Argus-G: A Low-Cost Error Detection Scheme for GPGPUs

Ralph Nathan
Dept. of ECE
Duke University
Durham, NC, USA
rn19@duke.edu

Daniel J. Sorin
Dept. of ECE
Duke University
Durham, NC, USA
sorin@ee.duke.edu

ABSTRACT

We have developed Argus-G, a low-cost error detection mechanism for the SIMT cores found in GPGPUs. As GPUs make the transition into general purpose computing, detecting errors and dealing with them will become a more pressing issue. General purpose graphics processing units are increasingly used for scientific computing, where errors, if not detected, can significantly distort the results of these scientific simulations. Argus-G is an adaptation of the Argus error detection scheme for general purpose cores that has been tailored for GPUs. Our e*xperiments show that, on average, our implementation incurs a 4% overhead in runtime and a 10% increase in the number of instructions executed.*

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance**;** I.3.1 [**Computer Graphics**]: Hardware Architecture – *graphics processors.*

General Terms

Performance and Reliability.

Keywords

Error detection, GPU, GPGPU, fault tolerance, computer architecture.

1. Introduction

As GPUs complete their transition into the general purpose computing space, detecting errors and dealing with them will become a more pressing issue. Due to technology scaling, transistors have been decreasing in size, thereby increasing the chance of faults developing [6]. In addition, the die sizes of GPGPUs have been increasing. Having a greater number of smaller transistors increases the probability of a transient or a permanent fault.

In the past, when GPUs were primarily used for graphical applications, there was no demand for error detection mechanisms. At the worst, an error would affect a pixel or two on the screen. But, since GPUs have now transitioned into the general purpose computing space, including high performance scientific computing, faults can significantly distort the results of the scientific simulations run on these systems. Thus, error detection for General Purpose GPUs (GPGPUs) is now a pressing concern for architects.

To the best of our knowledge, the only scheme that is currently in use, or that has been suggested, is the one provided by Nvidia on the Fermi series on GPGPUs. Nvidia uses Error Correcting Codes [21] (ECC) to detect and fix soft errors in the register files and the memory system. However, ECC does not detect errors in the logic, including the numerous functional units and control logic.

In order to remedy this problem, we propose Argus-G, an error checking mechanism designed for the Single Instruction Multiple Thread (SIMT) cores present in the current generations of Nvidia and ATI GPGPUs. Argus-G is an implementation of the Argus [16] error detection scheme, adapted to be compatible with SIMT cores. Argus-G detects errors in the computation of results [27, 22, 23, 18], control flow [6, 12, 28] and the data flow [15] of programs run on GPGPUs. Detecting errors avoids silent data corruption and, if combined with an error recovery mechanism, can enable transparent fault tolerance.

The primary contribution of this paper is a low-cost mechanism for detecting errors in the SIMT cores present in GPGPUs. Although the Argus approach has already been proposed for general purpose cores, our application and adaptation of Argus to GPUs is novel. We experimentally evaluated the feasibility of this scheme using GPGPU-Sim [4], a simulator that models CUDA [20] capable GPUs.

In Section 2, we present an overview of the Argus framework and how it has been implemented for general purpose cores. In Section 3, we describe the Argus-G implementation. In Section 4, we present our experimental evaluation of Argus-G. Section 5 presents the related work, and in Section 6 we discuss future work. We conclude in Section 7.

2. Argus Overview

This section describes the Argus methodology (Section 2.1) and how it has been implemented for general purpose cores (Section 2.2). The implementation of Argus is core-specific, and we will present our Argus-G implementation for GPGPUs in Section 3.

2.1 Argus Framework

The actions performed by von Neumann machines can be decomposed into three basic activities: choosing the sequence of instructions to execute (“control flow”), performing the computation specified by each instruction (“computation”), and passing the result of the computation to other data-dependent instructions (“dataflow”). Checking the computation, control flow and dataflow is a provably complete method of detecting errors in cores. We now discuss the requirements of the three checkers present in any Argus implementation.

WRA '10, December 4–8, 2010, Atlanta, Georgia, U.S.A.

