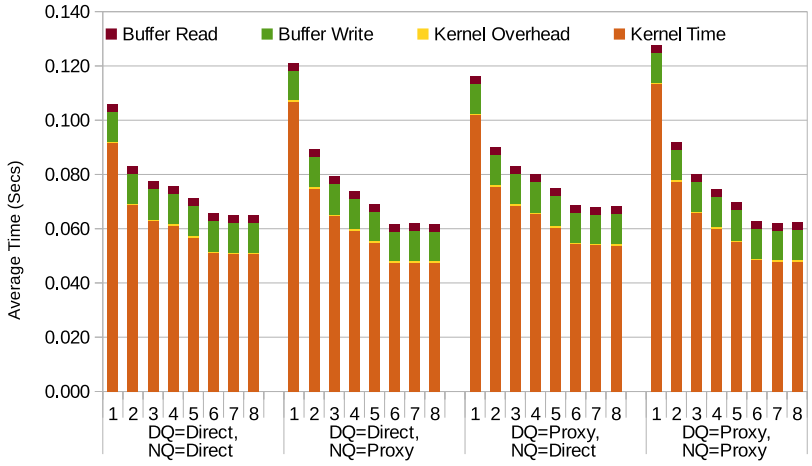
Figure 7.5a shows the number of threads cleared at each dependency level. This dataset requires 620 levels to process. Figure 7.5b breaks down the execution times by queuing algorithm and chunk size for the Spectre GPU. Figure 7.5c breaks down the execution times by queuing algorithm and chunk size for the Fiji GPU.

Neither the Spectre (2,048 threads) nor the Fiji (8,192 threads) is ever fully saturated. This means there are always more persistent threads than available work. Thus, only a relatively small number of threads are active at any given time. This lessens thread contention and favors direct access to the queue on both GPUs.

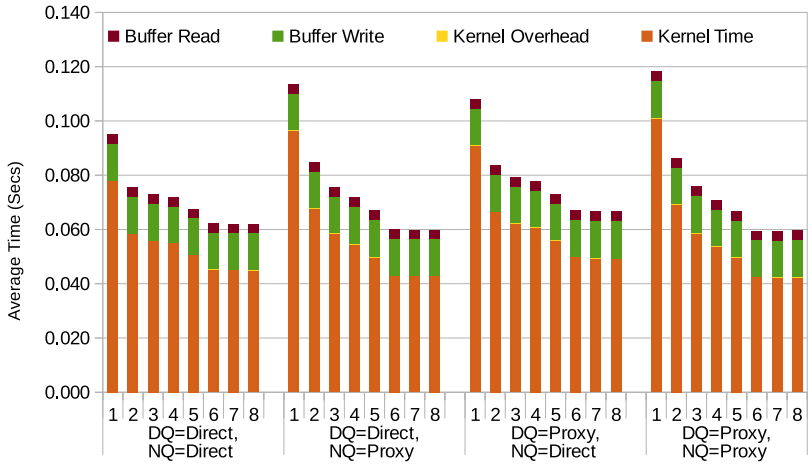
For both GPUs, the direct enqueue/dequeue method with a chunk size of 7 produced the best results.



(a) Dependency Clearance by Level



(b) Spectre Execution Times



(c) Fiji Execution Times

Figure 7.6. USA-road-d.LKS analysis.

7.1.5 USA-road-d.LKS dataset optimal queuing method and chunk size

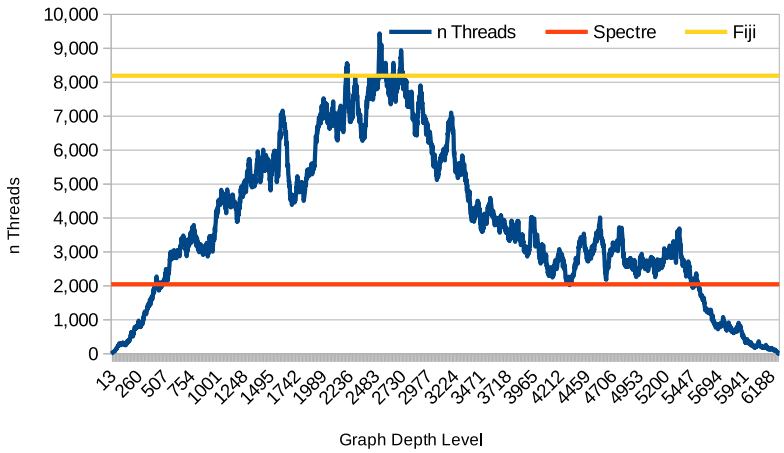
This dataset is a representation of the Great Lakes area road grid. Figure 7.6 gives the analysis of that dataset.

Figure 7.6a shows the number of threads cleared at each dependency level. This dataset requires 3,241 levels to process. This is significantly more than the USA-road-d.NY dataset because it covers a larger area. Figure 7.6b breaks down the execution times by queuing algorithm and chunk size for the Spectre GPU. Figure 7.6c breaks down the execution times by queuing algorithm and chunk size for the Fiji GPU.

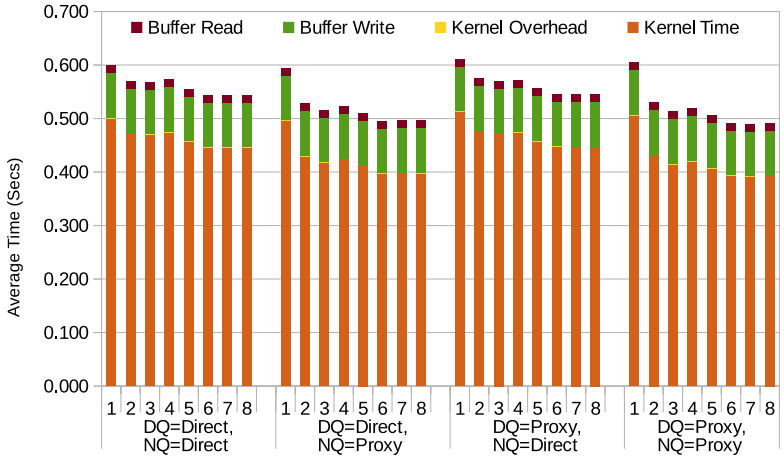
The Spectre's 2,048 threads are briefly saturated (briefly there is more work than persistent threads), but the Fiji's 8,192 threads are never fully saturated (there are always more persistent threads than available work.) In the brief period where the Spectre is saturated, the Fiji has a slight advantage. Since some saturation does occur, this begins to favor proxy queue access. Generally there is more work available than for the USA-road-d.NY dataset, and that helps thread parallelism.

For the Spectre GPU, the direct dequeue method and the proxy enqueue method with a chunk size of 8 produced the best results. For the Fiji GPU, the proxy enqueue/dequeue method with a chunk size of 7 produced the best results.

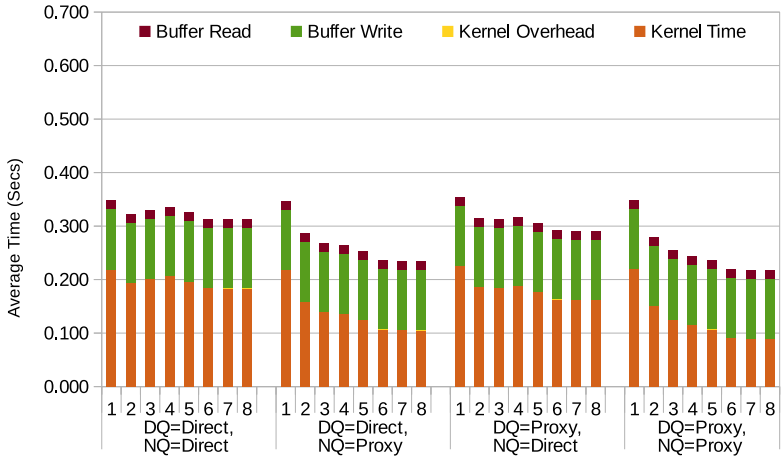
The Fiji execution time is slightly faster because is clocks faster and briefly has more active threads than the Spectre.



(a) Dependency Clearance by Level



(b) Spectre Execution Times



(c) Fiji Execution Times

Figure 7.7. USA-road-d.USA analysis.

7.1.6 USA-road-d.USA dataset optimal queuing method and chunk size

This dataset is a representation of the USA road grid. Figure 7.7 gives the analysis of that dataset.

Figure 7.7a shows the number of threads cleared at each dependency level. This dataset requires 6,262 levels to process. Figure 7.7b breaks down the execution times by queuing algorithm and chunk size for the Spectre GPU. Figure 7.7c breaks down the execution times by queuing algorithm and chunk size for the Fiji GPU.

This is the largest road map dataset. The Spectre's 2,048 threads are significantly saturated (there is significantly more work than persistent threads), and the Fiji's 8,192 threads are briefly fully saturated. Since saturation does occur, this favors proxy queue access. The number of threads is above 2,048 threads for most of the processing levels. This favors the Fiji, and it is significantly faster than the Spectre for this dataset.

For both GPUs, the proxy enqueue/dequeue method which a chunk size of 7 produced the best results.

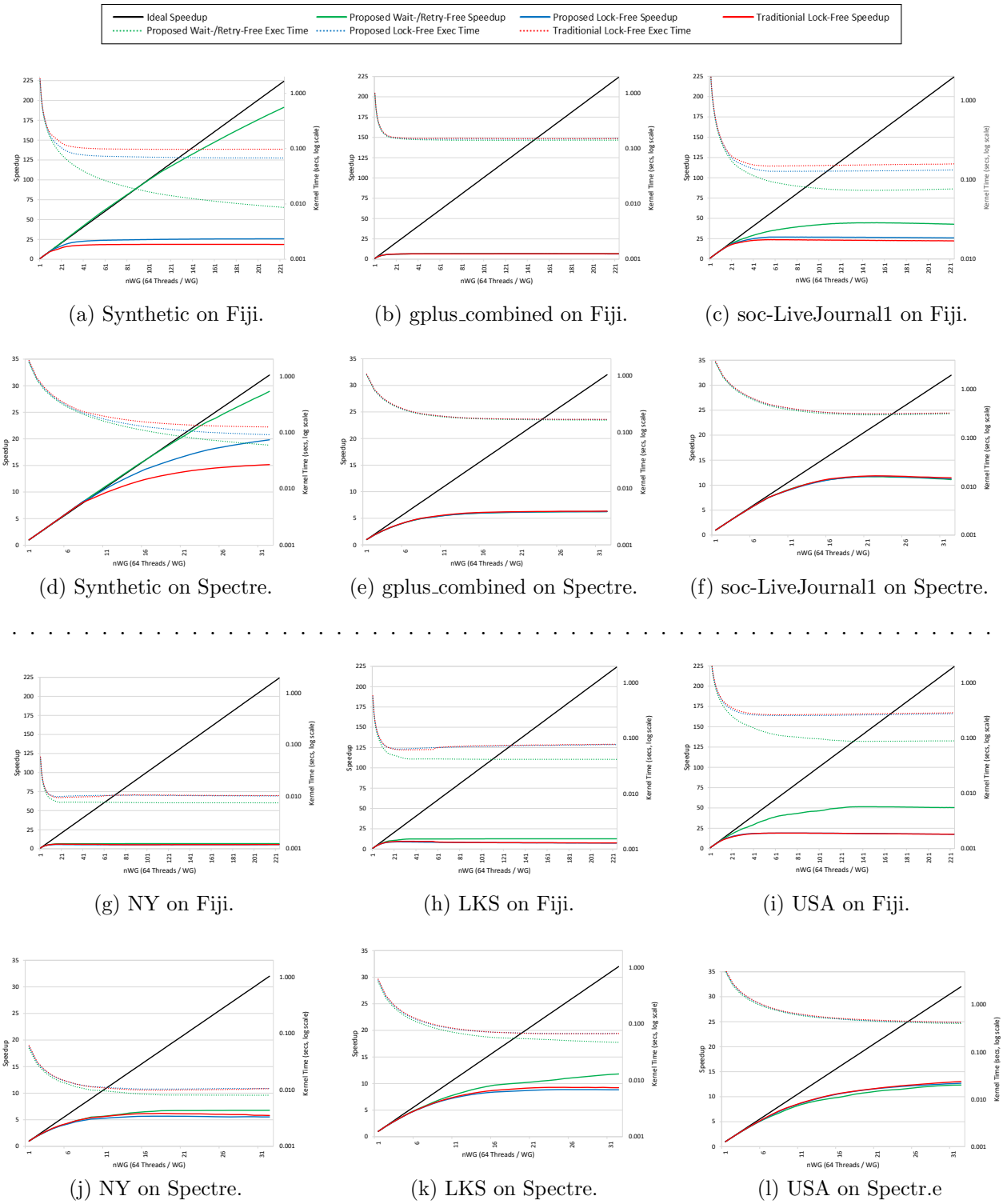
The significant saturation of the Spectre gives the Fiji a noticeable performance advantage, which is clearly visible in the graphs.

7.1.7 Optimal queuing method and chunk size conclusions

When the persistent threads are not saturated, the direct queuing algorithm performs better than the proxy queuing algorithm, but only marginally. The proxy overhead is a fixed cost due to the arbitrary-n property. When saturation occurs the per thread cost of the direct queuing algorithm overhead becomes significant and exceeds the cost proxy method.

The purpose of chunking is to reduce the cost of implementing the persistent thread work cycle. It achieves this by processing multiple chunks each work cycle. However, as the number of chunks per work cycle increase so does the chance individual threads in the wavefront complete their task earlier than others and go idle. The performance benefit of chunking significantly tappers off by a chunk size of 8.

Therefore, the best general guidance when the characteristics of the dataset are not known in advance is to use the proxy queuing algorithm for both enqueues and dequeues, and a chunk of 8.



(a) Synthetic on Fiji. (b) gplus_combined on Fiji. (c) soc-LiveJournal1 on Fiji.

(d) Synthetic on Spectre. (e) gplus_combined on Spectre. (f) soc-LiveJournal1 on Spectre.

(g) NY on Fiji. (h) LKS on Fiji. (i) USA on Fiji.

(j) NY on Spectre. (k) LKS on Spectre. (l) USA on Spectr.e

Figure 7.8. Execution time and speedup.

7.2 Effects of the arbitrary-n and retry-free properties on performance.

Three concurrent queue variations are used to expose the effects of the retry-free and arbitrary-n properties. The queue variants are:

- **BASE**: This is a traditional queue using CAS-based lock-free atomics. 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 atomic failures.
- **WRF/AN**: This is the proposed wait-/retry-free and arbitrary-n concurrent queue.

The difference between the AN and WRF/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.

In this analysis, the chunk size was fixed at 8, and the proxy queuing algorithm was used. Both GPUs were configured for 64 threads per workgroup/wavefront. Both the Spectre and Fiji were configured for 4 workgroups per CU, yielding 2,048 threads (32 workgroups) for the Spectre GPU and 14,336 (224 workgroups) for the Fiji GPU. To obtain a data point, 100 BFS runs were averaged for each selected dataset and queue variant on each GPU varying the number of workgroups from 1 to 32 for the Spectre GPU or 224 for the Fiji GPU. There were no outlier data points.

Figure 7.8 shows the execution time and speedup curves for each queue variant across all selected datasets are presented. The legend is given at the top of the figure. The solid lines show speedup using the scale on the left y-axis. The dotted lines show execution time using the scale on the right y-axis. The ideal speedup is shown in black. Results for the proposed WRF/AN queue (wait-/retry-free queue) are shown in green. Results for the AN queue are shown in blue. Results for the BASE queue (traditional lock-free queue) are shown in red.

The WRF/AN queue outperformed the other variants except for the USA dataset on the Spectre GPU, which had a marginally worse speedup.

