Understanding the Performance of Concurrent Data Structures on Graphics Processors

Daniel Cederman, Bapi Chatterjee, and Philippas Tsigas*

Chalmers University of Technology, Sweden
{cederman,bapic,tsigas}@chalmers.se

Abstract. In this paper we revisit the design of concurrent data structures – specifically queues – and examine their performance portability with regard to the move from conventional CPUs to graphics processors. We have looked at both lock-based and lock-free algorithms and have, for comparison, implemented and optimized the same algorithms on both graphics processors and multi-core CPUs. Particular interest has been paid to study the difference between the old Tesla and the new Fermi and Kepler architectures in this context. We provide a comprehensive evaluation and analysis of our implementations on all examined platforms. Our results indicate that the queues are in general performance portable, but that platform specific optimizations are possible to increase performance. The Fermi and Kepler GPUs, with optimized atomic operations, are observed to provide excellent scalability for both lock-based and lock-free queues.

1 Introduction

While multi-core CPUs have been available for years, the use of GPUs as efficient programmable processing units is more recent. The advent of CUDA [1] and OpenCL [2] made general purpose programming on graphics processors more accessible to the non-graphics programmers. But still the problem of efficient algorithmic design and implementation of generic concurrent data structures for GPUs remains as challenging as ever.

Much research has been done in the area of concurrent data structures. There are efficient concurrent implementations of a variety of common data structures, such as stacks [3], queues [4–9] and skip-lists [10]. For a good overview of several concurrent data structures we refer to the chapter by Cederman et al. [11].

But while the aforementioned algorithms have all been implemented and evaluated on many different multi-core architectures, very little work has been done to evaluate them on graphics processors. Data structures targeting graphics applications have been implemented on GPUs, such as the kd-tree [12] and octree [13]. A C++ and Cg based template library [14] has been provided for random

* This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481 and the Swedish Foundation for Strategic Research as part of the project RIT-10-0033 "Software Abstractions for Heterogeneous Multi-core Computer".

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 883–894, 2012.

access data structures for GPUs. Load balancing schemes on GPUs [15] using different data structures have been designed. A set of blocking synchronization primitives for GPUs [16] has been presented that could aid in the development or porting of data structures.

With the introduction of atomic primitives on graphics processors, we hypothesize that many of the existing concurrent data structures for multi-core CPUs could be transferred to graphics processors, perhaps without much change in the design. To evaluate how performance portable the designs of already existing common data structure algorithms are, we have, for this paper, implemented a set of concurrent FIFO queues with different synchronization mechanisms on both graphics processors and on multi-core CPUs. We have performed experiments comparing and analyzing the performance and cache behavior of the algorithms. We have specifically looked at how the performance changes by the move from NVIDIA's Tesla architecture to the newer Fermi [17] and Kepler (GK104) [18] architectures.

The paper is organized as follows. In section 2, we introduce the concurrent data structures and describe the distinguishing features of the algorithms considered. Section 3 presents a brief description of the CUDA programming model and different GPU architectures. In section 4, we present the experimental setup. A detailed performance analysis is presented in section 5. Section 6 concludes the paper.

2 Concurrent Data Structures

Depending on the synchronization mechanism, we broadly classify concurrent data structures into two categories, namely *blocking* and *non-blocking*. In blocking synchronization, no progress guarantees are made. For non-blocking synchronization, there are a number of different types of progress guarantees that can be assured. The two most important ones are known as *wait-free* and *lock-free*. Wait-free synchronization ensures that all the non-faulty processes eventually succeed in finite number of processing steps. Lock-free synchronization guarantees that at least one of the non-faulty processes out of the contending set will succeed in a finite number of processing steps. In practice, wait-free synchronization is usually more expensive and is mostly used in real-time settings with high demands on predictability, while lock-free synchronization targets high-performance computing.

Lock-free algorithms for multiple threads require the use of atomic primitives, such as Compare-And-Swap (CAS). CAS can conditionally set the value of a memory word, in one atomic step, if at the time, it holds a value specified as a parameter to the operation. It is a powerful synchronization primitive, but is unfortunately also expensive compared to normal read and write operations.