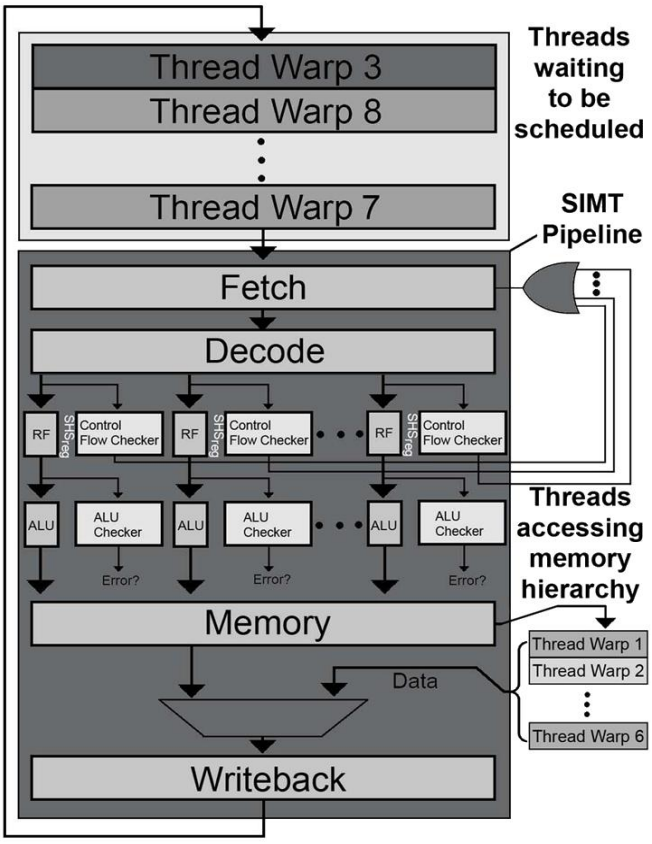

Figure 1: SIMT Pipeline with the Argus-G Checking Methodology.

Computation Checker: A computational checker detects errors in the functional units. The way in which these checkers are implemented depends on the type of the functional unit. Many checkers can use knowledge about the initial result to simplify the checking logic. Sellers et al. [27] provide a survey of checkers for adders, multipliers, dividers, bit-wise logic units, etc. Computation checking is a well-known problem with well-studied solutions.

Control Flow Checker: The control flow checker verifies that the dynamic (runtime) execution path of the program is valid with respect to its static control flow graph. However, if the two graphs differ, then an error has occurred. The control flow checker detects errors in the instruction fetch unit, the branch destination computation and the PC update logic. One limitation of control flow checking is that it cannot detect whether the core has incorrectly entered one of two possible control flow paths at a branch instruction. This coverage hole is eliminated when we add a dataflow checker and a computation checker.

Dataflow checker: The dataflow checker verifies that the runtime dataflow is the same as that specified in the program's binary. The dataflow checker detects errors in the fetch, decode, and register read/write units.

2.2 Argus-1 Implementation

Argus-1 [16] is a low-cost error detection implementation of the Argus scheme for simple, in-order, general purpose cores. Argus-1 includes checkers for computation, control flow, and dataflow. At first, one might assume that the Argus-1 implementation would be appropriate for GPGPUs. However, we would not want to use Argus-1 to detect errors for GPGPUs for cost reasons. Even though Argus-1 could detect the same errors for a single-threaded pipeline that Argus-G would detect for a SIMT pipeline, Argus-1 would not be able to take advantage of the SIMT nature of GPGPUs, leading to an increase in hardware costs.

3. Argus-G Implementation

In this section, we describe an implementation of Argus adapted to the SIMT pipelined cores present in GPGPUs, called Argus-G. Argus-G is a low-cost error detection mechanism for GPGPUs.