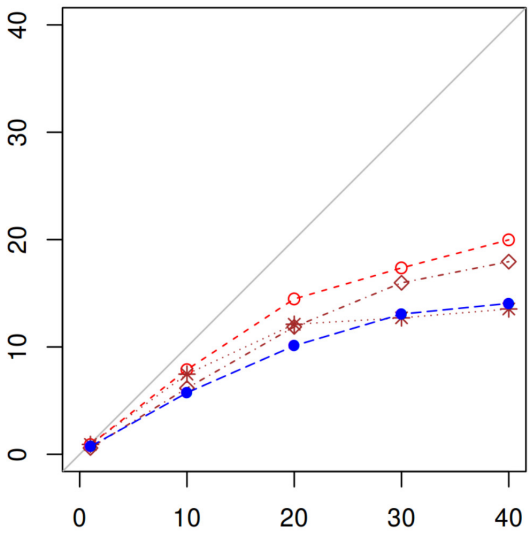
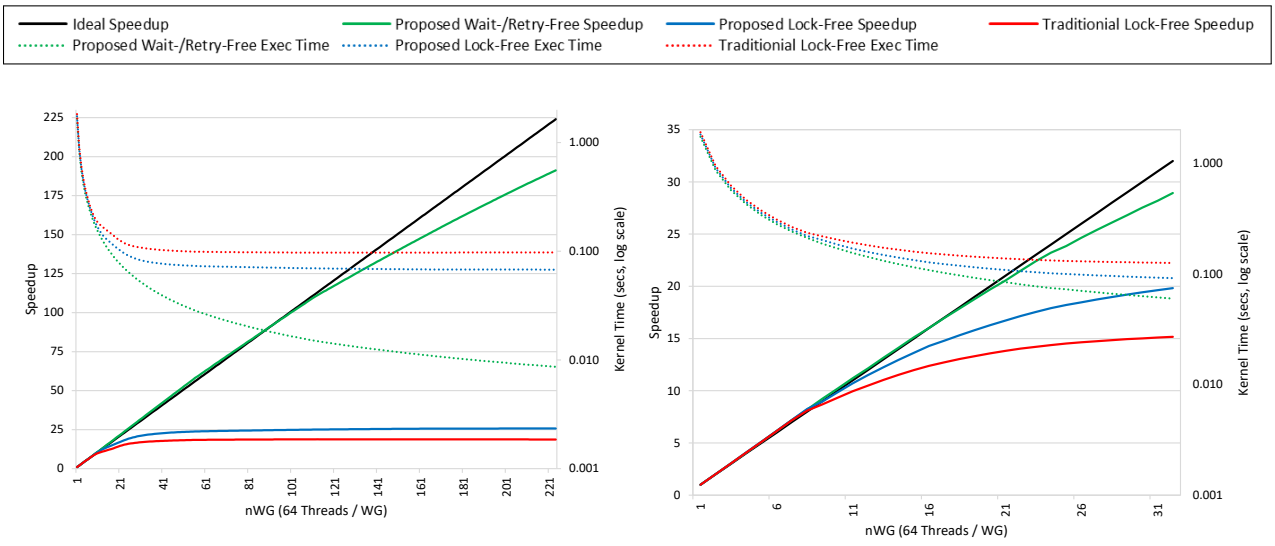


Figure 7.9. Traditional speedup curves.



(a) Fiji scalablility.

(b) Spectre scalablility.

Figure 7.10. Scalability.

7.3 Scalability

The single most important result of this dissertation is the scalability of the proposed queue in a massively parallel GPU environment.

Acar et al. [55] studied the performance of parallel workloads. Figure 7.9 is extracted from their paper and shows the effects of parallelism for four tested applications. The gray line is the ideal speedup. The cumulative effects of parallelism quickly flatten speedup curves. For GPU-based persistent thread applications, retries due to atomic failure and dequeue queue empty conditions are most significant factors.

There are three factors that affect scalability:

1. **Lack of work:** When a dataset lacks sufficient parallelism, there may not be enough tasks for all persistent threads. Those threads encountering a queue empty failure when they attempt to dequeue a task token idle for the work cycle. They do not accelerate and impact scalability.

2. **Atomic retries:** When atomic operations fail they must retry in the next work cycle. The work cycle encountering the failure does accelerate and impacts scalability.

3. **Other overhead:** This is a catch-all category to account for all other overhead.

The synthetic dataset is used to measure scalability. It massively saturates all threads and thus removes lack of work as a factor affecting scalability. Figure 7.10 shows the scalability of the synthetic dataset. The legend is given at the top of the figure. The solid lines show speedup using the scale on the left y-axis. The dotted lines show execution time using the scale on the right y-axis. The ideal speedup is shown in black. Results for the proposed WRF/AN queue (wait-/retry-free queue) are shown in green. Results for the AN queue are shown in blue. Results for the BASE queue (traditional lock-free queue) are shown in red.

The proposed queue scales with 10% of the ideal speedup with 14,336 active threads. Loss of either the retry-free property (AN) or arbitrary-n properties results in significant degradation of scalability. Loss of both (BASE) results in typical parallel application scalability.

7.4 BFS performance comparison

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

1. **CHAI** [29]: 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 GPU only.

2. **Rodinia** [7]: 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 load on the GPU.

7.4.1 Comparison to the CHAI BFS benchmark

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

Table 7.1. Performance comparison with CHAI BFS (ms).

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 heterogeneous kernel uses 8 GPU CUs, and 2 CPU CUs. Their test datasets are relatively small road map graphs with only modest dynamic parallelism. Table 7.1 details the kernel times for CHAI and the proposed queue (WRF/AN). All times are in milliseconds. Our proposed algorithm outperforms CHAI BFS by at least 2.57 times.

7.4.2 Comparison to the Rodinia BFS benchmark

Dataset	Device	Rodinia	WRF/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 7.2. Performance comparison with Rodinia BFS (ms).

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 Rodinia tests were run on both the Spectre and Fiji GPUs. Table 7.2 compares the kernel times for Rodinia and the proposed queue (WRF/AN).

Our analysis shows that the proposed queue outperforms both the compared BFS implementations consistently because:

- The proposed queue does not suffer retries due to exception conditions (i.e., queue empty or queue full).
- In the proposed queue excess threads dequeue only once, whereas the other implementations a retry dequeues each work cycle when they encounter a queue empty exception.

- The proposed queue atomic operations are wait-free and retry-free, whereas the other implementations use CAS atomics and must retry on each CAS failure.

CHAPTER 8

SSSP GPU SPECULATE AND CORRECT ALGORITHM

8.1 Motivation

Refer to Figure 7.8 on page 54, which details the speedup for all selected datasets and queue variants. Only the proposed queue for the synthetic dataset on both GPUs scale well (Figure 7.8a and Figure 7.8d). The other two variants do not scale because they suffer retry overhead. None of the other five datasets scale well because, to varying extents, those datasets have insufficient parallelism to saturate the persistent threads.

Thus, if sufficient work is available, the proposed queue will scale well as threads are added because there is no retry overhead. Thread saturation becomes a primary design motivation.

In some cases irregular work loads offer an opportunity. An irregular workload can saturate if its data dependencies can be safely ignored. Ignoring data dependencies can cause errors. If an algorithm can detect and correct errors caused by ignoring data dependencies, then the GPU can be saturated.

The basic mechanism is that rather than honor a data dependency, the algorithm speculates. Incorrect speculations are eventually detected and corrected. Corrections become a new form of overhead, which are analyzed later in this chapter. This dissertation refers to such algorithms as *speculate and correct*. This chapter develops a GPU speculate and correct SSSP algorithm.

Algorithm 5 Classic Bellman-Ford SSSP.

```
1: procedure BELLMAN-FORD(V, E, w, s)
2:     for each vertex v ∈ V do
3:         v.d ← ∞
4:         v.π ← NIL
5:     end for
6:     s.d ← 0
7:     for i ← 0 to |V| − 1 do
8:         for each edge (u,v)∈ E do
9:             if v.d > u.d + w(u,v) then
10:                v.d = u.d + w(u,v)
11:                v.π = u
12:            end if
13:        end for
14:    end for
15:    for each edge (u,v)∈ E do
16:        if v.d > u.d + w(u,v) then
17:            return FALSE
18:        end if
19:    end for
20:    return TRUE
21: end procedure
```

8.2 Bellman-Ford SSSP algorithm

The Bellman-Ford SSSP algorithm (Cormen et al. [10, pp 643–683]) is an example of an algorithm that is inherently a speculate and correct algorithm. Algorithm 5 gives the classic Bellman-Ford SSSP algorithm. Bellman-Ford works by traversing edges at most $|V| - 1$ times (lines 7-14). There is a dependency that requires one pass to complete before the next pass can begin. Within a pass, edges can be processed in any order.

In each pass, relaxation occurs if the traversed edge offers a lower cost path to a vertex than currently exists (lines 9-12). One perspective on this process is that a prior pass incorrectly speculated on the least cost path to a vertex and it is being corrected in the current pass (The correction is also a speculation that could be corrected in a subsequent pass).

Bellman-Ford traverses the edges one final time (lines 15-19). If, after $|V| - 1$ passes, relaxations still occur, then a negative weight cycle exists and it returns false (line 17). Otherwise, it return true (line 20). Thus, Bellman-Ford allows negative edge weights and detects any negative weight cycles.

There is a significant inefficiency in the Bellman-Ford algorithm. If an edge relaxes a vertex cost, observe that only the descendants of the relaxed vertex need be corrected. A full pass is not required. Further, multiple passes are not required. SSSP can be performed in a single pass if descendant path errors are corrected before speculation continues. If corrections are performed in parallel, the GPU is saturated either by speculation or correction.

8.3 Proposed GPU speculate and correct SSSP algorithm

The proposed GPU speculate and correct SSSP algorithm performs a single traversal of the vertices in a width-first manner. This is done by visiting the children of a node using a speculation queue. The speculation frontier grows exponentially and quickly saturates all threads. When relaxation occurs, the vertex is queued for a re-visit in a higher priority correction queue. This corrects the descendents of a corrected vertex. Both the speculate and correct queues are implemented using the proposed queue. Since correction occurs before speculation, corrections impede the speculation process, which helps minimize propagating the effects of an incorrect speculation.

To duplicate all the properties of the Bellman-Ford algorithm, the proposed algorithm must detect negative weight cycles. Observe that relaxation is triggered by an edge with a lower path cost to a vertex. Thus, the maximum number of non-cyclic relaxations on a given vertex is bounded by the number of edges. When a vertex undergoes relaxation more than $|E| - 1$ times, a edge has been reused meaning a cycle exists. Further, the cycle must be a negative weight cycle because relaxation occurred. The proposed algorithm detects loops in this manner.

The proposed GPU speculate and correct Bellman-Ford algorithm variant is the second

most important contribution made in this dissertation.

8.3.1 Canonical solution

In a multi-threaded SSSP algorithm, the check for a shorter path and updating to a less costly path and noting the new parent must be done in a single atomic operation to avoid race conditions. In the proposed SSSP algorithm, relaxation is performed by a 64-bit atomic_fetch_min. The high-order 32-bits is the path cost, and the low-order 32-bits is the parent for that path cost. An atomic_fetch_min is a wait-free alternative to a critical section. It avoids the retries associated with failing to obtain the lock on the critical section.

There may be several equal-cost paths to a vertex, which results in a non-unique solution. In the proposed SSSP algorithm, the parent vertex index participates in the cost check as the low order 32-bits. Thus, only one parent vertex index will be selected for any given path, and makes solutions using the proposed SSSP algorithm unique.

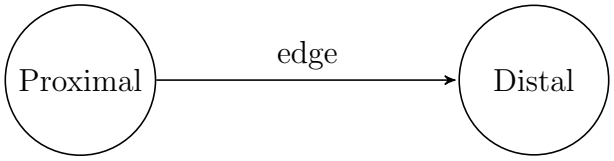


Figure 8.1. Vertex/Edge terminology.

8.3.2 Details of the proposed GPU speculate and correct SSSP algorithm

The proposed speculate and correct SSSP algorithm uses the chunked persistent thread algorithm. Figure 8.1 shows the terminology used to describe vertices relative to an edge. See Algorithm 4 on page 30. Two proposed queues are used – one for speculation and one for correction, with the correction queue having higher priority. The correction queue is implemented as a circular queue. Appendix F on page 110 gives the full source of the proposed SSSP kernel. The salient sections are detailed below:

Listing 8.1. Wait-free, retry-free, arbitrary-n SSSP dequeue.

```
1   // ************************************************************************************************
2   // Step 1: Dequeue
3   // Hungry threads are assigned a queue slot. Work may or may not have arrived for that slot.
4   // ************************************************************************************************
5
6   // Get base index of the slots for hungry threads.
7   if (IsProxyThread)
8   {
9     lnSpecQueueSlotsNeeded =
10      lnCorrQueueSlotsNeeded = 0u;
11    lnThreadsEndingThisCycle = 0u;
12  }
13
14  if (ThreadNeedsSpecWork)
15  {
16    // Count all threads and assign each thread it's relative slot index;
17    DequeueThreadSpecSlotIndex = atomic_inc(&lnSpecQueueSlotsNeeded);
18  }
19
20  if (ThreadNeedsCorrWork)
21  {
22    // Count all threads and assign each thread it's relative slot index;
23    DequeueThreadCorrSlotIndex = atomic_inc(&lnCorrQueueSlotsNeeded);
24  }
25
26  // Get base index of the slots for hungry threads.
27  if (IsProxyThread)
28  {
29    lQueueSlotSpecBaseIndex = atomic_add(&Parms->SpecQueueFront, lnSpecQueueSlotsNeeded);
30    lQueueSlotCorrBaseIndex = atomic_add(&Parms->CorrQueueFront, lnCorrQueueSlotsNeeded);
31  }
32
33  if (ThreadNeedsSpecWork)
34  {
35    DequeueThreadSpecSlotIndex += lQueueSlotSpecBaseIndex;
36    ThreadNeedsSpecWork = false;
37  }
38
39  if (ThreadNeedsCorrWork)
40  {
41    DequeueThreadCorrSlotIndex += lQueueSlotCorrBaseIndex;
42    ThreadNeedsCorrWork = false;
43  }
```

8.3.3 Dequeuing details

Listing 8.1 details the proposed SSSP dequeue process:

1. **Lines 7-12:** Initialize the number of hungry threads needing speculate and/or correct slots. It also flags that no threads have (yet) ended this cycle. This is done by the proxy thread.

2. **Lines 14-18:** Each thread checks to see if it needs a speculate slot index. If so, it atomically increments *lnSpecQueueSlotsNeeded*; *DequeueThreadSpecSlotIndex* remembers the wavefront-relative slot index. It is later to converted to an absolute index. Initially, all threads are hungry.

3. **Lines 20-24:** Each thread checks to see if it needs a speculate slot index. If so, it

atomically increments *lnCorrQueueSlotsNeeded*; *DequeueThreadCorrSlotIndex* remembers the wavefront-relative slot index. It is later to converted to an absolute index. Initially, all threads are hungry.

4. **Lines 27-31:** The proxy thread allocates the required number of slots. Line 29 allocates speculate queue slots and stores the base in *lQueueSlotSpecBaseIndex*. Line 30 allocates correction queue slots and stores the base in *lQueueSlotCorrBaseIndex*.

5. **Lines 33-37:** If a thread was hungry for a speculate queue slot index, this converts its wavefront-relative slot index to an actual speculate queue slot index.

6. **Lines 39-43:** If a thread was hungry for a correction queue slot index, this converts its wavefront-relative slot index to an actual correction queue slot index.

