Experimental Fault-Tolerant Synchronization for Reliable Computation on Graphics Processors

Tor-Magne Stien Hagen, Phuong Hoai Ha, and Otto J. Anshus

Department of Computer Science, Faculty of Science, University of Tromsø.
{tormsh,phuong,otto}@cs.uit.no

Abstract. Graphics processors (GPUs) are emerging as a promising platform for highly parallel, compute-intensive, general-purpose computations, which usually need support for inter-process synchronization. Using the traditional lock-based synchronization (e.g. mutual exclusion) makes the computation vulnerable to faults caused by both scientists' inexperience and hardware transient errors. It is notoriously difficult for scientists to deal with deadlocks when their computation needs to lock many objects concurrently. Hardware transient errors may make a process, which is holding a lock, stop progressing (or crash). While such hardware transient errors are a non-issue for graphics processors used by graphics computation (e.g. an error in a single pixel may not be noticeable), this no longer holds for graphics processors used for scientific computation. Such scientific computation requires a fault-tolerant synchronization mechanism. However, most of the powerful GPUs aimed at high-performance computing (e.g. NVIDIA Tesla series) do not support any strong synchronization primitives like test-and-set and compare-and-swap, which are usually used to construct fault-tolerant synchronization mechanisms.
This paper presents an experimental study of fault-tolerant synchronization mechanisms for NVIDIA's Compute Unified Device Architecture (CUDA) without the need of strong synchronization primitives in hardware. We implement a *lock-free* synchronization mechanism that eliminates lock-related problems like the deadlock and, moreover, can tolerate process crash-failure. We address the experimental issues that arise in the implementation of the mechanism and evaluate its performance on commodity NVIDIA GeForce 8800 graphics cards.

1 Introduction

Graphics processors (GPUs) are now considered the most powerful computation hardware available for the price. Contemporary commodity GPUs can theoretically achieve up to 624 GFLOPS (e.g. NVIDIA's GeForce 8800 GTS (G92)) [9], more than triple the throughput of the fastest supercomputer in the world about a decade ago [1] and about thirty times of the throughput of current commodity dual-core CPUs [5]. GPU computational power doubles every ten months, surpassing Moore's Law for traditional microprocessors. This results from the fact that GPUs are specialized for computation-intensive, highly-parallel, graphics computations, and thus GPUs devote more transistors to data processing than to data caching and flow control as in CPUs. As a result, GPUs are emerging as a promising platform for highly parallel, compute-intensive, general-purpose computation.

However, unlike graphics computation, general-purpose computation usually needs support for reliability and inter-process synchronization. Errors in computation domains like radiology, in which GPUs are used for medical image processing, are very costly and potentially harmful to people. Although hardware transient errors in logic have not occurred frequently, such errors are expected to become a significant problem within the next five years [8]. Such hardware transient errors are a non-issue for graphics processors used by graphics computation (e.g. an error in a single pixel may not be noticeable) [6], but this no longer holds for graphics processors used for general-purpose computation. Realizing the problem, researchers have recently proposed a hardware redundancy and recovery mechanism for reliable computation on GPUs [7].

In this paper, we try to address the GPU reliability issues at the software layer. In particular, we are looking at fault-tolerant synchronization mechanisms. Using the traditional lock-based synchronization (e.g. mutual exclusion) makes the computation vulnerable to faults caused by both scientists' inexperience and hardware transient errors. It is notoriously difficult for scientists to deal with deadlocks when their computation needs to lock many objects concurrently. Hardware transient errors may make a process, which is holding a lock, stop progressing (or crash). However, most of the powerful GPUs aimed at high-performance computing (e.g. NVIDIA Tesla series) do not support any strong synchronization primitives like test-and-set and compare-and-swap, which are usually used to construct fault-tolerant synchronization mechanisms. These facts motivated us to carry out an experimental study of fault-tolerant synchronization mechanisms for NVIDIA's Compute Unified Device Architecture (CUDA) without the need for strong synchronization primitives in hardware. We implemented a *lock-free* synchronization mechanism that eliminates lock-related problems like the deadlock and, moreover, can tolerate process crash-failure. The correctness of the synchronization mechanism has been theoretically analyzed in [3] and thus in this paper we concentrate on the practical aspect of the mechanism's implementation on commodity graphics processors. We employ the CUDA capability of reading and writing several words to/from global memory in one instruction called *coalesced memory accesses* [2] to establish an agreement between warps of threads. This agreement is fundamental to constructing fault-tolerant synchronization primitives for threads running on different processors.