3.1 Baseline GPGPU Configurations

In this section, we describe the system model for which we designed Argus-G. A GPGPU has the ability to execute hundreds of threads simultaneously. A GPGPU has many shader cores, each of which has a SIMT pipeline. The width of the SIMT pipeline depends on the specific architecture of the GPGPU. The SIMT pipeline consists of a shared fetch stage, but each thread in the SIMT pipeline has its own register files and Stream Processors (SP). The integer multiply and divide units and all the FP units are shared between the threads. GPGPUs have five different memory address spaces: local, shared, global, constant, and texture. Argus-G has to check that loads and stores from these memories are indeed from the correct memory space. The fact that GPGPUs have multiple memory types adds another level of complexity over simple cores that the checking mechanism has to consider.

Argus-G is implemented at the SP level where each thread context needs checking. Argus-G performs these checks at the SPs due to the fact that errors in control flow and dataflow occur at the thread granularity. In a system where the width of the SIMT pipeline is n, we require n Argus-G check mechanisms (see Figure 1).

The simulator we use, GPGPU-Sim [4], simulates Parallel Thread Execution (PTX) instructions. PTX [20] is an intermediate, pseudo-assembly language used by Nvidia's CUDA programming environment [19]. PTX is generated when CUDA code is compiled, but when the binary is run on the GPGPU, it is converted just-in-time (JIT) into the native model specific assembly language (see Figure 2). Having no direct translation of the instructions into bits allows Nvidia to have PTX run on multiple GPGPU architectures. PTX also has virtual registers so there is no register file size limit, again allowing PTX to be ported to multiple architectures.

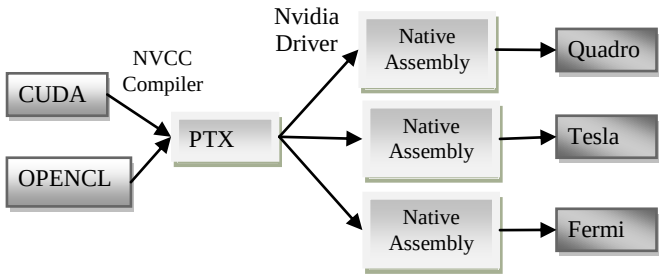

Figure 2: The CUDA Chain from High Level to Assembly

3.2 Control Flow and Dataflow Checkers

The control flow and dataflow checkers are based on the work done by DDFV [15]. Even though DDFV was developed for superscalar cores, it can be adapted to work for SIMT cores. The initial Argus-1 implementation demonstrated that DDFV could be streamlined to work with simple, single-threaded cores. We extended the concepts that were proposed by DDFV and Argus-1 and adapted them to make them compatible with the SIMT cores in GPGPUs.

Argus-G detects errors in the core's dataflow by comparing the static dataflow graph (DFG) specified by the program to the dynamic DFG computed by the processor during execution. Both DFGs are represented using constant-sized signatures. The static signatures are computed during the translation process from PTX to the native assembly, which occurs just before runtime (PTX is translated JIT to the native assembly code). To avoid problems with data dependent branches, which can dynamically alter the DFG, Argus-G performs checks at the granularity of basic blocks. However, some errors may not be detected due to aliasing (i.e., having multiple basic blocks with the same signature), though the chance of this occurring can be diminished by increasing the size of the signatures.

The mechanism described in the previous paragraph checks the dataflow, and it implicitly checks the control flow *within* a basic block, but not the control flow *between* basic blocks. To provide full control flow checking, we add another mechanism over the basic dataflow checker. We use a block's basic dataflow signature as both a representation of the block's internal dataflow, and as a unique, address-independent block identifier that can be used for full control flow checking. This signature is called the dataflow and control flow signature (DCS). Argus-G embeds into each basic block the signatures of its legal successor basic blocks. Then at runtime, if an error occurs, the error will be detected (barring aliasing), because the block's runtime DCS will not match the signature computed statically at compile time.

Figure 3 illustrates how Argus-G embeds the DCS for basic blocks with one or two successor basic blocks. The DCS is embedded in a NOP instruction. A basic block that has only one legal basic block after it contains only that basic block's signature (e.g., the signature embedded in BB3 is that of BB4). However, a basic block that has two legal successor basic blocks must contain the signatures of both those basic blocks (e.g., in basic block BB1, we embed the signatures of basic blocks BB2 and BB3).