In this paper we have looked at different types of queues to evaluate their performance portability when moved from the CPU domain to the GPU domain. The queue data structures that we have chosen to implement are representative of several different design choices, such as being array-based or linked-list-based, cache-aware or not, lock-free or blocking. We have divided them up into

two main categories, Single-Producer Single-Consumer (SPSC) and Multiple-Producer Multiple-Consumer (MPMC).

2.1 SPSC Queues

In '83, **Lamport** presented a lock-free array-based concurrent queue for the SPSC case [19]. For this case, synchronization can be achieved using only atomic read and write operations on shared head and tail pointers. No CAS operations are necessary. Having shared pointers cause a lot of cache thrashing however, as both the producer and consumer need to access the same variables in every operation.

The **FastForward** algorithm lowered the amount of cache thrashing by keeping the head and tail variables private to the consumer and producer, respectively [4]. The synchronization was instead performed using a special empty element that was inserted into the queue when an element was dequeued. The producer would then only insert elements when the next slot in the array contained such an element. Cache thrashing does however still occur when the producer catches up with the consumer. To lower this problem it was suggested to use a small delay to keep the threads apart. The settings used for the delay function are however so application dependant that we decided not to use it in our experiments.

The **BatchQueue** algorithm divides the array into two batches [5]. When the producer is writing to one batch, the consumer can read from the other. This removes much of the cache thrashing and also lowers the frequency at which the producer and consumer need to synchronize. The major disadvantage of this design is that a batch must be full before it can be read, leading to large latencies if elements are not enqueued fast enough. A suggested solution to this problem was to at regular intervals insert null elements into the queue. We deemed this as a poor solution and it is not used in the experiments. To take better advantage of the graphics hardware, we have also implemented a version of the BatchQueue where we copy the entire batch to the local shared memory, before reading individual elements from it. We call this version **Buffered BatchQueue**.

The **MCRingBuffer** algorithm is similar to the BatchQueue, but instead of having just two batches, it can handle an arbitrary number of batches. This can be used to find a balance between the latency caused by waiting for the other threads and the latency caused by synchronization. As for the BatchQueue we provide a version that copies the batches to the local shared memory. We call this version **Buffered MCRingBuffer**.

2.2 MPMC Queues

For the MPMC case we used the lock-free queue by Michael and Scott, henceforth called the **MS-Queue** [7]. It is based on a linked-list and adds items to the queue by using CAS to swap in a pointer at the tail node. The tail pointer is then moved to point to the new element, with the use of a CAS operation. This second step can be performed by the thread invoking the operation, or by another thread that

needs to help the original thread to finish before it can continue. This helping behavior is an important part of what makes the queue lock-free, as a thread never has to wait for another thread to finish.

We also used the lock-free queue by Tsigas and Zhang, henceforth called the **TZ-Queue**, which is an array-based queue [8]. Elements are here inserted into the array using CAS. The head and tail pointers are also moved using CAS, but it is done lazily, after every x:th element instead of after every element. In the experiments we got the best performance doing it every second operation.

To compare lock-free synchronization with blocking, we used the **lock-based** queue by Michael and Scott, which stores elements in a linked-list [7]. We used both the standard version, with separate locks for the enqueue and dequeue operation, and a simpler version with a common lock for both operations. For locks we used a basic spinlock, which spins on a variable protecting a critical section, until it can acquire it using CAS. As CAS operations are expensive, we also implemented a lock that does not use CAS, the bakery-lock by Lamport [20].

3 GPU Architectures

Graphics processors are massively parallel shared memory architectures excellently suitable for data parallel programs. A GPU has a number of stream multiprocessors (SMs), each having many cores. The SMs have registers and a local fast shared memory available for access to threads and thread blocks (group of threads) respectively, executing on them. The global memory, the main graphics memory, is shared by all the thread blocks and the access is relatively slow compared to that of the local shared memory.