This paper makes two contributions: (i) a CUDA kernel supporting fault-tolerant synchronization mechanisms in global and shared GPU memory, and (ii) a performance comparison between software and hardware support of fault-tolerant synchronization.

2 The Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) is the latest GPGPU technology from NVIDIA [2]. It enables developers to write programs to be executed on a GPU without first mapping them to a graphics API. CUDA improves memory access by giving the programmer full read/write support to the entire device memory with some minor exceptions. From the programmers perspective, the GPU can be seen as a set of highly parallel multi-core processors. Each processor is capable of running multiple threads in parallel in a SIMD fashion.

The CUDA memory architecture comprises several memory layers. Each processor has 4 different types of on-chip memory:

– Each core has their own set of on-chip registers
– Each processor has on-chip shared memory which is shared by the processor cores. Reads and writes to the shared memory are serialized in case of bank conflicts
– Each processor has an on-chip read-only constant cache
– Each processor has an on-chip read-only texture cache

The processors share device memory, which is divided into global memory, texture memory and constant memory. Reads from texture and constant memory are cached using the on-chip, read-only, constant and texture cache. Global device memory is not cached.

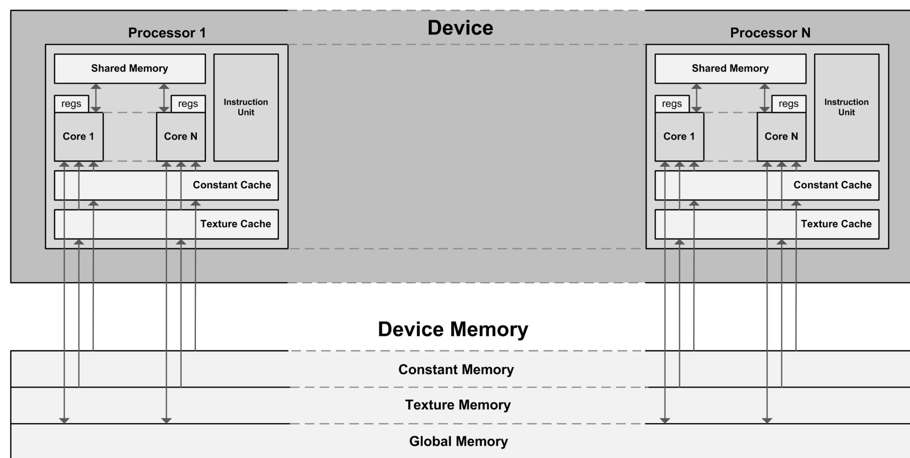

Fig. 1. The Compute Unified Device Architecture

A CUDA compiled program is referred to as a kernel. The kernel is organized as a grid of thread blocks. These thread blocks are organized into batches and executed on the processors. A block is processed by only one processor to maximize the utilization of the shared memory. A block is split into SIMD groups of threads called warps and each of the threads within the warp is executed on the processor cores in a SIMD fashion. The warps of a running block are time-sliced to maximize the utilization of the processor. The way a block is split into warps is always the same, but the issue order of the different warps is undefined. The time-slicing is hardware scheduled, yielding little overhead for context switches. Threads within the same block can communicate using the processors' on-chip shared memory.

NVIDIA uses the term "Compute Capability" to separate the different architectures of their CUDA cards. A card's compute capability is defined by a major revision number and a minor revision number. Devices of the same major revision number have the

same core architecture. The minor revision number corresponds to minor updates to the core architecture. Currently, there are two compute capabilities, 1.0 and 1.1. The major difference between these two capabilities is that cards with compute capability 1.1 have atomic operations for global memory. Cards with compute capability 1.0 have no safe way of communicating through global memory. This implies that processors on a card with compute capability 1.0 can't communicate in a safe manner. However, access to global memory has one property that can be used to create synchronization primitives. The device is capable of reading and writing 32-bit, 64-bit and 128 bit words to or from global memory in one step. This requires the variable type to be a multiple of 4, 8 or 16, and the read or write instructions must be arranged so that the memory accesses can be coalesced into a single contiguous, aligned memory address [2].