Listing 8.2. Wait-free, retry-free, arbitrary-n SSSP data arrival.

```
1   // ************************************************************************************************
2   // Step 2: Data Arrrival and Prolog
3   // ************************************************************************************************
4   if (!QueueDataAvailable)
5   {
6     // Check to see if data has arrived. Correction queue is higher priority than speculate queue
7     ProximalNodeIndex = Missing;
8     if ((DequeueThreadCorrSlotIndex < Parms->CorrQueueRear) && (QueueDataAvailable = (CorrQueue[DequeueThreadCorrSlotIndex % CorrQueueSize]
          != Missing)))
9     {
10      // Work as arrived in spec queue. Setup to process this node.
11      // No atomics are needed because this is the only thread accessing the slot or node.
12
13      // Get work token (index of node to process).
14      ProximalNodeIndex = CorrQueue[DequeueThreadCorrSlotIndex % CorrQueueSize];
15
16      // Correction queue may reuse this slot. Set it to missing.
17      CorrQueue[DequeueThreadCorrSlotIndex % CorrQueueSize] = Missing;
18      ThreadNeedsCorrWork = true;
19    } else
20    if ((DequeueThreadSpecSlotIndex < Parms->SpecQueueRear) && (QueueDataAvailable = (SpecQueue[DequeueThreadSpecSlotIndex] != Missing)))
21    {
22      // Work as arrived in spec queue. Setup to process this node.
23      // No atomics are needed because this is the only thread accessing the slot or node.
24
25      // Get work token (index of node to process).
26      ProximalNodeIndex = SpecQueue[DequeueThreadSpecSlotIndex];
27      ThreadNeedsSpecWork = true;
28    }
29
30    if (ProximalNodeIndex != Missing)
31    {
32      // Get assigned node.
33      ProximalNode = Nodes[ProximalNodeIndex];
34
35      // Get number of nodes to process.
36      nEdgesLeft = ProximalNode.nEdges;
37
38      // Get starting edge for this node.
39      CurrentEdge = Edges + ProximalNode.StartingEdgeIndex;
40
41      // Get proximal node cost;
42      ProximalNodeCost = GetCost(CostsParents[ProximalNodeIndex]);
43    }
44  }
```

8.3.4 Data arrival details

The correction queue has higher priority than the specuate queue. So, the correction queue is checked before the speculate queue. Listing 8.2 details the proposed SSSP data arrival and prolog processes:

1. **Lines 4-44:** *QueueDataAvailable* is false if data has not yet arrived in either queue. When it is false, the correction queue is checked for data arrival first. If no data has arrived there, then the speculate queue is checked.

2. **Lines 8-19:** When data arrives from the correction queue, *ProximalNodeIndex* is set from the correction queue, the correction queue slot is set to the data-not-arrived sentinel, *QueueDataAvailable* is set to true, and the thread is flagged as needing another correction queue slot.

3. **Lines 20-28:** If data did not arrive in the correction queue, these lines check the speculate queue. If data arrives, *ProximalNodeIndex* is set from the speculate queue, the speculate queue slot is set to the data-not-arrived sentinel, *QueueDataAvailable* is set to true, and the thread is flagged as needing another speculate queue slot.

4. **Lines 30-43:** These lines correspond to the prolog on line 13 of Algorithm 4. They are executed once for each data arrival and setup chunk processing.

Listing 8.3. Wait-free, retry-free, arbitrary-n SSSP chunk processing.

```
1
2
3     // ************************************************************************************************
4     // Step 3: Do up to __ChunkSize__ chunks
5     // ************************************************************************************************
6
7     if (QueueDataAvailable)
8     {
9       // Process one chunk.
10      for (uint32_t Chunk = 0u; (nEdgesLeft > 0u) && (Chunk < __ChunkSize__); ++Chunk)
11      {
12        // For an arbtrary graph, the edge can point to a node that has already been assigned a cost,
13        // or at the current level, two nodes can concurrently access a node at the next level.
14        // When that happens, this atomic selects a winner thread that will assign the cost.
15        uint32_t DistalNodeIndex = CurrentEdge->DistalNodeIndex;
16
17        // Ignore reverse edge in undirected graphs.
18        // Ensure distal node index is not my parent.
19        if (DistalNodeIndex != GetParent(CostsParents[ProximalNodeIndex]))
20        {
21          // The distal node does not point back to proximal node.
22          uint32_t NewDistalCost = ProximalNodeCost + CurrentEdge->Weight;
23          uint64_t NewDistalCostParent = MakeCostParent(NewDistalCost, ProximalNodeIndex);
24          uint64_t OldDistalCostParent = atom_min(CostsParents + DistalNodeIndex, NewDistalCostParent);
25
26          uint32_t OldDistalCost = GetCost(OldDistalCostParent);
27          if (OldDistalCost != NewDistalCost)
28          {
29            // Cost has improved. We need to do a relaxation.
30            if (atomic_inc(RelaxationCount + DistalNodeIndex) >= Parms->nEdges)
31            {
32              // Vertrex relaxed too many times. We must have a negative loop.
33              atomic_store((volatile __global atomic_uint *) &Parms->AbortCode, 1u);
34            }
35
36            // Check where to queue the new work.
37            if (OldDistalCost == Missing)
38            {
39              // Queue this node as new work for speculate queue
40              NewlyDiscoveredSpecWork[nNewlyDiscoveredSpecWork++] = DistalNodeIndex;
41            }
42            else
43            if (NewDistalCost < OldDistalCost)
44            {
45              // We have found a better cost and need to correct.
46              NewlyDiscoveredCorrWork[nNewlyDiscoveredCorrWork++] = DistalNodeIndex;
47            }
48          }
49        }
50
51        // Move to next edge.
52        ++CurrentEdge;
53
54        // Finished this edge, count it.
55        --nEdgesLeft;
56      }
57
58      // If all edges processed, show thread is hungry.
59      if (nEdgesLeft == 0u)
60      {
61        atomic_inc(&lnThreadsEndingThisCycle);
62        QueueDataAvailable = false;
63      }
64    }
```

8.3.5 Chunk processing details

The correction queue has higher priority than the specuate queue. So, the correction queue is checked before the speculate queue. Listing 8.3 details the proposed SSSP chunk processing:

1. **Lines 7-64:** If data has arrived, these lines process up to *__ChunkSize__* chunks.

2. **Lines 10-56:** These lines form the chunk processing loop.

3. **Lines 19-49:** SSSP processes directed graphs. If an undirected graphs is given, this code ensures reverse edges are not processed.

4. **Line 24:** This lines atomically checks for a better path. It uses a 64-bit atomic_fetch_add to check and update the cost and parent if a better path is found.

5. **Lines 27-48:** These lines handle relaxation details.

6. **Lines 30-34:** These lines check for a negative-weight loop, and atomically set the abort flag is there is a loop.

7. **Lines 37-41:** If this is the first time a vertex is relaxed, then its children can be speculated. The child vertex is marked for speculation. Once the correction process encounters first time relaxation, correction ceases and speculation resumes.

8. **Lines 43-47:** If an existing path is being corrected, the child vertex is queued to the correction queue for high priority correction.

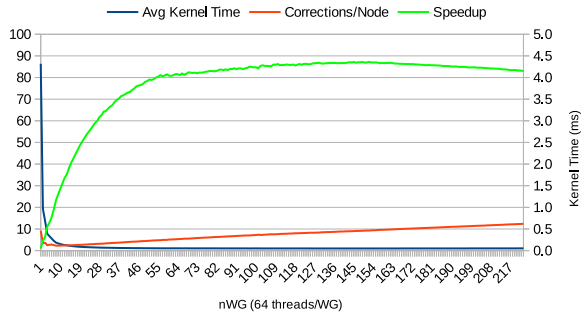
9. **Lines 59-63:** These lines check if the last child of a vertex has been processed. When this happens, *lnThreadsEndingThisCycle* is incremented and the *QueueDataAvailable* is set to false, indicating data has not yet arrive.

Dataset	n Vertices	n Edges	B-F Passes	Edges Per Vertex			
				Min	Max	Avg	Std
USA-road-d.NE	1,524,453	3,897,636	1,464	1	9	2.5567	0.9551
USA-road-d.NW	1,207,945	2,840,208	2,039	1	9	2.3513	0.9463
USA-road-d.NY	264,346	733,846	613	1	8	2.7761	0.9814

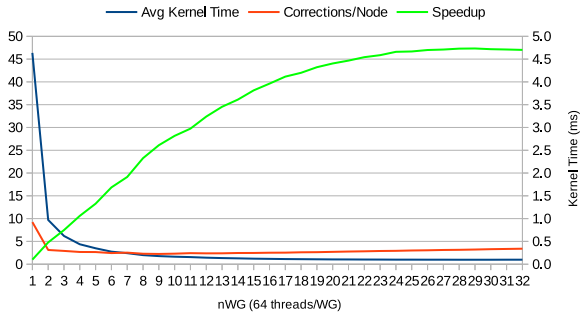
Table 8.1. Selected ninth DIMACS implementation challenged dataset statistics.

8.4 SSSP benchmark datasets

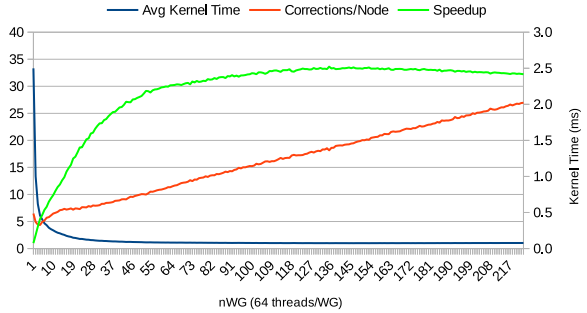
Table 8.1 details the statistics for the 3 selected DIMACS roadmap datasets. The column labeled "B-F Passes" gives the number of Bellman-Ford passes required to solve the SSSP problem for each dataset. The higher the value, the more relaxations due to an incorrect path choice in a previous pass, and the longer the execution time. While the NE and NW datasets are approximately the same size, the NW dataset required about twice as many Bellman-Ford passes as the NE dataset. This indicates the NW dataset required more corrections than the NE dataset. This resulted in longer runtimes across all benchmarks.



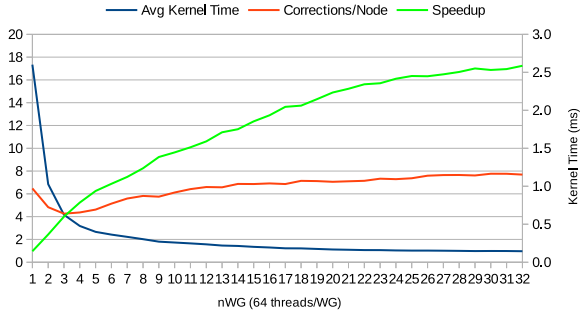
(a) USA-road-d.NE on Fiji



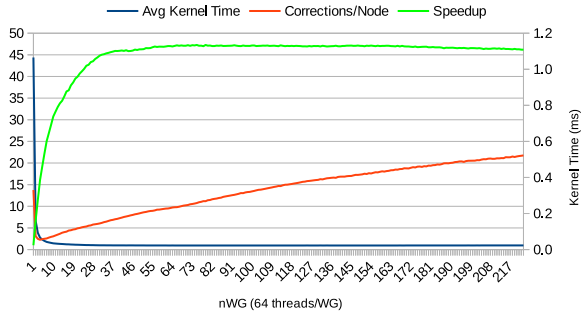
(b) USA-road-d.NE on Spectre



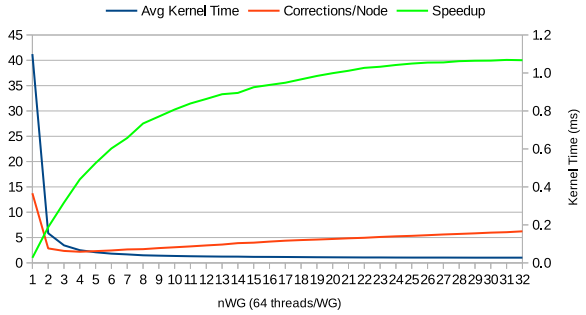
(c) USA-road-d.NW on Fiji.



(d) USA-road-d.NW on Spectre.



(e) USA-road-d.NY on Fiji.



(f) USA-road-d.NY on Spectre.

Figure 8.2. Speculate and correct SSSP performance on selected datasets/GPUs.

8.5 Speculate and correct SSSP kernel performance details

The correction process introduces a new form of overhead, which must be analyzed. This section analyzes the scalability of the proposed SSSP algorithm using three metrics measured as the number of threads are scaled:

1. **Execution Time:** Average kernel time was measured in ms. It is an absolute measure

of the overall performance, and shown as a blue line in the figures.

2. **Correction/Node:** This metric measures the correction overhead normalized by the number of vertices in the graph. It is shown as a red line in the figures.

3. **Speedup:** Speedup measures how well adding threads improves performance.

Figures 8.2 presents these metrics for the selected datasets and GPUs. The following summarizes those results.

1. Correction overhead rapidly flattened scalability.

2. Most execution time improvements are realized with only a few workgroups. This suggests mobile GPUs can benefit from this algorithm.

3. Corrections initially fall, but then gradually increase as workgroups are added. Thus, in addition to the corrections inherent to multiple data paths, there is an interaction with the number of threads used.

8.6 SSSP benchmark comparison

The following benchmarks[1] are used to compare the performance of the proposed SSSP algorithm to other benchmarks found in the literature:

1. **Proposed SSSP Algorithm**: The proposed SSSP algorithm is used as the baseline for computing speedup. It was configured for 4 chunks per work cycle and 64 threads per workgroup. Since the threads are heavily saturated, the proxy method was used for both enqueues and dequeues. On the Spectre GPU there were 32 workgroups (2,048 threads). On the Fiji GPU there were 224 workgroups (14,336 threads).

2. **CHAI**: CHAI [29] is a true heterogeneous suite of applications. The comparison is based on the recommended configuration of 2 CPU threads and 2,048 GPU threads.

[1]The Rodinia benchmark did not include an SSSP benchmark.

Since the CHAI suite is a true heterogeneous application, it will not run on the Fiji GPU.

3. **Pannotia**: Pannotia [8] is a suite of OpenCL 1.0-based applications. Pannotia has no configuration options.

Dataset	GPU	Proposed	CHAI		Pannotia	
		Avg Time	Speedup	Avg Time	Speedup	Avg Time
USA-road-d.NE	Fiji	52			28x	1,470
	Spectre	99	140x	13,811	231x	22,841
USA-road-d.NW	Fiji	77			23x	1,765
	Spectre	145	160x	23,305	169x	24,564
USA-road-d.NY	Fiji	23			11x	247
	Spectre	27	47x	1,274	131x	3,532

Table 8.2. SSSP performance (average kernel time in ms) summary.

Table 8.2 summarizes the performance. This shows:

1. On the Spectre, the proposed SSSP algorithm is at least two orders of magnitude faster than either CHAI or Pannotia.

2. On the Fiji, the proposed SSSP algorithm is at least one order of magnitude faster than either CHAI or Pannotia.

3. The scalability limiting effect of corrections limit the proposed SSSP algorithm's ability to exploit the extra threads available on the Fiji (See the green lines in Figures 8.2a, 8.2c and 8.2e). On the other hand, each Bellman-Ford pass can process all edges in parallel and without dependencies. These passes are able to exploit fully the extra threads on the Fiji, and explains why Pannotia was able to reclaim much of the performance deficit.

Even with correction overhead, the proposed SSSP algorithm's single-pass nature along with the use of the proposed queue offers significant processing advantage. For smaller

GPUs with a limited number of CUs, the advantage is more pronounced because the proposed SSSP algorithm achieves most of its performance using relatively few threads.

CHAPTER 9

RELATED WORK

This dissertation builds on prior work from several areas of research. This chapter presents related prior work by area of research. Research for this dissertation started by investigating why irregular workloads performed so poorly on GPUs. It quickly became evident that the atomic operations used to synchronize shared access to the task scheduler were at fault. Scalability curves exhibited a characteristic flattening as the number of threads increased. Early research identified the blocking mutex-based critical section approach to synchronization as a severe bottleneck. A further complicating factor is that traditional blocking mutex-base critical sections will deadlock on a GPU, and required significant restructuring. This inspired CDS research, especially lock-free CDSs. §9.1 (*Concurrent data structures*) gives the CDS research that most affected the research in this dissertation. .

9.1 Concurrent data structures

Concurrent lock-free linked lists based on CAS atomics were the early focus of research. They outperformed their blocking mutex-based critical section counterpart, but still exhibited the characteristic flattened scalability curve. Often the literature simply attributed this to "atomic overhead", but offered no details.