In this work we have used CUDA for all GPU implementations. CUDA is a mature programming environment for programming on GPUs. In CUDA threads are grouped into blocks where all threads in a specific block execute on the same SM. Threads in a block are in turn grouped into so called warps of 32 consecutive threads. These warps are then scheduled by the hardware scheduler. Exactly how the scheduler schedules warps is unspecified. This is problematic when using locks, as there is a potential for deadlocks if the scheduler is unfair. For lock-free algorithms this is not an issue, as they are guaranteed to make progress regardless of the scheduler.

The different generations of CUDA programmable GPUs are categorized in compute capabilities (CC) and are identified more popularly by their architecture's codename. CC 1.x are Tesla, 2.x are Fermi and 3.x are Kepler. The architectural features depend on the compute capability of the GPU. In particular the availability of atomic functions has been varying with the compute capabilities. In CC 1.0 there were no atomic operations available, from CC 1.1 onwards there are atomic operations available on the global memory and from CC 1.2 also for the shared memory. An important addition to the GPUs in the Fermi and Kepler architectures is the availability of a unified L2 cache and a configurable L1 cache. The performance of the atomic operations significantly

increased in Fermi, with the atomic unit working on the L2 cache, instead of on the global memory [16]. The bandwidth of L2 cache increased in Kepler so that it is now 73% faster than that in Fermi [18]. The speed of atomic operations has also been significantly increased in Kepler as compared to Fermi.

4 Experimental Setup

The experiments were performed on four different types of graphics processors, with different memory clock rates, multiprocessor counts and compute capabilities. To explore the difference in performance between CPUs and GPUs, the same experiments were also performed on a conventional multi-core system, a 12-core Intel system (24 cores with HyperThreading). See Table 1 for an overview of the platforms used.

Table 1. Platforms used in experiments. Counting multiprocessors as cores in GPU.

Name	Clock speed	Memory clock rate	Cores	Cache	Architecture(CC)
GeForce 8800 GT	1.6 GHz	1.0 GHz	14	0	1.1 (Tesla)
GeForce GTX 280	1.3 GHz	1.1 GHz	30	0	1.3 (Tesla)
Tesla C2050	1.2 GHz	1.5 GHz	14	786 kB	2.0 (Fermi)
GeForce GTX 680	1.1 GHz	3.0 GHz	8	512 kB	3.0 (Kepler)
Intel E5645 (2x)	2.4 GHz	0.7 GHz	24	12 MB	

In the experiments we only consider communication between thread blocks, not between individual threads in a thread block.

For the SPSC experiments, a thread from one thread block was assigned the role of the producer and another thread from a second block the role of the consumer. The performance was measured by counting the number of successful enqueue/dequeue operations per ms that could be achieved when communicating a set of integers from the producer to the consumer. Enqueue operations on full queues or dequeue operations on empty queues were not counted. Local variables, variables that are only accessed by either the consumer or the producer, are placed in the shared memory to remove unnecessary communication with the global memory. For buffered queues, 32 threads were used for memory transfer between global and shared memory to take advantage of the hardware's coalescing of memory accesses. All array-based queues had a maximum length of 4096 elements. The MCRingBuffer used a batch size of 128 whereas the BatchQueue by design has batches of size as of half the queue size, in this case 2048. For the CPU experiments care was taken to place the consumer and producer on different sockets, to emphasize the penalty taken by using an inefficient memory access strategy.

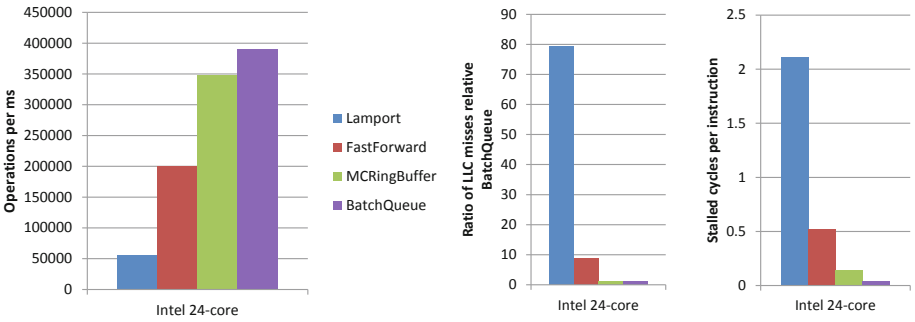
For the MPMC experiments a varying number of thread blocks were used, from 2 up to 60. Each thread block performed 25% enqueue operations and 75% dequeue operations randomly, using a uniform distribution. Two scenarios were used, one with high contention, where operations were performed one after

another, and one with low contention, in which a small amount of work was performed between the operations. The performance was measured in the number of successful operations per ms in total.