3 Design and Implementation

The algorithm used to create synchronization primitives is based on previous work [3]. The mechanisms in the algorithm are based on a long-lived consensus, which is used to achieve an agreement between participants. The algorithm has been theoretically proven in previous work [3]. We now apply the mechanisms of the algorithm to the Compute Unified Device Architecture.

For global memory, we utilize the property of coalesced memory access to establish an agreement between threads of warps running in different blocks. By arranging warp writes from each block as described in the *LongLivedConsensus* [3] (shown in figure 2), we can establish an agreement about a common winner. For shared memory, the writes do not need to fulfill the requirement of coalesced memory access, but memory writes do need to be arranged according to the *LongLivedConsensus*. From the properties of the long-lived consensus we construct a read-modify-write (RMW) object that encapsulates a memory region that different warps can communicate through.

The operation RMW(X, f), where X is a shared variable and f is a mapping, is defined to be equivalent to the indivisible execution of the following function [4]:

function RMW(X, f)
  begin
    $temp \leftarrow X;$
    $X \leftarrow f(X);$
    return temp;
  end

The RMW object is implemented by combining the *LongLivedConsensus* algorithm with a round numbering scheme. Basically, a warp that invokes an operation on the RMW object is assigned to a round. If more than one warp invokes the RMW object within the same round, the *RMW* algorithm [3] ensures that all warps will agree upon a common sequence of accesses, and thereby ensure the integrity of the RMW object. Each of the warps belonging to the same round suggests an order of accesses in that round. The *LongLivedConsensus* algorithm is used to achieve an agreement on the order to use. All warps that invoke the RMW object will pass a function and some arguments to the RMW object. The function and arguments of each warp that invokes the RMW

object is written to the warp's part of a shared memory location. This memory location is readable from all warps that invoke the RMW object. Each warp reads the function and parameters from all other warps that participate in the round, calculates a value of the RMW object based on its own sequence of the functions, and then writes the result to a known memory location that can be read by every participating warp. Then the warp invokes the *LongLivedConsensus* using the memory location of its proposal. For global memory, five threads of a half-warp in each block write to global memory in one coalesced memory operation. For shared memory, the first sixteen threads of a warp (the first half-warp) write to shared memory. After the writes, each memory entry is compared to the others in order to find the warp that wrote first. This is done using the *Ordering* algorithm in combination with a *RoundCheck* algorithm described in [3]. After the execution of the *Ordering* algorithm, the warp that wrote first will be known by all warps participating in that round. All warps then accept the buffer as the sequence of functions for that round. Since each of the warps executes one function on the RMW object at a time, functions are ordered according to both the round they participate in and the order agreed upon by warps in the same round.

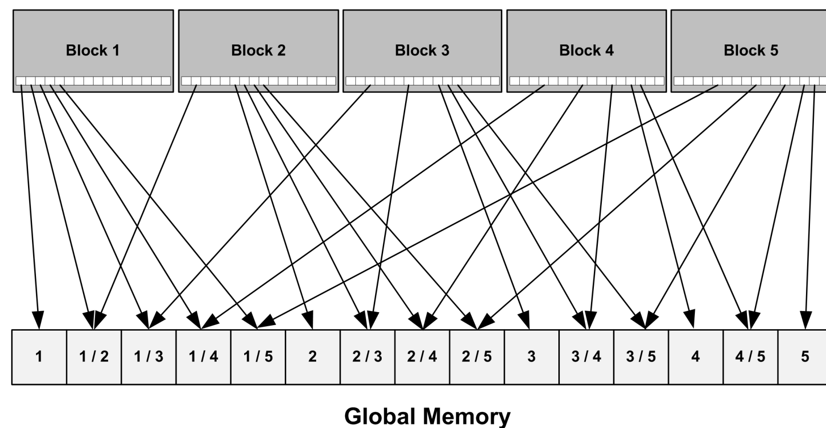

Fig. 2. The coalesced access pattern to global memory from each of the processor warps. Each processor has 5 active threads that fulfills the requirement of coalesced memory access

The current implementation of the algorithm supports wait-free synchronization between five blocks of warps using global memory and sixteen warps using shared memory, independent of the device compute capability. However, the actual number of warps synchronized through shared memory is limited to five because the data structures that encapsulate the RMW object consume too much on-chip memory when the number of warps exceeds this limit. This will be optimized for future implementations of the algorithm. The algorithm is designed for an asynchronous memory model. For CUDA, the access speed to shared memory is the same as for registers if no bank conflicts occur [2]. For this reason, the shared memory version contains many duplicates that can

be removed in future optimizations of the implementation. However, the duplicates are kept for the current implementation to ensure correctness.