DCS Computation: We compute the DCS similarly to the way in which DDFV computed the dataflow signature and Argus-1 computed the DCS. We maintain a State History Signature (SHS) for each architectural location, i.e., we have a SHS for each register (SHS_{reg}), program counter (SHS_{pc}), and memory (SHS_{mem}). Since GPGPUs have five different types of memory Argus-G needs five different SHS_{mem}'s, one for each type of memory. This is one of the differences between the original Argus-1 implementation and Argus-G. One scenario that Argus-G currently does not address is if multiple threads concurrently access the texture or the shared memories. We plan to address this in future work, but for now we simplistically assume that concurrent accesses to texture or shared memories are either disallowed or temporarily disable checking of the SHSs for these memories.

A SHS for a particular location represents the history of that particular location from the start of the current basic block. The SHS of each location is reset to its specific initial value whenever a basic block is found to have executed correctly, i.e., has succeeded its legal parent block and has its computed DCS match the embedded DCS. Each SHS has to be maintained for all previous mentioned locations in each lane of the SIMT pipelines. The SHS of a particular destination location depends on the operation that produced it, as well as the operand registers used. When an instruction is executed, for example, *add.u32 %r3, %r2, %r1*, the new SHS of destination register %r3 depends on $SHS_{\%r2}$, $SHS_{\%r1}$, and the fact that the operation was an *add*.

In order to detect whether the dynamic dataflow graph for memory operations was the same as the static dataflow graph, we assign an initial value to SHSs of the various memory address spaces. When a load instruction is executed, the SHS of the destination operand will depend on the memory space type, and the address type, whether the address was a constant, from a register, or from a register with an offset added to it. The dataflow checker does not, however, check the value of the offset, the immediate address, or the final computed address. These values are checked by the computation checker. For example, when the instruction *load.const.s32 %r4, [Mem_Address1]* is executed, the new SHS of destination register %r4 will depend on its previous value, $SHS_{\%r4}$, SHS_{CONST_MEM}, and the fact that the address was an immediate. The SHS of the PC, SHS_{pc}, is written to by jumps and branches, both of which are present in the CUDA architecture. We deal with stores in a similar manner, as we incorporate the fact that the store occurred and to which address space it occurred, but the DCS does not check to see if the value stored was correct.

Signature Size: The size of the DCS and the SHSs should allow a unique value for each of the registers, PC, and memory spaces. In addition, the DCS should be small enough to be easily embedded into the binary. The size of the DCS depends on the number of free bits available in the NOP instructions used for embedding the signatures. Because we do not have access to the actual architectures, our method of performing the computation of the static signatures is different from how we would have approached it if we had access to the actual assembly code run on the GPGPU.

Embedding Signatures: Signature instructions embedded into the program cause performance degradation because they increase the pressure on the instruction cache and also consume processor cycles. These signatures were added using NOPs in PTX as we are not able to access the actual architecture. Had it been available to us, we could have amortized the costs of adding NOPS by embedding the DCS in other instructions with unused bits.

PTX Assembly Code with Signatures Embedded

```
BB1:   cvt.u32.u16   %r1, %tid.x;
       add.u32       %r2, %r1, %r1;
       setp.gt.s32   %p1, %r1, %r1;
       Signature     {BB2, BB3}
       @%p1bra       $BB3; // branch instruction
BB2:   add.u32       %r1, %r1, 17;
       Signature     {BB4}
       bra.uni       BB4; // unconditional jump
BB3:   mul.wide.u16  %r3, %r1, %r2
       Signature     {BB4}
BB4:   ld.const.s32  %r4, [Mem_Address1];
```