5 Performance Analysis

5.1 SPSC Queues

Figure 1(a) depicts the result from the experiments on the CPU system. It is clear from the figure that even the small difference in access pattern between the Lamport and the FastForward algorithms has a significant impact on the performance. The number of operations per ms differ by a factor of four between the two algorithms. The cache access profile in Figure 1(b) shows that the number of cache misses goes down dramatically when the head and tail variables are no longer shared between processes. It goes down even further when the producer and the consumer are forced to work on different memory locations. The figure also shows that the number of stalled cycles per instructions matches the cache misses relatively well. The reason for the performance difference between the BatchQueue and the MCRingBuffer, which both have a similar number of cache misses, lies in the difference between the size of the batches. This causes more frequent reads and writes of shared variables compared to the BatchQueue. It was observed that increasing the batch size lowers the synchronization overhead and the number of stalled cycles and improves the performance of the MCRingBuffer and brings it close to that of the BatchQueue.



(a) Comparison of SPSC queues. (b) Cache profile.

Fig. 1. Comparison of SPSC queues on the CPU based system

Figure 2 shows the results for the same experiment performed on the graphics processors. On the Tesla processors there are no cache memories available, which removes the problem of cache thrashing and causes the Lamport and FastForward algorithms to give similar results. In contrast to the CPU implementations, here the MCRingBuffer is actually faster than the BatchQueue. This is due to the fact that the BatchQueue enqueuer is faster than the dequeuer and has to

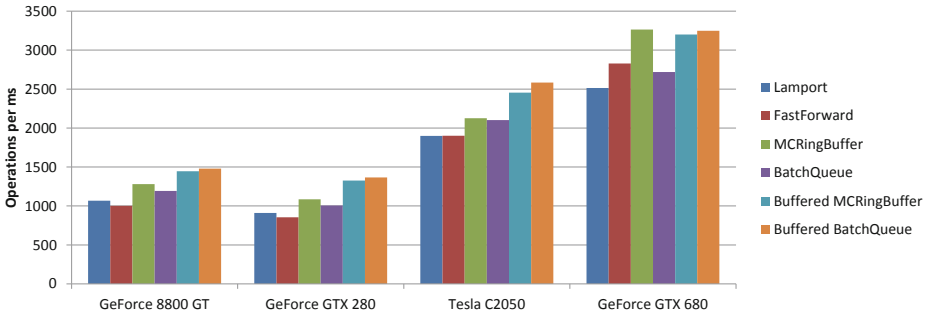
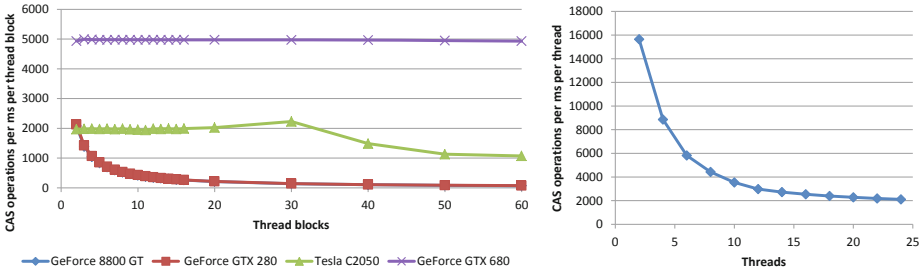