Our preliminary research revealed the actual cause was two-fold: retries for any reason, and lack of tasks to saturate and exploit all GPU threads. To address the retry issue our research focused on wait-free, k-FIFO concurrent queues.

This following citations in CDS research were the most influential. Each citation is presented along with a brief description of relevant content:

- **Treiber [51]:** Treiber was the first to acheive a lock-free CDS. It was a lock-free concurrent stack. This work demonstrated the concept and effectiveness of CDSs, and pioneered research in the area.

- **Herlihy [27]:** Herlihy did early theoretical work on wait-free synchronization. His main contribution was to prove that, while weak, atomic_fetch_add could be used as a wait-free primitive. It was a valuable hint toward the approach used.

- **Kirsch et al. [32, 33]:** Kirsch et al. proposed a lock-free k-FIFO queue that allowed up to k enqueue and k dequeue operations to occur in parallel. While each parallel operation is on a single element, it did inspire our research into a single operation on multiple elements.

- **Tsigas et al. [52]:** Tsigas et al. proposed a non-blocking concurrent FIFO that scaled well. It was based on a linked list using CAS synchronization. It proposed a novel pointer recycling mechanism that avoided the "ABA" problem.

- **Valois [57, 58]:** Valois proposed a lock-free linked-list concurrent queue implementation that avoided the "ABA" problem by not freeing deleted nodes. The issues associated with a link-list queue implementation eventually moved our research in the direction of array-based queues.

- **Deschev [12] and Deschev et al. [13]:** Studied the "ABA" (or pointer recycling) problem. These papers helped in the understanding of the issues giving rise to the problem and moved our research away from solutions that suffer from this problem.

- **Fomitchev et al. [20]:** Fomitchev et al. studied lock-free linked lists and skip lists in multi-processors. They observe linked-list form the basis of many data structures. They avoided the "ABA" problem by proposing a three-step deletion process. The paper influenced our research by highlighting the intractable issues related to lock-free CAS-based linked-list implementations.

- **Gao et al. [23]:** Gao et al. presented a generalized lock-free CAS-based algorithm for linked list concurrent data structures, and also presented a novel solution the "ABA" problem. This paper furthered highlighted the intractable issues related to lock-free CAS-based linked-list implementations.

- **Shafiei [49]:** This doctoral dissertation studied non-blocking array-based algorithms for stacks and queues. It used counters to implement array-based stacks and queues. The simplicity and effectiveness of the approach influenced our decision to move away from traditional linked-list solutions and embrace array-based solutions.

- **Kogan et al. [35]:** Kogan et al. proposed a methodology for creating fast wait-free data structures. The paper influenced our work by noting the effects of slow and fast paths on wait-free designs. It helped motivate our chunking strategy that partitions tasks into roughly uniform complexity. More importantly, it eventually lead to a design independent of path speed.

- **Kogan et al. [34]:** Kogan et al. studied wait-free queues with multiple enqueuers and dequeuers. This work extended the prior citation to include multiple enqueuers and dequeuers. It further motivated queue operations on multiple elements.

- **Evéquoz [18]:** Evéquoz presented an efficient and practical non-blocking implementation of a concurrent array-based FIFO queue. This paper emphasized the issues with linked-list solutions and reinforced the decision to use array-based solutions.

The net result of the above citations was to move our research away from link-list solutions in favor of array-based solutions. It also motivated queues that operate on multiple elements in parallel. Finally, it lead to a research on wait-free concurrent queues rather than lock-free solutions.

9.2 GPU persistent thread scheduling

This dissertation relies on an efficient GPU persistent thread scheduler. The SIMD architecture of GPUs added additional performance constraints to the concurrent queue design. The papers cited below moved our research to monolithic queues with arbitrary-n operations in favor of more complicated hierarchical designs such as work stealing and donations. Our research builds on the following prior work as described below:

- **Matthes et al.[19]:** Matthes was the first to propose persistent threads for managing long-term cooperative process. It is especially well-suited for managing irregular GPU workloads. This work pioneered the use of persistent threads for irregular workloads on GPUs.

- **Blumofe et al. [3]:** Blumofe et al. proposed the Cilk multi-threaded runtime package using work stealing.

- **Blumofe et al. [4]:** Blumofe et al. proposed a work-stealing scheduling strategy. This paper describes the work strealing algorithm using in Cilk, and ultimately on GPUs.

- **Hendler et al. [26]:** Hendler et al. proposed a work stealing scheduler that steals half the workload from overworked threads.

- **Cederman et al. [6]:** Cederman et al. studied task stealing in a GPU environment.

- **Cong et al. [22]:** Cong et al. studied adaptive work stealing for solving large, irregular graph problem.

- **Tzeng et al. [54]:** Tzeng studied task management for irregular GPU workloads. They advocated task sharing (donation). Task donation is an alternative

- **Aila et al. [1], Patney et al. [47] and Zhou et al. [60]:** Used simple monolithic queues for workload scheduling. They lead to an understanding that an arbitrary-n strategy could reduce atomic attempts, and thus retries.

- **Tseng [53]:** In his doctoral dissertation, Tseng studied the scheduling problem in many-core and heterogeneous GPUs. He examined various irregular workloads.

9.3 GPU graph algorithms

This dissertation implements BFS and SSSP algorithms on a GPU using a concurrent queue based persistent thread scheduler. It builds on the the following prior work:

- **Luo et al. [39]:** Luo et al.effectively accelerated the BFS problem on a GPU using CUDA. Their contribution was a hierarchical queue structure that reduced global memory access. It showed how a hierarchial queue structure can mitigate the effects of lock-step execution.

- **Lumsdaine et al. [38]:** Lumsdaine et al. studied the design challenges of parallel graph processing.

- **Merrill et al. [43]:** Merrill et al. studied scalable GPU graph traversal.

- **Hong et al. [28]:** Hong et al. studied efficient graph exploration on GPUs and CPUs.

- **Remis et al. [36]:** Remis et al. did a case study of BFS on social network graphs using heterogeneous processors. The large fan-out of social networks can cause special scheduling problems. Some threads can be scheduled a task with thousands of children while others have relatively few. No thread in the wavefront is scheduled new work until all threads complete their work. This helped understand the benefits of chunking.

- **Liu et al. [37]:** Liu et al. proposed a hybrid top down/bottom up BFS approach using CUDA. Their bottom up approach on social networks is the fastest algorithm we are aware of for social media graphs.

- **Bader et al. [11]:** Bader et al. studied the architectural requiremeents for efficient graph algorithm processing.

CHAPTER 10

SUMMARY AND CONCLUSION

This dissertation identifies retry overhead as the primary scalability limiting factor for a queue CDS. A SIMD thread environment guarantees worst-case atomic retry performance. The dissertation addresses these issues by proposing a queue that is retry-free for both atomic and queue operations. Further, to address SIMD issues, it proposes arbitrary-n queue operations that process an arbitrary number of elements in each queue operation. This enables use of a proxy thread in each wavefront that performs queue operations on behalf of all wavefront threads, thus avoiding SIMD thread issues because only the proxy thread in each SIMD thread group accesses the queue.

The dissertation empirically identifies a recommended configuration. It then analyzes the proposed queue using BFS on a wide variety of datasets and configurations to drive a persistent thread task scheduler based on the proposed queue. It shows a typical 2x performance improvement over CHAI, the closest competing benchmark.

The synthetic dataset significantly saturates all CUs. It was used to demonstrate the scalability of the proposed queue under full load. It shows the proposed queue scales to within 10% of ideal speedup with 14,336 active threads.

The scalability of the proposed queue adds saturation of CUs as a design objective. Before this work, retry overhead limited scalability. So, even if work were available, the effect of the extra threads was marginal.

This dissertation proposes a novel SSSP algorithm that saturates the GPU. Using two proposed queues – one for speculation and one for correction – the algorithm retains all the features of the classic Bellman-Ford algorithm, but does so in a single speculation pass, correcting errors in parallel as they are encountered.

The dissertation analyzes the performance characteristics of the algorithm. It then compares its performance to benchmarks found in the literature. The results strongly indicate the algorithm is a worthy competitor.

Finally, the performance of the proposed queue suggest research into improving dynamic parallelism would now be productive. The proposed SSSP algorithm is one example of this type of research. The queue's ability to scale gives the researcher the ability to exploit and analyze the effectiveness of new designs.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Aila, Timo and Laine, Samuli (2009), Understanding the Efficiency of Ray Traversal on GPUs, in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pp. 145–149, ACM, New York, NY, USA, doi:10.1145/1572769.1572792.

[2] Bellman, Richard (1958), On a Routing Problem, *Quarterly of Applied Mathematics*, 16(1), 87–90.

[3] Blumofe, Robert D and Joerg, Christopher F and Kuszmaul, Bradley C and Leiserson, Charles E and Randall, Keith H and Zhou, Yuli (1996), Cilk: An efficient multithreaded runtime system, *Journal of parallel and distributed computing*, 37(1), 55–69.

[4] Blumofe, Robert D. and Leiserson, Charles E. (1999), Scheduling Multithreaded Computations by Work Stealing, *J. ACM*, 46(5), 720–748, doi:10.1145/324133.324234.

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

[6] 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, pp. 57–64, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland.

[7] Che, S., 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)*, pp. 44–54, doi: 10.1109/IISWC.2009.5306797.

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

[9] Chin, Francis Y. and Lam, John and Chen, I-Ngo (1982), Efficient Parallel Algorithms for Some Graph Problems, *Commun. ACM*, 25(9), 659–665, doi:10.1145/358628.358650.

[10] Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009), *Introduction to Algorithms, Third Edition*, 3rd ed., The MIT Press.

[11] D. A. Bader and G. Cong and J. Feo (2005), On the architectural requirements for efficient execution of graph algorithms, in *2005 International Conference on Parallel Processing (ICPP'05)*, pp. 547–556, doi:10.1109/ICPP.2005.55.

[12] D. Dechev (2011), The ABA Problem in Multicore Data Structures with Collaborating Operations, in *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pp. 158–167, doi:10.4108/icst.collaboratecom.2011.247161.

[13] D. Dechev and P. Pirkelbauer and B. Stroustrup (2010), Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs, in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp. 185–192, doi:10.1109/ISORC.2010.10.

[14] Deng, Yangdong (Steve) and Wang, Bo David and Mu, Shuai (2009), Taming Irregular EDA Applications on GPUs, in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pp. 539–546, ACM, New York, NY, USA, doi:10.1145/1687399.1687501.

[15] Denysyuk, Oksana and Woelfel, Philipp (2015), Wait-Freedom is Harder Than Lock-Freedom Under Strong Linearizability, in *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, DISC 2015, pp. 60–74, Springer-Verlag, Berlin, Heidelberg, doi:10.1007/978-3-662-48653-5_5.

[16] Dijkstra, Edsger W (1959), A Note onTwo Problems in Connexion with Graphs, *Numerische Mathematik*, 1(1), 269–271.

[17] DIMACS (), DIMACS Challenge, http://dimacs.rutgers.edu/Challenges/.

[18] Evéquoz, Claude (2008), Practical, Fast and Simple Concurrent FIFO Queues Using Single Word Synchronization Primitives, in *Proceedings of the 13th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '08, pp. 59–72, Springer-Verlag, Berlin, Heidelberg, doi:10.1007/978-3-540-68624-8_5.

[19] Florian Matthes and Joachim W. Schmidt (1994), Persistent Threads, in *In Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pp. 403–414.

[20] Fomitchev, M., and E. Ruppert (2004), Lock-free linked lists and skip lists, in *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pp. 50–59, ACM, New York, NY, USA, doi:10.1145/1011767.1011776.

[21] Ford, LR and Fulkerson, Delbert R (1962), *Flows in Networks*.

[22] G. Cong and S. Kodali and S. Krishnamoorthy and D. Lea and V. Saraswat and T. Wen (2008), Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing, in *2008 37th International Conference on Parallel Processing*, pp. 536–545, doi:10.1109/ICPP.2008.88.

[23] Gao, H. and Hesselink, W. H. (2007), A General Lock-free Algorithm Using Compare-and-swap, *Inf. Comput.*, 205(2), 225–241, doi:10.1016/j.ic.2006.10.003.

[24] Harish, Pawan and Narayanan, PJ (2007), Accelerating large graph algorithms on the GPU using CUDA, in *International conference on high-performance computing*, pp. 197–208, Springer.

[25] Harris, Timothy L. (2001), A Pragmatic Implementation of Non-blocking Linked-Lists, in *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pp. 300–314, Springer-Verlag, London, UK, UK.

[26] Hendler, Danny and Shavit, Nir (2002), Non-blocking Steal-half Work Queues, in *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pp. 280–289, ACM, New York, NY, USA, doi:10.1145/571825.571876.

[27] Herlihy, M. (1991), Wait-free Synchronization, *ACM Trans. Program. Lang. Syst.*, 13(1), 124–149, doi:10.1145/114005.102808.

[28] Hong, S., T. Oguntebi, and K. Olukotun (2011), Efficient Parallel Graph Exploration on Multi-Core CPU and GPU, in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 78–88, doi:10.1109/PACT.2011.14.

[29] 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)*, pp. 43–54, doi:10.1109/ISPASS.2017.7975269.

[30] Jure Leskovec and Andrej Krevl (2014), SNAP Datasets: Stanford large network dataset collection, http://snap.stanford.edu/data.

[31] 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)*, pp. 1–14, doi:10.1109/InPar.2012.6339596.

[32] Kirsch, C., M. Lippautz, and H. Payer (2012), Fast and scalable k-fifo queues, *Tech. rep.*, Citeseer.

[33] Kirsch, Christoph M and Lippautz, Michael and Payer, Hannes (2013), Fast and scalable, lock-free k-FIFO queues, in *International Conference on Parallel Computing Technologies*, pp. 208–223, Springer.

[34] Kogan, Alex and Petrank, Erez (2011), Wait-free Queues with Multiple Enqueuers and Dequeuers, *SIGPLAN Not.*, 46(8), 223–234, doi:10.1145/2038037.1941585.

[35] Kogan, Alex and Petrank, Erez (2012), A Methodology for Creating Fast Wait-free Data Structures, *SIGPLAN Not.*, 47(8), 141–150, doi:10.1145/2370036.2145835.

[36] L. Remis and M. J. Garzaran and R. Asenjo and A. Navarro (2016), Breadth-First Search on Heterogeneous Platforms: A Case of Study on Social Networks, in *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 118–125, doi:10.1109/SBAC-PAD.2016.23.

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

[38] Lumsdaine, Andrew and Gregor, Douglas and Hendrickson, Bruce and Berry, Jonathan (2007), Challenges in Parallel Graph Processing, *Parallel Processing Letters*, 17(01), 5–20, doi:10.1142/S0129626407002843.

[39] Luo, Lijuan and Wong, Martin and Hwu, Wen-mei (2010), An Effective GPU Implementation of Breadth-First Search, in *Proceedings of the 47th Design Automation Conference*, DAC '10, pp. 52–55, ACM, New York, NY, USA, doi:10.1145/1837274.1837289.

[40] M. J. Flynn (1972), Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, C-21(9), 948–960, doi:10.1109/TC.1972.5009071.

[41] Mellor-Crummey, John M. and Scott, Michael L. (1991), Algorithms for Scalable Synchronization on Shared-memory Multiprocessors, *ACM Trans. Comput. Syst.*, 9(1), 21–65, doi:10.1145/103727.103729.

[42] Merrill, Duane and Garland, Michael and Grimshaw, Andrew (2012), Scalable GPU Graph Traversal, *SIGPLAN Not.*, 47(8), 117–128, doi:10.1145/2370036.2145832.

[43] Merrill, Duane and Garland, Michael and Grimshaw, Andrew (2015), High-Performance and Scalable GPU Graph Traversal, *ACM Trans. Parallel Comput.*, 1(2), 14:1–14:30, doi:10.1145/2717511.

[44] Michael, M. M. (2003), CAS-based lock-free algorithm for shared deques, in *European Conference on Parallel Processing*, pp. 651–660, Springer.