Figure 3: DCS Embedding

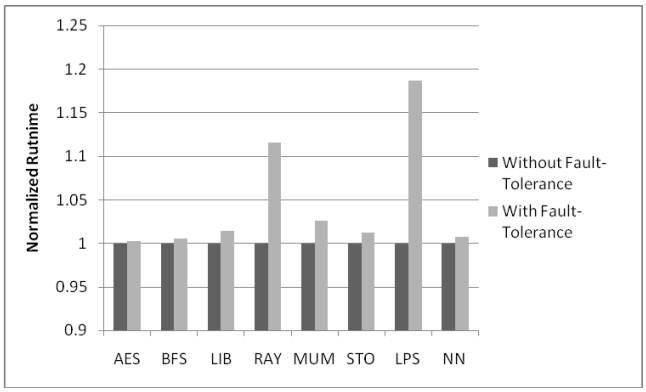

Figure 4(a): Increase in Normalized Runtime with Argus-G

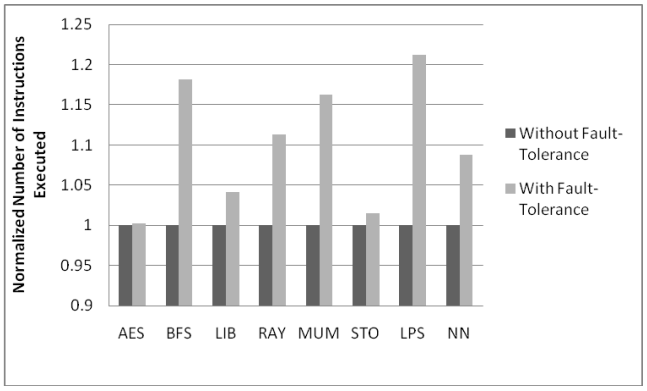

Figure 4(b): Increase in the Number of Instructions Executed

Because the CUDA programming environment mandates that PTX assembly is translated JIT to the native assembly, Argus-G can work with legacy code and the programmer can make optimizations at the PTX level, not just at the source code level. We can then embed the signatures during the conversion process from PTX to the native assembly.

3.3 Computation Checker

There are many well-known methods to check the results of the functional units [27]. The choice of computation checkers is a matter of trading off cost (area and power) versus error detection coverage. Because computation checking is, in general, the same for GPGPUs as it is for general purpose cores, we do not discuss it further here. We only note that GPUs may have some functional unit types not found in general purpose cores, and these functional units would require different checkers than those used in general purpose cores.

3.4 Error Detection Coverage

Argus is a provably complete methodology for detecting errors, although implementations of Argus may miss some errors due to the use of imperfect checkers (e.g., modulo checking of a multiplier is cost-effective but imperfect). The Argus-G implementation of Argus detects errors, both transient and permanent, throughout most of the chip. Because Argus-G relies on a lossy signature of dataflow graphs, the DCS, there is some probability of aliasing and thus missing an error. The computation checkers are also likely to use lossy checking, such as modulo checking, and thus would also miss some errors. Lastly, we have not yet extended Argus-G to detect errors outside the cores, including errors in the interconnect between the various shader cores and errors in the transmission of data between the GPU and the host CPU.

Table 1: List of Benchmarks

Benchmarks	Description
AES [13]	AES cryptography
BFS [11]	Search in a Large Graph Algorithm
LIB [10]	Monte Carlo Simulations
LPS [9]	3D Laplace Solver
MUM [26]	DNA Sequence Alignment
NN [5]	Neural Network
RAY [14]	Ray tracing
STO [1]	Distributed Storage Systems Acceleration

3.5 Responding to Detected Errors

The current implementation of Argus-G naively deals with detected errors by clearing all registers and restarting the thread in which an error was found. A better approach would be to leverage the nature of the CUDA programming environment's hierarchy of threads, warps, and programs. If the thread has modified the memory space shared by threads within that warp, then the entire warp would be restarted since we cannot guarantee that other threads in that warp did not access the corrupted data. If the thread modified the global memory, then the entire program will be restarted. One could avoid restarting by adding a memory checkpoint/recovery scheme, but it is not clear that the costs of doing this would exceed the benefits.

If the detected error is due to a permanent fault, the error will recur after restarting. To diagnose this situation, we can add a counter to each core that is incremented when Argus-G detects an error in that core. If the counter exceeds a threshold, that core would be deemed permanently faulty.