Fig. 2. Comparison of SPSC queues on four different GPUs

wait for a longer time for the larger batches to be processed. The smaller batch size in MCRingBuffer thus has an advantage here. The two buffered versions lower the overhead, as for most operations the data will be available locally from the shared memory. It is only at the end of a batch that the shared variables and the elements stored in the queue need to be accessed. This access is done using coalesced reads and writes, which speeds up the operation. When the queues are buffered, the BatchQueue becomes faster than the MCRingBuffer. Thus the overhead of the more frequent batch copies became more dominant. The performance on the Fermi and Kepler graphics processor is significantly better compared to the other processors, benefiting from the faster memory clock rate and the cache memory. The speed of the L2 cache is however not enough to make the unbuffered queues comparable with the buffered ones on the Fermi processor. On the Kepler processor, on the other hand, with its faster cache and higher memory clock rate, the unbuffered MCRingbuffer performs similarly to the buffered queues. The SPSC queues that we have examined thus need to be rewritten to achieve maximum performance on most GPUs. This might however change with the proliferation of the Kepler architecture.

5.2 MPMC Queues

All MPMC queue algorithms, except the ones that used the bakery-lock, make use of the CAS primitive. To visualize the behavior of the CAS primitive we measured the number of CAS operations that could be performed per thread block per ms for a varying number of thread blocks. The result is available in Figure 3. We see in Figure 3(a) that when the contention increases for the Tesla processors the number of CAS operations per ms drops quickly. However, it is observed that the CAS operations scale well on the Fermi, for up to 40 thread blocks, at high speed. The increased performance of the atomic primitives was one of the major improvements done when creating the Fermi architecture. The atomic operations are now handled at the L2 cache level and no longer need to access the global memory [16]. The Kepler processor has twice the memory clock rate of the Fermi processor and we can see that the CAS operations scales



(a) CAS behavior on the GPUs. (b) CAS behavior on the CPU.

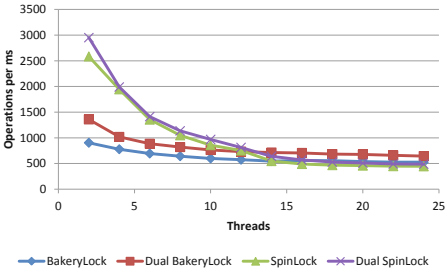
Fig. 3. Visualization of the CAS behavior on the GPUs and the CPU

perfect despite increased contention. Figure 3(b) shows that on the conventional system the performance is quite high when few threads perform CAS operations, but the performance drops rapidly as the contention increases.

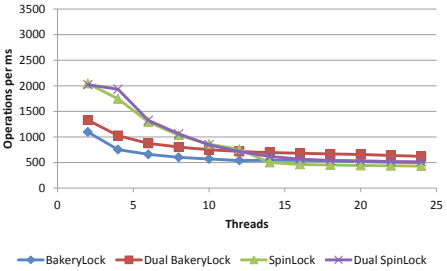
Figure 4 shows the performance of the MPMC queues on the CPU-based system. Looking first at the topmost graphs, which shows the result using just lock-based queues, we see that for a low number of threads the dual spinlock based queue clearly outperforms the bakery lock based queues. The bakery lock does not use any expensive CAS operation, but the overhead of the algorithm is still too high, until the number of threads goes above the number of cores and starts to use hyperthreading. The difference between dual and single spinlock is insignificant, however between the dual and the single bakery lock there is a noticeable difference.

The lower two graphs show the comparison results for the two lock-free queues together with the best lock-based one, the dual spinlock queue. The lock-free queues clearly outperform the lock-based one for all number of threads and for both contention levels. The array-based TZ-queue exhibits better results for the lower range of threads, but it is quickly overtaken by the linked-list based MS-queue. When hyperthreading kicks in, the performance does not drop any more for any of the queues.