[45] Michael, M. M., and M. L. Scott (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, pp. 267–275, ACM, New York, NY, USA, doi:10.1145/248052.248106.

[46] Moir, Mark and Shavit, Nir (2004), Concurrent Data Structures.

[47] Patney, Anjul and Ebeida, Mohamed S. and Owens, John D. (2009), Parallel View-dependent Tessellation of Catmull-Clark Subdivision Surfaces, in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pp. 99–108, ACM, New York, NY, USA, doi:10.1145/1572769.1572785.

[48] Sedgewick, R. (1984), *Algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[49] Shafiei, Niloufar (2009), Non-blocking Array-Based Algorithms for Stacks and Queues, in *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ICDCN '09, pp. 55–66, Springer-Verlag, Berlin, Heidelberg, doi:10.1007/978-3-540-92295-7_10.

[50] Sundell, Håkan and Tsigas, Philippas (2005), Lock-free and Practical Doubly Linked List-based Deques Using Single-word Compare-and-swap, in *Proceedings of the 8th International Conference on Principles of Distributed Systems*, OPODIS'04, pp. 240–255, Springer-Verlag, Berlin, Heidelberg, doi:10.1007/11516798_18.

[51] Treiber, R. K. (1986), *Systems programming: Coping with parallelism*, International Business Machines Incorporated, Thomas J. Watson Research Center.

[52] Tsigas, Philippas and Zhang, Yi (2001), A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems, in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pp. 134–143, ACM, New York, NY, USA, doi:10.1145/378580.378611.

[53] Tzeng, Stanley (2013), *Scheduling on Manycore and Heterogeneous Graphics Processors*, University of California, Davis.

[54] 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, pp. 29–37, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland.

[55] Umut A. Acar and Arthur Charguéraud and Mike Rainey (2017), Parallel Work Inflation, Memory Effects, and their Empirical Analysis, *CoRR*, abs/1709.03767.

[56] V. Agarwal and F. Petrini and D. Pasetto and D. A. Bader (2010), Scalable Graph Exploration on Multicore Processors, in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, doi:10.1109/SC.2010.46.

[57] Valois, J. D. (1994), Implementing Lock-Free Queues, in *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pp. 64–69.

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

[59] Xia, Yinglong and Prasanna, Viktor K (2009), Topologically Adaptive Parallel Breadth-first Search on Multicore Processors, in *IASTED*, vol. 9, p. 91.

[60] Zhou, Kun and Hou, Qiming and Ren, Zhong and Gong, Minmin and Sun, Xin and Guo, Baining (2009), RenderAnts: Interactive Reyes Rendering on GPUs, in *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pp. 155:1–155:11, ACM, New York, NY, USA, doi:10.1145/1661412.1618501.

Appendices

Appendix A

Kernel Support Variables

Listing A.1. Kernel Support Variables

```
#ifndef __ChunkSize__
#define __ChunkSize__ 1
#endif

#ifndef __WGSize__
#define __WGSize__ 64
#endif

#define Missing (~0u)

typedef unsigned int uint32_t;

typedef struct cNode
{
  uint32_t StartingEdgeIndex;
  uint32_t nEdges;
} tNode;

typedef struct cEdge
{
  uint32_t NodeIndex;
} tEdge;

typedef uint32_t tWorkToken;

typedef struct sGPUExecutionContext
{
  // GPU geometry.
  const    uint32_t             WorkgroupSize;
  const    uint32_t             nWorkgroups;
  const    uint32_t             nGPUThreads;

  // Graph geometry.
  const    uint32_t             nNodes;
  const    uint32_t             nEdges;

  // Work queue support
  const    uint32_t             QueueSize;

  // BFS status and progress
  volatile uint32_t             CurrentCost;
  volatile uint32_t             WorkRemains;

  // Atomic counters.
  uint32_t CacheLineFiller1[16u];
  volatile uint32_t             nCurrentCostThreads;
  uint32_t CacheLineFiller2[16u];
  volatile uint32_t             nNextCostThreads;
  uint32_t CacheLineFiller3[16u];
```

Listing A.1 (Cont.): Kernel Support Variables

```
50
51    // Queue access atomic variables.
52    volatile uint32_t             WorkQueueFront;
53    uint32_t CacheLineFiller4[16u];
54    volatile uint32_t             WorkQueueRear;
55    uint32_t CacheLineFiller5[16u];
56  } tGPUExecutionContext;
57
58  // Loop cycle control.
59  #ifdef __MaxCycles__
60  #define WorkCycleLoopControl for (unsigned int Cycle = 0u; Parms->WorkRemains && (Cycle
        < __MaxCycles__); ++Cycle)
61  #else
62  #define WorkCycleLoopControl while (Parms->WorkRema)
63  #endif
```

Appendix B

Code Listing: The Direct Dequeue / Enqueue Kernel

Listing B.1. The Direct Dequeue / Enqueue Kernel

```
1  // ************************************************************************
2  // ************************************************************************
3  // ************************************************************************
4  //
5  // BFS, Dequeue Direct, Enqueue Direct
6  //
7  // ************************************************************************
8  // ************************************************************************
9  // ************************************************************************
10
11 __kernel void BFSDQDirectNQDirect(
12     volatile __global tGPUExecutionContext *Parms,
13              __global  const tNode         *Nodes,
14              __global  const tEdge         *Edges,
15     volatile __global  uint32_t            *Costs,
16     volatile __global  tWorkToken          *WorkQueue
17     )
18 {
19   // ************************************************************************
20   // Private variables.
21   // ************************************************************************
22
23   // CurrentEdge: Operating on this edge.
24   const __global tEdge *CurrentEdge;
25
26   // CurrentNode: Operating on this node.
27   tNode CurrentNode;
28
29   // CurrentNodeCost: Current cost level
30   uint32_t CurrentNodeCost = Missing;
31
32   // CurrentNodeIndex: Index of the current node.
33   uint32_t CurrentNodeIndex;
34
35   // ThreadSlotIndex: Work queue index associated with dequeues for this thread.
36   uint32_t DequeueThreadSlotIndex;
37
38   // EnqueueThreadSlotIndex: Work queue index associated with enqueues this thread.
39   uint32_t EnqueueThreadSlotIndex;
40
41   // GlobalID: This is the global thread ID.
42   uint32_t GlobalID = get_global_id(0);
43
44   // LocalID: This is the global thread ID.
45   uint32_t LocalID = get_local_id(0);
46
47   // nNextLevelEnqueues: The number of next level nodes enqueued for a current level
         node.
48   uint32_t nNextLevelEnqueues = 0u;
```

Listing B.1 (Cont.): The Direct Dequeue / Enqueue Kernel

```
49
50    // WorkgroupID: ID (sequence number) of this wavefront.
51    uint32_t WorkgroupID = get_global_id(0) / Parms->WorkgroupSize;
52
53    // QueueDataAvailable: True when data has arrived in queue slot, false otherwise.
54    bool QueueDataAvailable = false;
55
56    // QueueSize: Private copy of queue size for speed.
57    uint32_t QueueSize = Parms->QueueSize;
58
59    // ThreadNeedsWork: True when thread is hungry, false when it is fed.
60    // Initially all threads need to be fed.
61    bool ThreadNeedsWork = true;
62
63    // ************************************************************************
64    // Work cycle.
65    // ************************************************************************
66
67    WorkCycleLoopControl
68    {
69      // ************************************************************************
70      // Step 1:  Dequeue
71      // Hungry threads are assigned a queue slot.  Work may or may not have arrived for
          that slot.
72      // ************************************************************************
73
74      if (ThreadNeedsWork)
75      {
76        DequeueThreadSlotIndex    =  atomic_inc(&Parms->WorkQueueFront);
77        ThreadNeedsWork           =  false;
78        QueueDataAvailable        =  false;
79      }
80
81      // ************************************************************************
82      // Step 2:  Prolog
83      // ************************************************************************
84
85      if (!QueueDataAvailable)
86      {
87        // Check to see if data has arrived.
88        if ((DequeueThreadSlotIndex < QueueSize) && (QueueDataAvailable = (atomic_load((
              volatile atomic_uint *) (WorkQueue + DequeueThreadSlotIndex)) != Missing)))
89        {
90          // Work as arrived.  Setup to process this node.
91          // No atomics are needed because this is the only thread accessing the slot or
                node.
92
93          // Get work token (index of node to process).
94          CurrentNodeIndex = WorkQueue[DequeueThreadSlotIndex];
95
96          // Get assigned node.
97          CurrentNode = Nodes[CurrentNodeIndex];
98
99          // Get starting edge for this node.
100         CurrentEdge = Edges + CurrentNode.StartingEdgeIndex;
101
102         // Get current node cost;
103         CurrentNodeCost = Costs[CurrentNodeIndex];
104
105         // Node nodes enqueued yet.
106         nNextLevelEnqueues = 0u;
107       }
```

Listing B.1 (Cont.): The Direct Dequeue / Enqueue Kernel

```
108     }
109
110     // ****************************************************************************
111     // Step 3:  Do work unit.
112     // ****************************************************************************
113
114     if (QueueDataAvailable)
115     {
116       // Cannot process node unless at right level in graph.
117       if (CurrentNodeCost == Parms->CurrentCost)
118       {
119         // Process one chunk.
120         for (uint32_t Chunk = 0u; (CurrentNode.nEdges > 0u) && (Chunk < __ChunkSize__);
              ++Chunk)
121         {
122           // For an arbtrary graph, the edge can point to a node that has already been
                assigned a cost,
123           // or at the current level, two nodes can concurrently access a node at the
                next level.
124           // When that happens, this atomic selects a winner thread that will assign the
                 cost.
125           uint32_t EdgeNodeIndex      = CurrentEdge->NodeIndex;
126           volatile __global uint32_t *ThisCost = Costs + CurrentEdge->NodeIndex;
127           if (atomic_cmpxchg(ThisCost, Missing, CurrentNodeCost + 1u) == Missing)
128           {
129 #ifdef __EarlyLeafDetection_
130             if (Nodes[EdgeNodeIndex].nEdges)
131             {
132               // Queue only if node not a leaf node.
133               WorkQueue[atomic_inc(&Parms->WorkQueueRear)] = EdgeNodeIndex;
134               ++nNextLevelEnqueues;
135             }
136 #else
137             // Queue this node for new work.
138             WorkQueue[atomic_inc(&Parms->WorkQueueRear)] = EdgeNodeIndex;
139             ++nNextLevelEnqueues;
140 #endif
141           }
142           // Move to next edge.
143           ++CurrentEdge;
144
145           --CurrentNode.nEdges;
146         }
147
148         // If all edges processed, show thread is hungry (which triggers epilog).
149         ThreadNeedsWork = (CurrentNode.nEdges == 0u);
150       }
151     }
152     // ****************************************************************************
153     // Step 4:  Enqueue newly discovered work.
154     // ****************************************************************************
155
156     // ****************************************************************************
157     // Step 5:  Epilog
158     // ****************************************************************************
159
160     if (ThreadNeedsWork)
161     {
162       atomic_add(&Parms->nNextCostThreads, nNextLevelEnqueues);
163
164       // Check for level change.
165       if (atomic_dec(&Parms->nCurrentCostThreads) == 1u)
```

Listing B.1 (Cont.): The Direct Dequeue / Enqueue Kernel

```
166       {
167         // The only time the above test is true is when the last wavefront for a level
                completes.
168         // Multiple threads can complete, but these would be the last threads at the
                current level.
169         // No atomics are needed since no other thread completes.
170
171         // We're about to change levels.  Update level counts.
172         Parms->nCurrentCostThreads = Parms->nNextCostThreads;
173         Parms->nNextCostThreads    = 0u;
174
175         // If there are no queued current cost threads, we are done.
176         if (Parms->nCurrentCostThreads == 0u)
177         {
178           Parms->WorkRemains = 0u;
179         }
180
181         // Move to next cost level.
182         ++Parms->CurrentCost;
183       }
184     }
185
186     // Ensure global memory consistency
187     mem_fence(CLK_GLOBAL_MEM_FENCE);
188   }
189
190   return;
191 }
```

Appendix C

Code Listing: The Direct Dequeue / Proxy Enqueue Kernel

Listing C.1. The Direct Dequeue / Proxy Enqueue Kernel

```
1  // ************************************************************************
2  // ************************************************************************
3  // ************************************************************************
4  //
5  // BFS, Dequeue Direct, Enqueue Proxy
6  //
7  // ************************************************************************
8  // ************************************************************************
9  // ************************************************************************
10
11 __kernel void BFSDQDirectNQProxy(
12     volatile __global tGPUExecutionContext *Parms,
13              __global  const tNode         *Nodes,
14              __global  const tEdge         *Edges,
15     volatile __global  uint32_t            *Costs,
16     volatile __global  tWorkToken          *WorkQueue
17     )
18 {
19   // ************************************************************************
20   // Local variables.
21   // These must be declared volatile because the proxy thread performs work on behalf of
       all threads
22   // in workgroup.  The others threads must reload the data to see it.  The alternative
       is a more
23   // expensive fence.
24   // ************************************************************************
25
26   // lnThreadsEndingThisCycle: Count of the number of threads ending this cycle.
27   volatile __local uint32_t lnThreadsEndingThisCycle;
28
29   // lQueueSlotBaseIndex: Base index of dequeue slots for this workgroup.
30   volatile __local uint32_t lQueueSlotBaseIndex;
31
32   // lnQueueSlotsNeeded: Local int containg number of threads in this workgroup that
33   // need to be fed.
34   volatile __local uint32_t lnQueueSlotsNeeded;
35
36   // ************************************************************************
37   // Private variables.
38   // ************************************************************************
39
40   // CurrentEdge: Operating on this edge.
41   const __global tEdge *CurrentEdge;
42
43   // CurrentNode: Operating on this node.
44   tNode CurrentNode;
45
46   // CurrentNodeCost: Current cost level
47   uint32_t CurrentNodeCost = Missing;
```

Listing C.1 (Cont.): The Direct Dequeue / Proxy Enqueue Kernel

```
48
49    // CurrentNodeIndex: Index of the current node.
50    uint32_t CurrentNodeIndex;
51
52    // ThreadSlotIndex: Work queue index associated with dequeues for this thread.
53    uint32_t DequeueThreadSlotIndex;
54
55    // EnqueueThreadSlotIndex: Work queue index associated with enqueues this thread.
56    uint32_t EnqueueThreadSlotIndex;
57
58    // GlobalID: This is the global thread ID.
59    uint32_t GlobalID = get_global_id(0);
60
61    // IsProxyThread: True if this is the proxy thread, false otherwise.
62    // Local thread 0 is used as the proxy thread.
63    bool IsProxyThread = (get_local_id(0) == 0);
64
65    // LocalID: This is the global thread ID.
66    uint32_t LocalID = get_local_id(0);
67
68    // WorkgroupID: ID (sequence number) of this wavefront.
69    uint32_t WorkgroupID = get_global_id(0) / Parms->WorkgroupSize;
70
71    // Storage for newly discovered work.
72    uint32_t NewlyDiscoveredWork[__ChunkSize__];
73    uint32_t nNewlyDiscoveredWork = 0u;
74
75    // QueueDataAvailable: True when data has arrived in queue slot, false otherwise.
76    bool QueueDataAvailable = false;
77
78    // QueueSize: Private copy of queue size for speed.
79    uint32_t QueueSize = Parms->QueueSize;
80
81    // ThreadNeedsWork: True when thread is hungry, false when it is fed.
82    // Initially all threads need to be fed.
83    bool ThreadNeedsWork = true;
84
85    // ************************************************************************
86    // Initialize
87    // ************************************************************************
88
89    if (IsProxyThread)
90    {
91      lnQueueSlotsNeeded       = 0u;
92      lnThreadsEndingThisCycle = 0u;
93    }
94
95    // ************************************************************************
96    // Work cycle.
97    // ************************************************************************
98
99    WorkCycleLoopControl
100   {
101     // ************************************************************************
102     // Step 1:  Dequeue
103     // Hungry threads are assigned a queue slot.  Work may or may not have arrived for
          that slot.
104     // ************************************************************************
105
106     // Get base index of the slots for hungry threads.
107     if (IsProxyThread)
108     {
```