The RMW object supports any read-modify-write operation such as the atomic operations in graphics cards with compute capability 1.1. That is: ADD, SUB, EXCH, MIN, MAX, INC, DEC and CAS. In addition, the RMW object supports atomic operations on floating point numbers, which currently is not supported by any CUDA card of any compute capability. Further, the RMW object guarantees that the atomic operation is wait-free if the number of failing warps is less than or equal to four for global memory and less than or equal to fifteen for the shared memory version. This ensures that no failure or corruption can make the atomic operation hang.

4 Experiments

4.1 Methodology

To evaluate the RMW object implemented in both global and shared memory we have conducted two experiments.

The first experiment was conducted to determine the extra overhead involved using software synchronization in global memory. For this experiment the RMW object was invoked 30 000 times recording the time for all iterations to complete. We compared the results with a graphics card with compute capability 1.1 to evaluate the additional overhead of software synchronization in global memory. We used the atomic operation atomicAdd for hardware support. The experiment was repeated with 1 to 5 blocks, each block having 16 threads. Each of the configurations was repeated 10 times.

For the second experiment we compare synchronization between hardware support in global memory and software support in shared memory. For this experiment the RMW object is invoked 30 000 times recording the time for all iterations to complete. We repeated the experiment for 1 to 5 warps, and each of the configurations was repeated 10 times.

The hardware used in the experiments was: (i) a NVIDIA Geforce 8800GT PCX graphics card with CUDA compute capability 1.1, and (ii) a NVIDIA Geforce 8800GTS graphics card with CUDA compute capability 1.0. Each graphics card is powered by an Intel Pentium 4 EM64T 3.2 GHz computer with 2 GB of RAM. The Geforce 8800GT card is used as a reference for the atomic operations in hardware. Both graphics cards are used to evaluate the software implementation.

4.2 Results

Figure 3 shows the time used to invoke the RMW object in global memory compared to hardware support. As the figure illustrates hardware support is an order of magnitude faster than software support. For 1 block, the atomic operation in hardware uses 0.0124 seconds to complete. For the Geforce 8800GT card, the time to invoke the RMW objects is 0.89 seconds and the Geforce 8800GTS card uses 0.935 seconds. For 5 blocks, atomic operation in hardware takes 0.062 seconds for 30 000 iterations. For software synchronization, the time is 2.38 for the 8800GT card and 2.56 for the 8800GTS card.

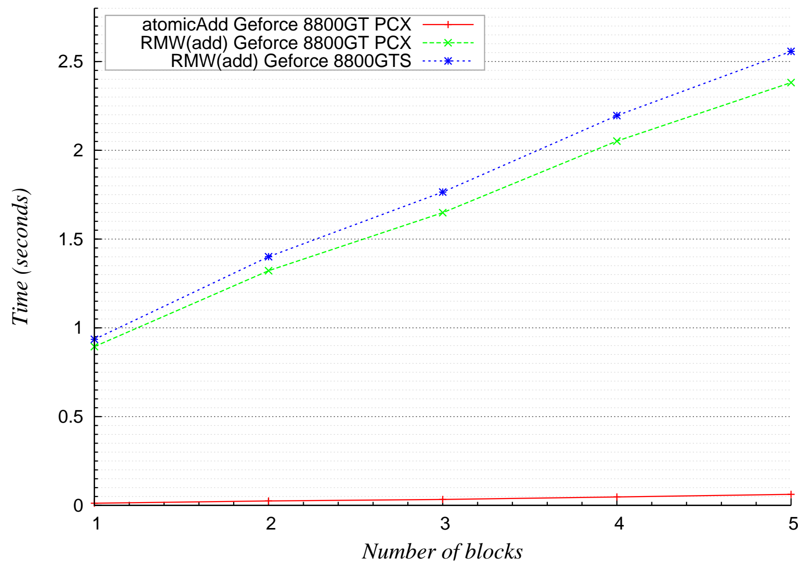

Fig. 3. The time used for 30 0000 invocations of the RMW object in global memory compared to atomic support in hardware (global memory)