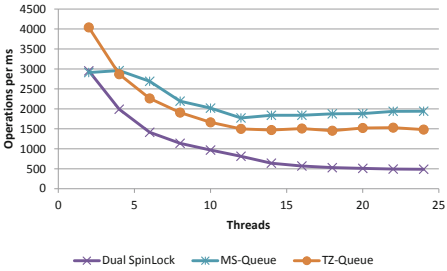
The measurements taken for the lock-based queues on the Fermi and one of the Tesla graphics processors are shown in Figure 5. Just as in the CPU experiments the dual spinlock queue excels among the lock-based queues. There is however a much clearer performance difference between the dual and single spinlock queues in all graphs, although not for the low contention cases when using few thread blocks. The peak in the result in Figure 5(a) is due to the overhead of the benchmark and the non-atomic parts of the queue. When contention is lowered, as in Figure 5(b), the peak moves to the right. After the peak the cost of the atomic operations become dominant, and the performance drops. For the Fermi-processor, in Figure 5(c), the performance for the spinlock based queues is significantly higher, while at the same time scaling much better. As we could see in Figure 3(a), this is due to the much improved atomic operations of the Fermi-architecture.



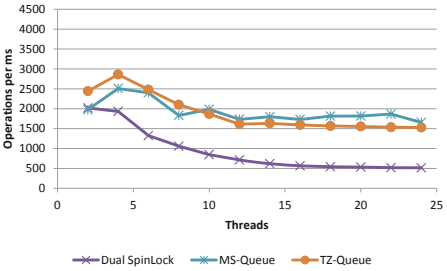
(a) Lock-based queues (*High* contention).



(b) Lock-based queues (*Low* contention).

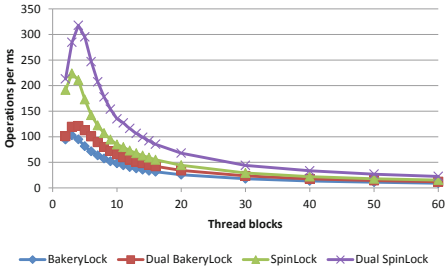


(c) Best lock-based and lock-free queues (*High* contention).

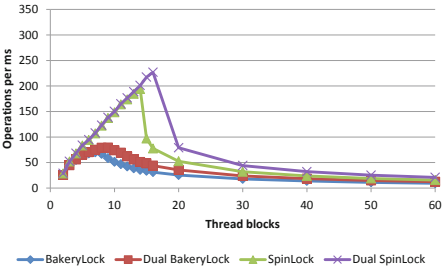


(d) Best lock-based and lock-free queues (*Low* contention).

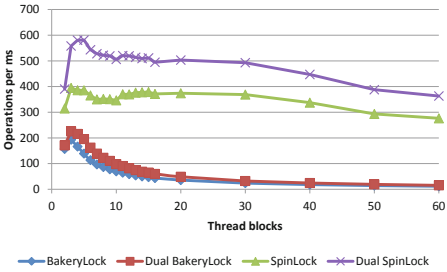
Fig. 4. Comparison of MPMC queues on the Intel 24-core system under *high* and *low* contention scenarios



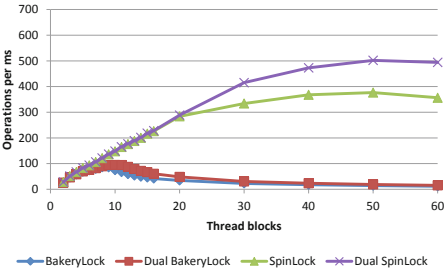
(a) GeForce GTX 280 (*High* Contention)



(b) GeForce GTX 280 (*Low* Contention)

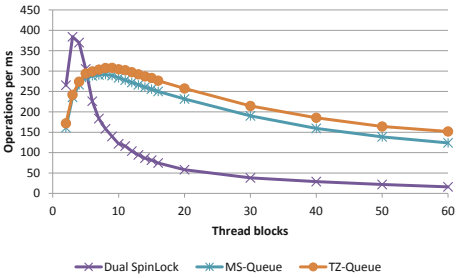


(c) Tesla C2050 (*High* Contention)