Listing C.1 (Cont.): The Direct Dequeue / Proxy Enqueue Kernel

```
109       // No threads have ended this cycle.
110       lnThreadsEndingThisCycle = 0u;
111     }
112
113     if (ThreadNeedsWork)
114     {
115       DequeueThreadSlotIndex    =  atomic_inc(&Parms->WorkQueueFront);
116       ThreadNeedsWork           =  false;
117       QueueDataAvailable        =  false;
118     }
119
120     // ************************************************************************
121     // Step 2:  Prolog
122     // ************************************************************************
123
124     if (!QueueDataAvailable)
125     {
126       // Check to see if data has arrived.
127       if ((DequeueThreadSlotIndex < QueueSize) && (QueueDataAvailable = (WorkQueue[
              DequeueThreadSlotIndex] != Missing)))
128       {
129         // Work as arrived.  Setup to process this node.
130         // No atomics are needed because this is the only thread accessing the slot or
              node.
131
132         // Get work token (index of node to process).
133         CurrentNodeIndex = WorkQueue[DequeueThreadSlotIndex];
134
135         // Get assigned node.
136         CurrentNode = Nodes[CurrentNodeIndex];
137
138         // Get starting edge for this node.
139         CurrentEdge = Edges + CurrentNode.StartingEdgeIndex;
140
141         // Get current node cost;
142         CurrentNodeCost = Costs[CurrentNodeIndex];
143       }
144     }
145
146     // ************************************************************************
147     // Step 3:  Do work unit.
148     // ************************************************************************
149
150     if (QueueDataAvailable)
151     {
152       // Cannot process node unless at right level in graph.
153       if (CurrentNodeCost == Parms->CurrentCost)
154       {
155         // Process one chunk.
156         for (uint32_t Chunk = 0u; (CurrentNode.nEdges > 0u) && (Chunk < __ChunkSize__);
              ++Chunk)
157         {
158           // For an arbtrary graph, the edge can point to a node that has already been
                assigned a cost,
159           // or at the current level, two nodes can concurrently access a node at the
                next level.
160           // When that happens, this atomic selects a winner thread that will assign the
                cost.
161           uint32_t EdgeNodeIndex      = CurrentEdge->NodeIndex;
162           volatile __global uint32_t *ThisCost = Costs + CurrentEdge->NodeIndex;
163           if (atomic_cmpxchg(ThisCost, Missing, CurrentNodeCost + 1u) == Missing)
164           {
```

Listing C.1 (Cont.): The Direct Dequeue / Proxy Enqueue Kernel

```
165 #ifdef __EarlyLeafDetection_
166             if (Nodes[EdgeNodeIndex].nEdges)
167             {
168               // Queue only if node not a leaf node.
169               NewlyDiscoveredWork[nNewlyDiscoveredWork++] = EdgeNodeIndex;
170             }
171 #else
172             // Queue this node for new work.
173             NewlyDiscoveredWork[nNewlyDiscoveredWork++] = EdgeNodeIndex;
174 #endif
175           }
176           // Move to next edge.
177           ++CurrentEdge;
178 
179           --CurrentNode.nEdges;
180         }
181 
182         // If all edges processed, show thread is hungry.
183         if (CurrentNode.nEdges == 0u)
184         {
185           atomic_inc(&lnThreadsEndingThisCycle);
186           ThreadNeedsWork = true;
187         }
188       }
189     }
190 
191     // ************************************************************************
192     // Step 4:  Enqueue newly discovered work.
193     // ************************************************************************
194 
195     // Initialize
196     if (IsProxyThread)
197     {
198       lnQueueSlotsNeeded = 0u;
199     }
200 
201     // Count all newly discovered work in this cycle and assign slot index for each
          thread.
202     if (nNewlyDiscoveredWork)
203     {
204       EnqueueThreadSlotIndex = atomic_add(&lnQueueSlotsNeeded, nNewlyDiscoveredWork);
205     }
206 
207     // Reserve space in queue, and get base index.
208     if (IsProxyThread)
209     {
210       lQueueSlotBaseIndex = atomic_add(&Parms->WorkQueueRear, lnQueueSlotsNeeded);
211     }
212 
213     if (nNewlyDiscoveredWork)
214     {
215       // Convert slot index to base index within queue.
216       EnqueueThreadSlotIndex += lQueueSlotBaseIndex;
217 
218       // Copy newly discovered work to the queue slot reserved for this work token.
219       for (uint32_t i = 0u; i < nNewlyDiscoveredWork; ++i)
220       {
221         WorkQueue[EnqueueThreadSlotIndex++] = NewlyDiscoveredWork[i];
222       }
223     }
224 
225     // ************************************************************************
```

Listing C.1 (Cont.): The Direct Dequeue / Proxy Enqueue Kernel

```
226     // Step 5:  Epilog
227     // ********************************************************************************
228
229     if (IsProxyThread)
230     {
231       // All newly discovered (queued) work is at next level.
232       atomic_add(&Parms->nNextCostThreads, lnQueueSlotsNeeded);
233
234       // Check for level change.
235       if (lnThreadsEndingThisCycle && (atomic_sub(&Parms->nCurrentCostThreads,
              lnThreadsEndingThisCycle) == lnThreadsEndingThisCycle))
236       {
237         // The only time the above test is true is when the last wavefront for a level
              completes.
238         // Multiple threads can complete, but these would be the last threads at the
              current level.
239         // No atomics are needed since no other thread completes.
240
241         // We're about to change levels.  Update level counts.
242         Parms->nCurrentCostThreads = Parms->nNextCostThreads;
243         Parms->nNextCostThreads    = 0u;
244
245         // If there are no queued current cost threads, we are done.
246         if (Parms->nCurrentCostThreads == 0u)
247         {
248           Parms->WorkRemains = 0u;
249         }
250
251         // Move to next cost level.
252         ++Parms->CurrentCost;
253       }
254     }
255
256     // Newly discovered work has been queued.  Reset starting index.
257     nNewlyDiscoveredWork = 0u;
258
259     // Ensure global memory consistency
260     mem_fence(CLK_GLOBAL_MEM_FENCE);
261   }
262
263   return;
264 }
```

Appendix D

Code Listing: The Proxy Dequeue / Direct Enqueue Kernel

Listing D.1. The Proxy Dequeue / Direct Enqueue Kernel

```
1  // ************************************************************************
2  // ************************************************************************
3  // ************************************************************************
4  //
5  // BFS, Dequeue Proxy, Enqueue Direct
6  //
7  // ************************************************************************
8  // ************************************************************************
9  // ************************************************************************
10
11 __kernel void BFSDQProxyNQDirect(
12     volatile __global tGPUExecutionContext *Parms,
13              __global  const tNode         *Nodes,
14              __global  const tEdge         *Edges,
15     volatile __global  uint32_t            *Costs,
16     volatile __global  tWorkToken          *WorkQueue
17     )
18 {
19   // ************************************************************************
20   // Local variables.
21   // These must be declared volatile because the proxy thread performs work on behalf of
         all threads
22   // in workgroup.  The others threads must reload the data to see it.  The alternative
         is a more
23   // expensive fence.
24   // ************************************************************************
25
26   // lnThreadsEndingThisCycle: Count of the number of threads ending this cycle.
27   volatile __local uint32_t lnThreadsEndingThisCycle;
28
29   // lQueueSlotBaseIndex: Base index of dequeue slots for this workgroup.
30   volatile __local uint32_t lQueueSlotBaseIndex;
31
32   // lnQueueSlotsNeeded: Local int containg number of threads in this workgroup that
33   // need to be fed.
34   volatile __local uint32_t lnQueueSlotsNeeded;
35
36   // ************************************************************************
37   // Private variables.
38   // ************************************************************************
39
40   // CurrentEdge: Operating on this edge.
41   const __global tEdge *CurrentEdge;
42
43   // CurrentNode: Operating on this node.
44   tNode CurrentNode;
45
46   // CurrentNodeCost: Current cost level
47   uint32_t CurrentNodeCost = Missing;
```

Listing D.1 (Cont.): The Proxy Dequeue / Direct Enqueue Kernel

```
48
49    // CurrentNodeIndex: Index of the current node.
50    uint32_t CurrentNodeIndex;
51
52    // ThreadSlotIndex: Work queue index associated with dequeues for this thread.
53    uint32_t DequeueThreadSlotIndex;
54
55    // EnqueueThreadSlotIndex: Work queue index associated with enqueues this thread.
56    uint32_t EnqueueThreadSlotIndex;
57
58    // GlobalID: This is the global thread ID.
59    uint32_t GlobalID = get_global_id(0);
60
61    // IsProxyThread: True if this is the proxy thread, false otherwise.
62    // Local thread 0 is used as the proxy thread.
63    bool IsProxyThread = (get_local_id(0) == 0);
64
65    // LocalID: This is the global thread ID.
66    uint32_t LocalID = get_local_id(0);
67
68    // nNextLevelEnqueues: The number of next level nodes enqueued for a current level
          node.
69    uint32_t nNextLevelEnqueues = 0u;
70
71    // WorkgroupID: ID (sequence number) of this wavefront.
72    uint32_t WorkgroupID = get_global_id(0) / Parms->WorkgroupSize;
73
74    // QueueDataAvailable: True when data has arrived in queue slot, false otherwise.
75    bool QueueDataAvailable = false;
76
77    // QueueSize: Private copy of queue size for speed.
78    uint32_t QueueSize = Parms->QueueSize;
79
80    // ThreadNeedsWork: True when thread is hungry, false when it is fed.
81    // Initially all threads need to be fed.
82    bool ThreadNeedsWork = true;
83
84    // ************************************************************************
85    // Work cycle.
86    // ************************************************************************
87
88    WorkCycleLoopControl
89    {
90      // ************************************************************************
91      // Step 1:  Dequeue
92      // Hungry threads are assigned a queue slot.  Work may or may not have arrived for
            that slot.
93      // ************************************************************************
94
95      // Get base index of the slots for hungry threads.
96      if (IsProxyThread)
97      {
98        lnQueueSlotsNeeded = 0u;
99      }
100
101     if (ThreadNeedsWork)
102     {
103       // Count all threads and assign each thread it's relative slot index;
104       DequeueThreadSlotIndex = atomic_inc(&lnQueueSlotsNeeded);
105     }
106
107     // Get base index of the slots for hungry threads.
```

Listing D.1 (Cont.): The Proxy Dequeue / Direct Enqueue Kernel

```
108     if (IsProxyThread)
109     {
110       lQueueSlotBaseIndex = atomic_add(&Parms->WorkQueueFront, lnQueueSlotsNeeded);
111     }
112
113     if (ThreadNeedsWork)
114     {
115       DequeueThreadSlotIndex    += lQueueSlotBaseIndex;
116       ThreadNeedsWork           =  false;
117       QueueDataAvailable        =  false;
118     }
119
120     // ************************************************************************
121     // Step 2:  Prolog
122     // ************************************************************************
123
124     if (!QueueDataAvailable)
125     {
126       // Check to see if data has arrived.
127       if ((DequeueThreadSlotIndex < QueueSize) && (QueueDataAvailable = (atomic_load((
              volatile atomic_uint *) (WorkQueue + DequeueThreadSlotIndex)) != Missing)))
128       {
129         // Work as arrived.  Setup to process this node.
130         // No atomics are needed because this is the only thread accessing the slot or
               node.
131
132         // Get work token (index of node to process).
133         CurrentNodeIndex = WorkQueue[DequeueThreadSlotIndex];
134
135         // Get assigned node.
136         CurrentNode = Nodes[CurrentNodeIndex];
137
138         // Get starting edge for this node.
139         CurrentEdge = Edges + CurrentNode.StartingEdgeIndex;
140
141         // Get current node cost;
142         CurrentNodeCost = Costs[CurrentNodeIndex];
143
144         // Node nodes enqueued yet.
145         nNextLevelEnqueues = 0u;
146       }
147     }
148
149     // ************************************************************************
150     // Step 3:  Do work unit.
151     // ************************************************************************
152
153     if (QueueDataAvailable)
154     {
155       // Cannot process node unless at right level in graph.
156       if (CurrentNodeCost == Parms->CurrentCost)
157       {
158         // Process one chunk.
159         for (uint32_t Chunk = 0u; (CurrentNode.nEdges > 0u) && (Chunk < __ChunkSize__);
               ++Chunk)
160         {
161           // For an arbtrary graph, the edge can point to a node that has already been
                 assigned a cost,
162           // or at the current level, two nodes can concurrently access a node at the
                 next level.
163           // When that happens, this atomic selects a winner thread that will assign the
                 cost.
```

Listing D.1 (Cont.): The Proxy Dequeue / Direct Enqueue Kernel

```
164           uint32_t EdgeNodeIndex      = CurrentEdge->NodeIndex;
165           volatile __global uint32_t *ThisCost = Costs + CurrentEdge->NodeIndex;
166           if (atomic_cmpxchg(ThisCost, Missing, CurrentNodeCost + 1u) == Missing)
167           {
168 #ifdef __EarlyLeafDetection_
169             if (Nodes[EdgeNodeIndex].nEdges)
170             {
171               // Queue only if node not a leaf node.
172               atomic_store((volatile atomic_uint *) (WorkQueue + atomic_inc(&Parms->
                      WorkQueueRear)), EdgeNodeIndex);
173               ++nNextLevelEnqueues;
174             }
175 #else
176             // Queue this node for new work.
177             atomic_store((volatile atomic_uint *) (WorkQueue + atomic_inc(&Parms->
                    WorkQueueRear)), EdgeNodeIndex);
178             ++nNextLevelEnqueues;
179 #endif
180           }
181           // Move to next edge.
182           ++CurrentEdge;
183 
184           --CurrentNode.nEdges;
185         }
186 
187         // If all edges processed, show thread is hungry (which triggers epilog).
188         ThreadNeedsWork = (CurrentNode.nEdges == 0u);
189       }
190     }
191 
192     // ************************************************************************
193     // Step 4:  Enqueue newly discovered work.
194     // ************************************************************************
195 
196     // ************************************************************************
197     // Step 5:  Epilog
198     // ************************************************************************
199 
200     if (ThreadNeedsWork)
201     {
202       atomic_add(&Parms->nNextCostThreads, nNextLevelEnqueues);
203 
204       // Check for level change.
205       if (atomic_dec(&Parms->nCurrentCostThreads) == 1u)
206       {
207         // The only time the above test is true is when the last wavefront for a level
                completes.
208         // Multiple threads can complete, but these would be the last threads at the
                current level.
209         // No atomics are needed since no other thread completes.
210 
211         // We're about to change levels.  Update level counts.
212         Parms->nCurrentCostThreads = Parms->nNextCostThreads;
213         Parms->nNextCostThreads    = 0u;
214 
215         // If there are no queued current cost threads, we are done.
216         Parms->WorkRemains = Parms->nCurrentCostThreads;
217 
218         // Move to next cost level.
219         ++Parms->CurrentCost;
220       }
221     }
```

Listing D.1 (Cont.): The Proxy Dequeue / Direct Enqueue Kernel

```
222     }
223
224     return;
225 }
```