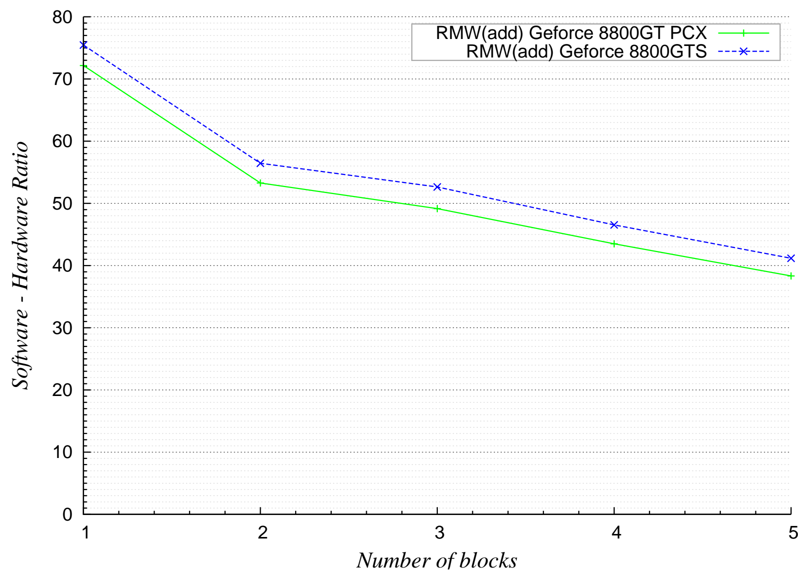

Fig. 4. The ratio between synchronization in software and hardware (global memory)

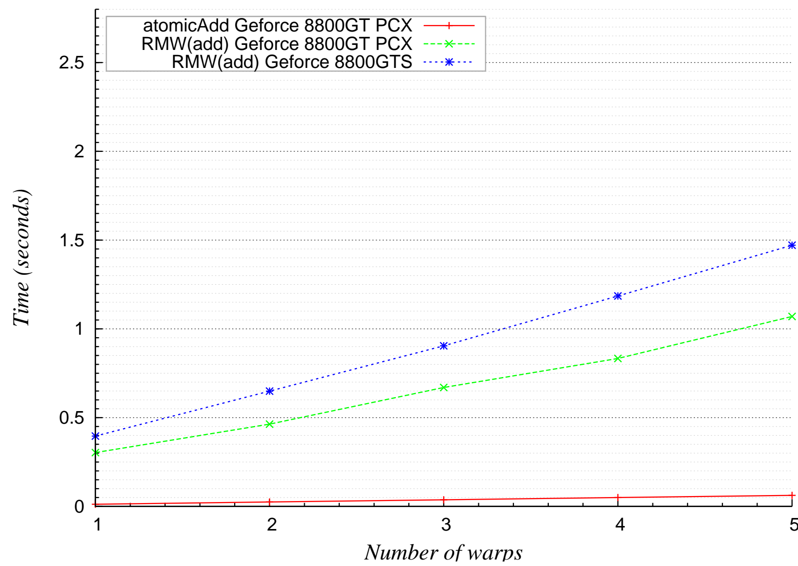

Fig. 5. The time used for 30 000 invocations of the RMW object in shared memory compared to atomic support in hardware (global memory).

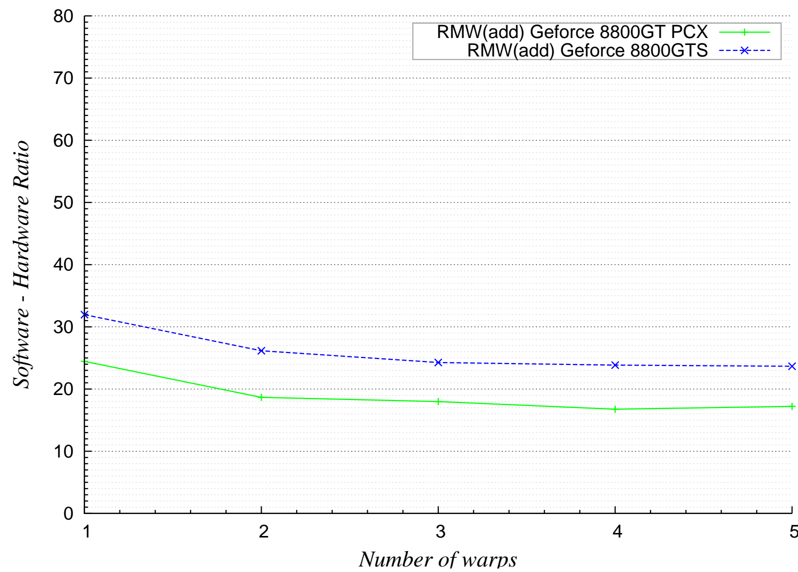

Fig. 6. The ratio between software synchronization support in shared memory and hardware support in global memory

The figure indicates a linear increase in time for both software and hardware support as the number of blocks increase.