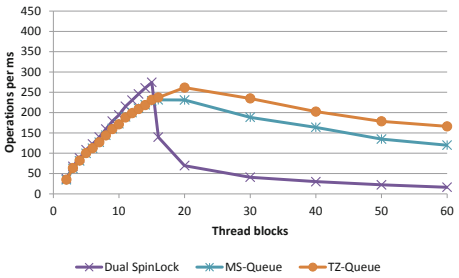


(d) Tesla C2050 (*Low* Contention)

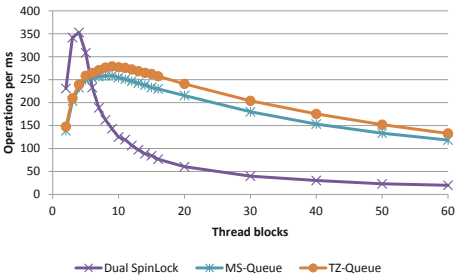
Fig. 5. Comparison of lock-based MPMC queues on two GPUs under *high* and *low* contention scenarios



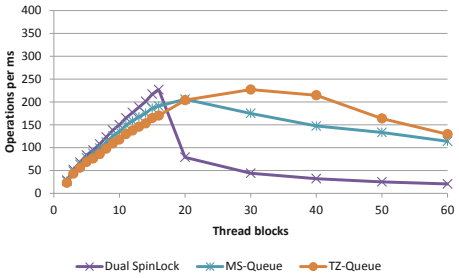
(a) GeForce 8800 GT (*High* Contention)



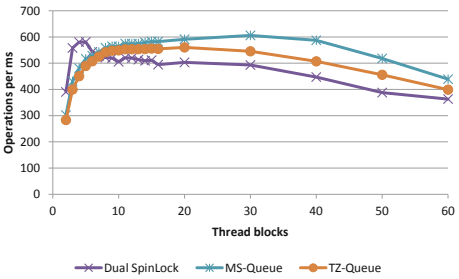
(b) GeForce 8800 GT (*Low* Contention)



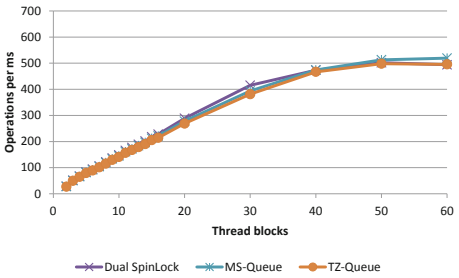
(c) GeForce GTX 280 (*High* Contention)



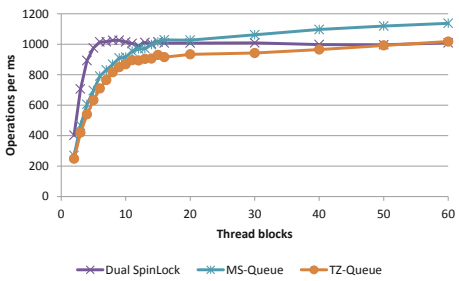
(d) GeForce GTX 280 (*Low* Contention)



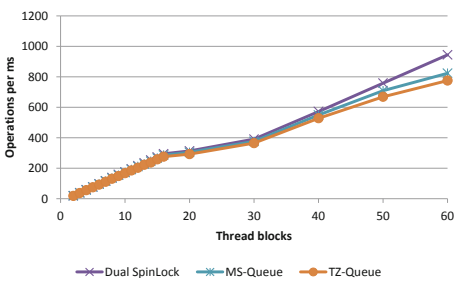
(e) Tesla C2050 (*High* Contention)



(f) Tesla C2050 (*Low* Contention)



(g) GeForce GTX 680 (*High* Contention)



(h) GeForce GTX 680 (*Low* Contention)

Fig. 6. Comparison of the best lock-based and lock-free MPMC queues on four GPUs under *high* and *low* contention scenarios

Comparing the dual spinlock queue with the lock-free queues, in Figure 6 we see that the lock-free queues scale much better than the lock-based one and provides the best performance when the thread block count is high. The spinlock queue does however achieve a better result on all graphics processors for a low number of thread blocks. As the contention is lowered, it remains useful for a higher number of threads. The array-based TZ-queue outperforms the linked-list based MS-queue on both the Tesla processors, but falls short on the Fermi and Kepler processors, Figure 6(e). When contention is lowered on the Fermi-processor, Figure 6(f), there is no longer any difference between the lock-based and the lock-free queues.