Appendix E

Code Listing: The Proxy Dequeue / Enqueue Kernel

Listing E.1. The Proxy Dequeue / Proxy Enqueue Kernel

```
1  // ************************************************************************
2  // ************************************************************************
3  // ************************************************************************
4  //
5  // BFS, Dequeue Proxy, Enqueue Proxy
6  //
7  // ************************************************************************
8  // ************************************************************************
9  // ************************************************************************
10
11 __kernel void BFSDQProxyNQProxy(
12     volatile __global tGPUExecutionContext *Parms,
13              __global  const tNode         *Nodes,
14              __global  const tEdge         *Edges,
15     volatile __global  uint32_t            *Costs,
16     volatile __global  tWorkToken          *WorkQueue
17     )
18 {
19   // ************************************************************************
20   // Local variables.
21   // These must be declared volatile because the proxy thread performs work on behalf of
         all threads
22   // in workgroup.  The others threads must reload the data to see it.  The alternative
         is a more
23   // expensive fence.
24   // ************************************************************************
25
26   // lnThreadsEndingThisCycle: Count of the number of threads ending this cycle.
27   volatile __local uint32_t lnThreadsEndingThisCycle;
28
29   // lQueueSlotBaseIndex: Base index of dequeue slots for this workgroup.
30   volatile __local uint32_t lQueueSlotBaseIndex;
31
32   // lnQueueSlotsNeeded: Local int containg number of threads in this workgroup that
33   // need to be fed.
34   volatile __local uint32_t lnQueueSlotsNeeded;
35
36   // ************************************************************************
37   // Private variables.
38   // ************************************************************************
39
40   // CurrentEdge: Operating on this edge.
41   const __global tEdge *CurrentEdge;
42
43   // CurrentNode: Operating on this node.
44   tNode CurrentNode;
45
46   // CurrentNodeCost: Current cost level
47   uint32_t CurrentNodeCost = Missing;
```

Listing E.1 (Cont.): The Proxy Dequeue / Proxy Enqueue Kernel

```
48
49    // CurrentNodeIndex: Index of the current node.
50    uint32_t CurrentNodeIndex;
51
52    // ThreadSlotIndex: Work queue index associated with dequeues for this thread.
53    uint32_t DequeueThreadSlotIndex;
54
55    // EnqueueThreadSlotIndex: Work queue index associated with enqueues this thread.
56    uint32_t EnqueueThreadSlotIndex;
57
58    // GlobalID: This is the global thread ID.
59    uint32_t GlobalID = get_global_id(0);
60
61    // IsProxyThread: True if this is the proxy thread, false otherwise.
62    // Local thread 0 is used as the proxy thread.
63    bool IsProxyThread = (get_local_id(0) == 0);
64
65    // LocalID: This is the global thread ID.
66    uint32_t LocalID = get_local_id(0);
67
68    // WorkgroupID: ID (sequence number) of this wavefront.
69    uint32_t WorkgroupID = get_global_id(0) / Parms->WorkgroupSize;
70
71    // Storage for newly discovered work.
72    uint32_t NewlyDiscoveredWork[__ChunkSize__];
73    uint32_t nNewlyDiscoveredWork = 0u;
74
75    // QueueDataAvailable: True when data has arrived in queue slot, false otherwise.
76    bool QueueDataAvailable = false;
77
78    // QueueSize: Private copy of queue size for speed.
79    uint32_t QueueSize = Parms->QueueSize;
80
81    // ThreadNeedsWork: True when thread is hungry, false when it is fed.
82    // Initially all threads need to be fed.
83    bool ThreadNeedsWork = true;
84
85    // ************************************************************************
86    // Initialize
87    // ************************************************************************
88
89    if (IsProxyThread)
90    {
91      lnQueueSlotsNeeded       = 0u;
92      lnThreadsEndingThisCycle = 0u;
93    }
94
95    // ************************************************************************
96    // Work cycle.
97    // ************************************************************************
98
99    WorkCycleLoopControl
100   {
101     // ************************************************************************
102     // Step 1:  Dequeue
103     // Hungry threads are assigned a queue slot.  Work may or may not have arrived for
          that slot.
104     // ************************************************************************
105
106     // Get base index of the slots for hungry threads.
107     if (IsProxyThread)
108     {
```

Listing E.1 (Cont.): The Proxy Dequeue / Proxy Enqueue Kernel

```
109       lnQueueSlotsNeeded = 0u;
110       lnThreadsEndingThisCycle = 0u;
111     }
112
113     if (ThreadNeedsWork)
114     {
115       // Count all threads and assign each thread it's relative slot index;
116       DequeueThreadSlotIndex = atomic_inc(&lnQueueSlotsNeeded);
117     }
118
119     // Get base index of the slots for hungry threads.
120     if (IsProxyThread)
121     {
122       lQueueSlotBaseIndex = atomic_add(&Parms->WorkQueueFront, lnQueueSlotsNeeded);
123     }
124
125     if (ThreadNeedsWork)
126     {
127       DequeueThreadSlotIndex    += lQueueSlotBaseIndex;
128       ThreadNeedsWork           =  false;
129       QueueDataAvailable        =  false;
130     }
131
132     // ************************************************************************
133     // Step 2:  Prolog
134     // ************************************************************************
135
136     if (!QueueDataAvailable)
137     {
138       // Check to see if data has arrived.
139       if ((DequeueThreadSlotIndex < QueueSize) && (QueueDataAvailable = (WorkQueue[
              DequeueThreadSlotIndex] != Missing)))
140       {
141         // Work as arrived.  Setup to process this node.
142         // No atomics are needed because this is the only thread accessing the slot or
              node.
143
144         // Get work token (index of node to process).
145         CurrentNodeIndex = WorkQueue[DequeueThreadSlotIndex];
146
147         // Get assigned node.
148         CurrentNode = Nodes[CurrentNodeIndex];
149
150         // Get starting edge for this node.
151         CurrentEdge = Edges + CurrentNode.StartingEdgeIndex;
152
153         // Get current node cost;
154         CurrentNodeCost = Costs[CurrentNodeIndex];
155       }
156     }
157
158     // ************************************************************************
159     // Step 3:  Do work unit.
160     // ************************************************************************
161
162     if (QueueDataAvailable)
163     {
164       // Cannot process node unless at right level in graph.
165       if (CurrentNodeCost == Parms->CurrentCost)
166       {
167         // Process one chunk.
168         for (uint32_t Chunk = 0u; (CurrentNode.nEdges > 0u) && (Chunk < __ChunkSize__);
```

Listing E.1 (Cont.): The Proxy Dequeue / Proxy Enqueue Kernel

```
            ++Chunk)
169         {
170           // For an arbtrary graph, the edge can point to a node that has already been
                assigned a cost,
171           // or at the current level, two nodes can concurrently access a node at the
                next level.
172           // When that happens, this atomic selects a winner thread that will assign the
                cost.
173           uint32_t EdgeNodeIndex      = CurrentEdge->NodeIndex;
174           volatile __global uint32_t *ThisCost = Costs + CurrentEdge->NodeIndex;
175           if (atomic_cmpxchg(ThisCost, Missing, CurrentNodeCost + 1u) == Missing))
176           {
177 #ifdef __EarlyLeafDetection_
178             if (Nodes[EdgeNodeIndex].nEdges)
179             {
180               // Queue only if node not a leaf node.
181               NewlyDiscoveredWork[nNewlyDiscoveredWork++] = EdgeNodeIndex;
182             }
183 #else
184             // Queue this node for new work.
185             NewlyDiscoveredWork[nNewlyDiscoveredWork++] = EdgeNodeIndex;
186 #endif
187           }
188           // Move to next edge.
189           ++CurrentEdge;
190
191           --CurrentNode.nEdges;
192         }
193
194         // If all edges processed, show thread is hungry.
195         if (CurrentNode.nEdges == 0u)
196         {
197           atomic_inc(&lnThreadsEndingThisCycle);
198           ThreadNeedsWork = true;
199         }
200       }
201     }
202
203     // ************************************************************************
204     // Step 4:  Enqueue newly discovered work.
205     // ************************************************************************
206
207     // Initialize
208     if (IsProxyThread)
209     {
210       lnQueueSlotsNeeded = 0u;
211     }
212
213     // Count all newly discovered work in this cycle and assign slot index for each
            thread.
214     if (nNewlyDiscoveredWork)
215     {
216       EnqueueThreadSlotIndex = atomic_add(&lnQueueSlotsNeeded, nNewlyDiscoveredWork);
217     }
218
219     // Reserve space in queue, and get base index.
220     if (IsProxyThread)
221     {
222       lQueueSlotBaseIndex = atomic_add(&Parms->WorkQueueRear, lnQueueSlotsNeeded);
223     }
224
225     if (nNewlyDiscoveredWork)
```

Listing E.1 (Cont.): The Proxy Dequeue / Proxy Enqueue Kernel

```
226     {
227       // Convert slot index to base index within queue.
228       EnqueueThreadSlotIndex += lQueueSlotBaseIndex;
229
230       // Copy newly discovered work to the queue slot reserved for this work token.
231       for (uint32_t i = 0u; i < nNewlyDiscoveredWork; ++i)
232       {
233         WorkQueue[EnqueueThreadSlotIndex++] = NewlyDiscoveredWork[i];
234       }
235     }
236
237     // ****************************************************************************
238     // Step 5:  Epilog
239     // ****************************************************************************
240
241     if (IsProxyThread)
242     {
243       // All newly discovered (queued) work is at next level.
244       atomic_add(&Parms->nNextCostThreads, lnQueueSlotsNeeded);
245
246       // Check for level change.
247       if (lnThreadsEndingThisCycle && (atomic_sub(&Parms->nCurrentCostThreads,
              lnThreadsEndingThisCycle) == lnThreadsEndingThisCycle))
248       {
249         // The only time the above test is true is when the last wavefront for a level
              completes.
250         // Multiple threads can complete, but these would be the last threads at the
              current level.
251         // No atomics are needed since no other thread completes.
252
253         // We're about to change levels.  Update level counts.
254         Parms->nCurrentCostThreads = Parms->nNextCostThreads;
255         Parms->nNextCostThreads    = 0u;
256
257         // If there are no queued current cost threads, we are done.
258         if (Parms->nCurrentCostThreads == 0u)
259         {
260           Parms->WorkRemains = 0u;
261         }
262
263         // Move to next cost level.
264         ++Parms->CurrentCost;
265       }
266     }
267
268     // Newly discovered work has been queued.  Reset starting index.
269     nNewlyDiscoveredWork = 0u;
270
271     // Ensure global memory consistency
272     mem_fence(CLK_GLOBAL_MEM_FENCE);
273   }
274
275   return;
276 }
```

Appendix F

Code Listing: SSSP Kernel

Listing F.1. The proxy dequeue / proxy enqueue SSSP kernel

```
1  #pragma OPENCL EXTENSION cl_khr_int64_extended_atomics : enable
2
3  typedef uint  uint32_t;
4  typedef ulong uint64_t;
5  typedef long  int64_t;
6
7  typedef uint32_t tCount;
8  typedef uint32_t tIndex;
9  typedef uint32_t tWeight;
10
11 #ifndef __ChunkSize__
12 #define __ChunkSize__ 1
13 #endif
14
15 #ifndef __WGSize__
16 #define __WGSize__ 64
17 #endif
18
19 #define Missing ((uint32_t)(~0u))
20
21 // Utility macros for managing CostParent.
22 // CostParent is a 64-bit unsigned.
23 // The high-order 32-bits is the cost and the low-order 32-bits is the parent.
24 // Both are unsigned.
25 #define MakeCostParent(Cost, Parent)\
26   (uint64_t)((uint64_t)((uint64_t)(Cost) << 32u) | (uint64_t)(Parent))
27 #define GetCostParent(UL, Cost, Parent)\
28   (Cost = (uint32_t)(UL >> 32UL), Parent = (uint32_t)(UL && 0xFFFFFFFFUL))
29 #define GetCost(CP) ((uint32_t)(CP >> 32UL))
30 #define GetParent(CP) ((uint32_t)(CP & 0xFFFFFFFFUL))
31
32 typedef struct cNode
33 {
34   tIndex StartingEdgeIndex;
35   tCount nEdges;
36 } tNode;
37
38 typedef struct cEdge
39 {
40   tIndex  DistalNodeIndex;
41   tWeight Weight;
42 } tEdge;
43
44 typedef uint32_t tWorkToken;
45
46 typedef struct sGPUExecutionContext
47 {
48   // GPU geometry.
49   const    uint32_t             WorkgroupSize;
```

Listing F.1 (Cont.): The proxy dequeue / proxy enqueue SSSP kernel

```
50    const    uint32_t             nWorkgroups;
51    const    uint32_t             nGPUThreads;
52
53    // Graph geometry.
54    const    uint32_t             nNodes;
55    const    uint32_t             nEdges;
56
57    // Work queue support
58    const    uint32_t             SpecQueueSize;
59    const    uint32_t             CorrQueueSize;
60
61    // SSSP status and progress
62    volatile uint32_t             WorkRemains;
63
64    // For stopping.
65    volatile uint32_t             nQueuedOrActiveTasks;
66
67    // Queue access atomic variables.
68    volatile uint32_t             SpecQueueFront;
69    uint32_t CacheLineFiller4[16u];
70    volatile uint32_t             SpecQueueRear;
71    uint32_t CacheLineFiller5[16u];
72    volatile uint32_t             CorrQueueFront;
73    uint32_t CacheLineFiller6[16u];
74    volatile uint32_t             CorrQueueRear;
75    uint32_t CacheLineFiller7[16u];
76    volatile uint32_t             AbortCode;
77    uint32_t CacheLineFiller8[16u];
78  } tGPUExecutionContext;
79
80  // Loop cycle control.
81  #ifdef __MaxCycles__
82  #define WorkCycleLoopControl \
83    for (unsigned int Cycle = 0u; \
84         !Parms->AbortCode && Parms->nQueuedOrActiveTasks && (Cycle < __MaxCycles__); \
85         ++Cycle)
86  #else
87  #define WorkCycleLoopControl while (!Parms->AbortCode && Parms->nQueuedOrActiveTasks)
88  #endif
89
90  // ****************************************************************************
91  //
92  // SSSPSpecAndCorr, Dequeue Proxy, Enqueue Proxy
93  //
94  // ****************************************************************************
95
96  __kernel void SSSPSpecAndCorr(
97      volatile __global tGPUExecutionContext *Parms,
98               __global  const tNode         *Nodes,
99               __global  const tEdge         *Edges,
100     volatile __global  uint64_t            *CostsParents,
101     volatile __global  uint32_t            *RelaxationCount,
102     volatile __global  tWorkToken          *SpecQueue,
103     volatile __global  tWorkToken          *CorrQueue
104     )
105 {
106   // ****************************************************************************
107   // Local variables.
108   // These must be declared volatile because the proxy thread performs work on behalf
109   // of all threads in workgroup.  The others threads must reload the data to see it.
110   // The alternative is a more expensive fence.
111   // ****************************************************************************
```

Listing F.1 (Cont.): The proxy dequeue / proxy enqueue SSSP kernel

```
112
113   // lnThreadsEndingThisCycle: Count of the number of threads ending this cycle.
114   volatile __local uint32_t lnThreadsEndingThisCycle;
115
116   // lQueueSlotSpecBaseIndex: Base index of dequeue slots for this workgroup.
117   volatile __local uint32_t lQueueSlotSpecBaseIndex;
118
119   // lQueueSlotBaseCorrIndex: Base index of dequeue slots for this workgroup.
120   volatile __local uint32_t lQueueSlotCorrBaseIndex;
121
122   // lnSpecQueueSlotsNeeded: Local int containg number of threads in this workgroup that
123   // need to be fed.
124   volatile __local uint32_t lnSpecQueueSlotsNeeded;
125
126   // lnQueueCorrSlotsNeeded: Local int containg number of threads in this workgroup that
127   // need to be fed.
128   volatile __local uint32_t lnCorrQueueSlotsNeeded;
129
130   // ************************************************************************
131   // Private variables.
132   // ************************************************************************
133
134   // CurrentEdge: Operating on this edge.
135   const __global tEdge *CurrentEdge;
136
137   // ProximalNode: Operating on this node.
138   tNode ProximalNode;
139
140   // Correct queue size. (for mod arithmetic efficiency)
141   uint32_t CorrQueueSize = Parms->CorrQueueSize;
142
143   // ProximalNodeCost: Proximal node cost
144   uint32_t ProximalNodeCost = Missing;
145
146   // ProximalNodeIndex: Index of the proximal node.
147   uint32_t ProximalNodeIndex;
148
149   // DequeueThreadSpecSlotIndex: Spec queue index associated with dequeues for this
150   // thread.
151   uint32_t DequeueThreadSpecSlotIndex;
152
153   // DequeueThreadCorrSlotIndex: Corr queue index associated with dequeues for this
154   // thread.
155   uint32_t DequeueThreadCorrSlotIndex;
156
157   // EnqueueThreadSpecSlotIndex: Spec queue index associated with enqueues this thread.
158   uint32_t EnqueueThreadSpecSlotIndex;
159
160   // EnqueueThreadCorrSlotIndex: Corr queue index associated with enqueues this thread.
161   uint32_t EnqueueThreadCorrSlotIndex;
162
163   // GlobalID: This is the global thread ID.
164   uint32_t GlobalID = get_global_id(0);
165
166   // IsProxyThread: True if this is the proxy thread, false otherwise.
167   // Local thread 0 is used as the proxy thread.
168   bool IsProxyThread = (get_local_id(0) == 0);
169
170   // LocalID: This is the global thread ID.
171   uint32_t LocalID = get_local_id(0);
172
173   // nEdgesLeft to process for proximal node.
```