Figure 4 shows the ratio between software support in global memory and hardware support in global memory. The graph is generated by dividing the time for software synchronization by the time used for hardware support. For 1 block, hardware support is 72.16 times faster than software support for the 8800GT card and 75.45 times faster for the 8800GTS card. However, for 5 blocks this factor has decreased to 38.33 for the 8800GT card and 41.17 for the 8800GTS card. The graph indicates that the software to hardware ratio decreases as the number of blocks increase.

Figure 5 shows the time used to invoke the RMW object in shared memory compared to hardware support in global memory. For 1 warp, the hardware supported atomic operation uses 0.01239 seconds to complete compared to 0.303 seconds for the 8800GT card and 0.396 for the 8800GTS card. For 5 warps, the atomic operation in hardware uses 0.0621 seconds and the shared memory version uses 1.07 seconds for the 8800GT card and 1.471 seconds for the 8800GTS card. The time to do synchronization in shared memory is greater than hardware support in global memory for all warp configurations. This indicates that the computation steps of the RMW object is the factor limiting the speed of the software synchronization, as accessing shared memory is two orders of magnitude faster than global memory.

Figure 6 shows the ratio between software support using shared memory and hardware support in global memory. The graph is generated by dividing the shared memory software synchronization time by the time for hardware support in global memory. For 1 warp, the factor is 24.46 for the 8800GT card and 31.95 for the 8800GTS card. For 5 warps, the factor has decreased to 17.22 for the 8000GT card and 23.67 for the 8800GTS card. As opposed to software synchronization in global memory, software synchronization in shared memory seems to flatten out between four and five warps.

5 Conclusion

We have presented an experimental study of a fault-tolerant synchronization mechanism for NVIDIA's Compute Unified Device Architecture without the need for strong synchronization primitives in hardware. The current prototype has been implemented as a proof of concept showing that read-modify-write objects can be constructed and mapped to the architecture of CUDA. The RMW object enables processors of CUDA graphics cards with compute capability 1.0 to communicate through global memory and graphics cards with either compute capability 1.1 or 1.0 to use atomic operations through shared memory. The current version of the RMW object implemented in global memory supports five blocks, and the synchronization is guaranteed to be wait-free for up to four failing blocks. For the shared memory version the synchronization primitive has a limit of sixteen warps. However, the amount of on-chip shared memory reduces the actual warp number to five.

The experiments conducted indicate that the performance bottleneck of the RMW objects are the computation steps needed to ensure consensus between the participating blocks and warps. The hardware implementation is an order of magnitude faster than the software implementation. However, as the number of blocks (global memory) and

warps (shared memory) increases from one to five, the performance gap between software and hardware is reduced from 72.16 times to 38.33 times for global memory, and 24.46 times to 17.22 times for shared memory. We are working on improvements to the implementation of the algorithm in order to increase its performance and allow for more blocks and warps for both global and shared memory.

6 Acknowledgements

The authors wish to thank Elizabeth Jensen and Tore Larsen for discussions, as well as the technical staff at the CS department at the University of Tromsø. Supported by the Norwegian Research Council, projects No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and No. 155550/420 - Display Wall with Compute Cluster.

References

1. *TOP500 List*. TOP500.ORG, 1994.
2. *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.1*. NVIDIA Corporation, 2007.
3. P. H. Ha, P. Tsigas, and O. J. Anshus. Wait-free programming for general purpose computations on graphics processors. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, page pages to appear. IEEE Computer Society, 2008.
4. C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, 1988.
5. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
6. J. W. Sheaffer, D. P. Luebke, and K. Skadron. The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, pages 9–16, 2006.
7. J. W. Sheaffer, D. P. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 55–64, 2007.
8. J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 177, 2004.
9. S. Wasson. *Nvidia's GeForce 8800 GTS 512 graphics card*. http://techreport.com/articles.x/13772, 2007.