6 Conclusion and Future Work

In this paper we have examined the performance portability of common SPSC and MPMC queues. From our experiments on the SPSC queues we found that the best performing queues on the CPU were also the ones that performed well on the GPUs. It was however clear that the cache on the Fermi-architecture was not enough to remove the benefit of redesigning the algorithms to take advantage of the local shared memory. For the MPMC queue experiments we saw similar results in scalability for the GPU-versions on the Tesla processors as we did for the CPU-version. On the Fermi processor the result was surprising however. The scalability was close to perfect and for low contention there was no difference between the lock-based and the lock-free queues. The Fermi architecture has significantly improved the performance of atomic operations and this is an indication that new algorithmic designs should be considered to more properly take advantage of this new behavior. The Kepler architecture has continued in this direction and now provides atomic operations with performance competitive to that of conventional CPUs.

We will continue this work by studying the behavior of other concurrent data structures with higher potential to scale than queues, such as dictionaries and trees. Most queue data structures suffer from the fact that only two operations can succeed concurrently in the best case, whereas for a dictionary there are no such limitations.

References

1. NVIDIA: NVIDIA CUDA C Programming Guide. 4.0 edn. (2011)
2. The Khronos Group Inc.: OpenCl Reference Pages. 1.2 edn. (2011)
3. Treiber, R.: System programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center (1986)
4. Giacomoni, J., Moseley, T., Vachharajani, M.: FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 43–52. ACM (2008)

5. Preud'homme, T., Sopena, J., Thomas, G., Folliot, B.: BatchQueue: Fast and Memory-Thrifty Core to Core Communication. In: 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 215–222 (2010)
6. Lee, P.P.C., Bu, T., Chandranmenon, G.: A lock-free, cache-efficient shared ring buffer for multi-core architectures. In: Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2009, pp. 78–79. ACM, New York (2009)
7. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, pp. 267–275. ACM (1996)
8. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 134–143. ACM (2001)
9. Gidenstam, A., Sundell, H., Tsigas, P.: Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency. In: Proceedings of the 14th International Conference on Principles of Distributed Systems, pp. 302–317 (2010)
10. Sundell, H., Tsigas, P.: Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In: Proceedings of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS), pp. 84–94. IEEE Press (2003)
11. Cederman, D., Gidenstam, A., Ha, P., Sundell, H., Papatriantafilou, M., Tsigas, P.: Lock-free Concurrent Data Structures. In: Pllana, S., et al. (eds.) Programming Multi-core and Many-core Computing Systems. John Wiley & Sons (2012)
12. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time kd-tree construction on graphics hardware. ACM Transactions on Graphics 27(5), 1–11 (2008)
13. Zhou, K., Gong, M., Huang, X., Guo, B.: Data-Parallel Octrees for Surface Reconstruction. IEEE Transactions on Visualization and Computer Graphics 17(5), 669–681 (2011)
14. Lefohn, A.E., Sengupta, S., Kniss, J., Strzodka, R., Owens, J.D.: Glift: Generic, efficient, random-access GPU data structures. ACM Transactions on Graphics 25(1), 60–99 (2006)
15. Cederman, D., Tsigas, P.: On dynamic load balancing on graphics processors. In: Proceedings of the 23rd Symposium on Graphics Hardware, GH 2008, pp. 57–64. Eurographics Association (2008)
16. Stuart, J., Owens, J.: Efficient synchronization primitives for gpus. Arxiv preprint arXiv:1110.4623 (2011)
17. NVIDIA: Whitepaper NVIDIA Next Generation CUDATM Compute Architecture: FermiTM. 1.1 edn. (2009)
18. NVIDIA: Whitepaper NVIDIA GeForce GTX 680. 1.0 edn. (2012)
19. Lamport, L.: Specifying concurrent program modules. ACM Transactions on Programming Languages and Systems 5, 190–222 (1983)
20. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM 17, 453–455 (1974)