Listing F.1 (Cont.): The proxy dequeue / proxy enqueue SSSP kernel

```
174   uint32_t nEdgesLeft = 0u;
175
176   // WorkgroupID: ID (sequence number) of this wavefront.
177   uint32_t WorkgroupID = get_global_id(0) / Parms->WorkgroupSize;
178
179   // Storage for newly discovered work.
180   uint32_t NewlyDiscoveredSpecWork[__ChunkSize__];
181   uint32_t nNewlyDiscoveredSpecWork = 0u;
182
183   uint32_t NewlyDiscoveredCorrWork[__ChunkSize__];
184   uint32_t nNewlyDiscoveredCorrWork = 0u;
185
186   // QueueDataAvailable: True when data has arrived in queue slot, false otherwise.
187   bool QueueDataAvailable = false;
188
189   // ThreadNeedsSpecWork: True when thread is hungry for SpecQueue, false when it is
190   // fed.
191   // Initially all threads need to be fed.
192   bool ThreadNeedsSpecWork = true;
193
194   // ThreadNeedsCorrWork: True when thread is hungry for CorrQueue, false when it is
195   // fed.
196   // Initially all threads need to be fed.
197   bool ThreadNeedsCorrWork = true;
198
199   // ************************************************************************
200   // Initialize
201   // ************************************************************************
202
203   if (IsProxyThread)
204   {
205     lnSpecQueueSlotsNeeded   =
206       lnCorrQueueSlotsNeeded = 0u;
207     lnThreadsEndingThisCycle = 0u;
208   }
209
210   // ************************************************************************
211   // Work cycle.
212   // ************************************************************************
213
214   WorkCycleLoopControl
215   {
216     // **********************************************************************
217     // Step 1:  Dequeue
218     // Hungry threads are assigned a queue slot.  Work may or may not have arrived
219     // for that slot.
220     // **********************************************************************
221
222     // Get base index of the slots for hungry threads.
223     if (IsProxyThread)
224     {
225       lnSpecQueueSlotsNeeded   =
226         lnCorrQueueSlotsNeeded = 0u;
227       lnThreadsEndingThisCycle = 0u;
228     }
229
230     if (ThreadNeedsSpecWork)
231     {
232       // Count all threads and assign each thread it's relative slot index;
233       DequeueThreadSpecSlotIndex = atomic_inc(&lnSpecQueueSlotsNeeded);
234     }
235
```

Listing F.1 (Cont.): The proxy dequeue / proxy enqueue SSSP kernel

```
236     if (ThreadNeedsCorrWork)
237     {
238       // Count all threads and assign each thread it's relative slot index;
239       DequeueThreadCorrSlotIndex = atomic_inc(&lnCorrQueueSlotsNeeded);
240     }
241
242     // Get base index of the slots for hungry threads.
243     if (IsProxyThread)
244     {
245       lQueueSlotSpecBaseIndex =
246         atomic_add(&Parms->SpecQueueFront, lnSpecQueueSlotsNeeded);
247       lQueueSlotCorrBaseIndex =
248         atomic_add(&Parms->CorrQueueFront, lnCorrQueueSlotsNeeded);
249     }
250
251     if (ThreadNeedsSpecWork)
252     {
253       DequeueThreadSpecSlotIndex += lQueueSlotSpecBaseIndex;
254       ThreadNeedsSpecWork         = false;
255     }
256
257     if (ThreadNeedsCorrWork)
258     {
259       DequeueThreadCorrSlotIndex += lQueueSlotCorrBaseIndex;
260       ThreadNeedsCorrWork         = false;
261     }
262
263     // Terminology:
264     // +++++++++++++++      Edge      +++++++++++++
265     // | Proximal Node| -------------->|Distal Node|
266     // +++++++++++++++                +++++++++++++
267
268     // ***************************************************************************
269     // Step 2:  Data Arrrival and Prolog
270     // ***************************************************************************
271     if (!QueueDataAvailable)
272     {
273       // Check to see if data has arrived.  Correction queue is higher priority than
274       // speculate queue
275       ProximalNodeIndex = Missing;
276       if ((DequeueThreadCorrSlotIndex < Parms->CorrQueueRear) &&
277         (QueueDataAvailable = (CorrQueue[DequeueThreadCorrSlotIndex % CorrQueueSize] !=
278         Missing)))
279       {
280         // Work as arrived in spec queue.  Setup to process this node.
281         // No atomics are needed because this is the only thread accessing the slot
282         //  or node.
283
284         // Get work token (index of node to process).
285         ProximalNodeIndex = CorrQueue[DequeueThreadCorrSlotIndex % CorrQueueSize];
286
287         // Correction queue may reuse this slot.  Set it to missing.
288         CorrQueue[DequeueThreadCorrSlotIndex % CorrQueueSize] = Missing;
289
290         ThreadNeedsCorrWork = true;
291       } else if ((DequeueThreadSpecSlotIndex < Parms->SpecQueueRear) &&
292         (QueueDataAvailable = (SpecQueue[DequeueThreadSpecSlotIndex] != Missing)))
293       {
294         // Work as arrived in spec queue.  Setup to process this node.
295         // No atomics are needed because this is the only thread accessing the slot
296         // or node.
297
```

Listing F.1 (Cont.): The proxy dequeue / proxy enqueue SSSP kernel

```
298         // Get work token (index of node to process).
299         ProximalNodeIndex = SpecQueue[DequeueThreadSpecSlotIndex];
300         ThreadNeedsSpecWork = true;
301       }
302
303       if (ProximalNodeIndex != Missing)
304       {
305         // Get assigned node.
306         ProximalNode       = Nodes[ProximalNodeIndex];
307
308         // Get number of nodes to process.
309         nEdgesLeft         = ProximalNode.nEdges;
310
311         // Get starting edge for this node.
312         CurrentEdge        = Edges + ProximalNode.StartingEdgeIndex;
313
314         // Get proximal node cost;
315         ProximalNodeCost   = GetCost(CostsParents[ProximalNodeIndex]);
316       }
317     }
318
319     // ************************************************************************
320     // Step 3:  Do work unit.
321     // ************************************************************************
322
323     if (QueueDataAvailable)
324     {
325       // Process one chunk.
326       for (uint32_t Chunk = 0u; (nEdgesLeft > 0u) && (Chunk < __ChunkSize__); ++Chunk)
327       {
328         // For an arbtrary graph, the edge can point to a node that has already been
329         // assigned a cost, or at the current level, two nodes can concurrently access
330         // a node at the next level.  When that happens, this atomic selects a winner
331         // thread that will assign the cost.
332         uint32_t DistalNodeIndex = CurrentEdge->DistalNodeIndex;
333
334         // Ignore reverse edge in undirected graphs.
335         // Ensure distal node index is not my parent.
336         if (DistalNodeIndex != GetParent(CostsParents[ProximalNodeIndex]))
337         {
338           // The distal node does not point back to proximal node.
339           uint32_t NewDistalCost       = ProximalNodeCost + CurrentEdge->Weight;
340           uint64_t NewDistalCostParent = MakeCostParent(NewDistalCost,
341                                                         ProximalNodeIndex);
342           uint64_t OldDistalCostParent = atom_min(CostsParents + DistalNodeIndex,
343                                                   NewDistalCostParent);
344
345           uint32_t OldDistalCost       = GetCost(OldDistalCostParent);
346           if (OldDistalCost != NewDistalCost)
347           {
348             // Cost has improved.  We need to do a relaxation.
349             if (atomic_inc(RelaxationCount + DistalNodeIndex) >= Parms->nEdges)
350             {
351               // Vertrex relaxed too many times.  We must have a negative loop.
352               atomic_store((volatile __global atomic_uint *) &Parms->AbortCode, 1u);
353             }
354
355             // Check where to queue the new work.
356             if (OldDistalCost == Missing)
357             {
358               // Queue this node as new work for speculate queue
359               NewlyDiscoveredSpecWork[nNewlyDiscoveredSpecWork++] = DistalNodeIndex;
```

Listing F.1 (Cont.): The proxy dequeue / proxy enqueue SSSP kernel

```
360             }
361             else if (NewDistalCost < OldDistalCost)
362             {
363               // We have found a better cost and need to correct.
364               NewlyDiscoveredCorrWork[nNewlyDiscoveredCorrWork++] = DistalNodeIndex;
365             }
366           }
367         }
368
369         // Move to next edge.
370         ++CurrentEdge;
371
372         // Finished this edge, count it.
373         --nEdgesLeft;
374       }
375
376       // If all edges processed, show thread is hungry.
377       if (nEdgesLeft == 0u)
378       {
379         atomic_inc(&lnThreadsEndingThisCycle);
380         QueueDataAvailable = false;
381       }
382     }
383
384     // ************************************************************************
385     // Step 4:  Enqueue newly discovered work.
386     // ************************************************************************
387
388     // Initialize
389     if (IsProxyThread)
390     {
391       lnSpecQueueSlotsNeeded   =
392         lnCorrQueueSlotsNeeded = 0u;
393     }
394
395     // Count all newly discovered work in this cycle and assign slot index for each
396     // thread.
397     if (nNewlyDiscoveredSpecWork)
398     {
399       EnqueueThreadSpecSlotIndex = atomic_add(&lnSpecQueueSlotsNeeded,
400                                               nNewlyDiscoveredSpecWork);
401     }
402
403     if (nNewlyDiscoveredCorrWork)
404     {
405       EnqueueThreadCorrSlotIndex = atomic_add(&lnCorrQueueSlotsNeeded,
406                                              nNewlyDiscoveredCorrWork);
407     }
408
409     // Reserve space in spec queue, and get base index.
410     if (IsProxyThread && lnSpecQueueSlotsNeeded)
411     {
412       lQueueSlotSpecBaseIndex = atomic_add(&Parms->SpecQueueRear,
413                                            lnSpecQueueSlotsNeeded);
414     }
415
416     // Reserve space in corr queue, and get base index.
417     if (IsProxyThread && lnCorrQueueSlotsNeeded)
418     {
419       lQueueSlotCorrBaseIndex = atomic_add(&Parms->CorrQueueRear,
420                                            lnCorrQueueSlotsNeeded);
421     }
```

Listing F.1 (Cont.): The proxy dequeue / proxy enqueue SSSP kernel

```
422
423     // Copy new spec queue entries to queue in parallel.
424     if (nNewlyDiscoveredSpecWork)
425     {
426       // Convert slot index to base index within queue.
427       EnqueueThreadSpecSlotIndex += lQueueSlotSpecBaseIndex;
428
429       // Copy newly discovered work to the queue slot reserved for this work token.
430       for (uint32_t i = 0u; i < nNewlyDiscoveredSpecWork; ++i)
431       {
432         SpecQueue[EnqueueThreadSpecSlotIndex++] = NewlyDiscoveredSpecWork[i];
433       }
434     }
435
436     // Copy new corr queue entries to queue in parallel.
437     if (nNewlyDiscoveredCorrWork)
438     {
439       // Convert slot index to base index within queue.
440       EnqueueThreadCorrSlotIndex += lQueueSlotCorrBaseIndex;
441
442       // Copy newly discovered work to the queue slot reserved for this work token.
443       for (uint32_t i = 0u; i < nNewlyDiscoveredCorrWork; ++i)
444       {
445         CorrQueue[EnqueueThreadCorrSlotIndex++ % CorrQueueSize] =
446           NewlyDiscoveredCorrWork[i];
447       }
448     }
449
450     // ************************************************************************
451     // Step 5:  Epilog
452     // ************************************************************************
453
454     if (IsProxyThread)
455     {
456       // Update nQueuedOrActiveTasks
457       atomic_add(&Parms->nQueuedOrActiveTasks,
458         lnSpecQueueSlotsNeeded + lnCorrQueueSlotsNeeded - lnThreadsEndingThisCycle);
459     }
460
461     // Newly discovered work has been queued.  Reset starting index.
462     nNewlyDiscoveredSpecWork =
463       nNewlyDiscoveredCorrWork = 0u;
464
465     // Ensure global memory consistency
466     mem_fence(CLK_GLOBAL_MEM_FENCE);
467   }
468
469   return;
470 }
```

VITA

David Arthur Troendle

EDUCATION

Doctor of Philosophy in Computer Science at the University of Mississippi. August, 2013 - 2018. Dissertation title: "Scheduling Irregular Workloads on GPUs"

Master of Science in Mathematics at the University of New Orleans. August 1972 - August, 1976.

Bachelor of Science in Mathematics at Louisiana State University in New Orleans, (now the University of New Orleans). August 1968 - December, 1971.

Bachelor of Arts in Philosophy/Theology at St. Mary's University in San Antonio, Texas (incomplete). August 1967 - May, 1968.

ACADEMIC EMPLOYMENT

Instructor of Biometry, Louisiana State Univerity Health Sciences Center, August, 1976 - April, 2011. Duties include teaching C to Biometry graduate students.

Adjunct Instructor of Computer Science, University of New Orleans, August, 1976 - May, 2001. Duties include various computer science classes to undergraduate and graduate students.

Graduate Instructor, Department of Computer and Information Science, University of Mississippi, August, 2018 - present. Duties include teaching the C++ programming language

to undergraduate students.

PATENTS

Pianykh, Oleg S., David Troendle, John M. Tyler, and Wilfrido Castaneda-Zuniga. "Radiologist workstation." U.S. Patent 6,909,436, issued June 21, 2005.

PUBLICATIONS

Garbus, S. B., M. Pore, D. Troendle, M. Wheeler, and S. Garbus. "Catch employe hypertension and cut absenteeism rate." The International journal of occupational health & safety 44, no. 5 (1975): 48-9.

Pianykh, Oleg S. Digital imaging and communications in medicine (DICOM): a practical introduction and survival guide. Springer Science & Business Media, 2009. (Foreword to first edition.)

Sarbazi-Azad, Hamid. Advances in GPU Research and Practice. Morgan Kaufmann, 2016. (Book chapter, "Thread Communication and Synchronization on Massively Parallel GPUs", Tuan Ta, David Troendle, and Byunghyun Jang)

Choo, Kyoshin, David Troendle, Esraa A. Gad, and Byunghyun Jang. "Contention-Aware Selective Caching to Mitigate Intra-Warp Contention on GPUs." In Parallel and Distributed Computing (ISPDC), 2017 16th International Symposium on, pp. 1-8. IEEE, 2017.

Ta, Tuan, David Troendle, Xiaoqi Hu, and Byunghyun Jang. "Understanding the Impact of Fine-Grained Data Sharing and Thread Communication on Heterogeneous Workload Development." In Parallel and Distributed Computing (ISPDC), 2017 16th International Symposium on, pp. 132-139. IEEE, 2017.

ACADEMIC AWARDS

Phi Kappa Phi honor society, (University of New Orleans, Spring, 1976)

SAP graduate scholarship award (Fall, 2014).

Upsilon Pi Epsilon (Mississippi Gamma Chapter, Spring, 2015).

Phi Kappa Phi honor society (University of Mississippi, Fall, 2017).