4. Experimental Evaluation

The goal of this evaluation is to determine the performance overheads of Argus-G. We are not evaluating error detection coverage, area overhead, and power costs since we do not yet have a realistic hardware model of the GPU core; we are currently developing this hardware to enable a more thorough evaluation of Argus-G.

4.1 Evaluation Methodology

We evaluated Argus-G using GPGPU-Sim [4], a cycle-accurate GPGPU simulator. The system modeled by GPGPU-Sim is similar to that of the Nvidia Quadro FX5800, a CUDA capable GPGPU. The Nvidia Quadro FX5800 has 30 SIMT shader cores and each shader core has 8 SPs. As previously mentioned, the complex integer arithmetic units, as well as, all the floating point units are shared between the threads.

We use scientific benchmarks that are prevalent in the contemporary general purpose GPU computing space. A full list of these benchmarks can be found in Table 1. We examine both the performance overhead, measured in processor cycles, as well as the increase in the number of instructions executed.

4.2 Performance Overhead

Argus-G's performance overheads come from the signature containing instructions embedded into the program's binary. These instructions take up space in the instruction cache and also consume processor cycles when executed. The increase in the runtime, shown in Figure 4(a), is, on average, 4%. The increase in the number of dynamic instructions, shown in Figure 4(b), is, on average, 10%. The difference in these two numbers is due to the parallel nature of GPGPUs, in which up to hundreds of threads can be executing simultaneously. The parallel nature of GPGPUs allows us to amortize the costs of adding the extra NOPs. If we had access to the binary translations of the instructions, we could embed the signatures within the unused bits of other instructions instead of NOPs and thereby further reduce the impact on performance.

Table 2 shows the average size of the basic blocks. When the basic blocks are relatively large (e.g., in *AES*, where the program consists of 4 basic blocks), the impact on performance (0.2%) is negligible. However, when the size of the basic blocks is much smaller (e.g., *RAY)*, the impact on performance (12%) is significant. For benchmarks with long basic blocks, the increase in the static code length is negligible, less than 1% for AES and NN, and the corresponding impact on runtime is also small. But benchmarks with smaller basic blocks need more NOPs inserted and thus affect performance more. However, the impact on performance also depends on the dynamic path selected through the program during runtime and we leave a detailed analysis of this for the future.

Table 2: Average Basic Block Size and increase in code length

Benchmarks	Basic Block Size (In Instructions)	Percentage Increase in Static Code Length
AES	425	0.2
BFS	9	10
LIB	11	3
LPS	12	8
MUM	5	18
NN	105	1
RAY	9	10
STO	23	4

5. Related Work

Although, to the best of our knowledge, no other error detection schemes have been proposed for GPGPUs, other schemes that detect errors in computation, control flow, the memory system and dataflow have been proposed. The schemes that check for control flow and dataflow were primarily targeted at superscalar cores or simpler cores.

Error Correcting Code: ECC [21] is used in the current generation of Nvidia GPGPUs. It is used to correct single-bit soft errors in data storage structures, such as, the register files, shared memories and the caches. To the best of our knowledge, this is the only error detection scheme for GPGPUs. ECC can be used in conjunction with Argus-G to make GPGPUs more fault tolerant than either system could in isolation.

Argus: Argus is a low-cost comprehensive system designed to detect errors in control flow, dataflow, computation and memory accesses. Argus combines dataflow checking and control flow checking into a single mechanism. It was designed to work with simple, in-order cores. As already mentioned, we modified Argus to make it compatible with GPGPUs.

DIVA: DIVA [2, 3, 29] is a heterogeneous DMR scheme that uses a simple, yet architecturally identical core, embedded in the pipeline stage of another core for checking. This system is well suited to high performance, speculative RISC architectures. It can be implemented on GPGPUs, but the costs of doing so will be significant.

Redundant Multithreading: Redundant multithreading [17, 24, 25] can be used to detect errors in GPGPUs without incurring much hardware costs since GPGPUs inherently have the capacity to run redundant threads. The n-way SIMT pipeline can be modified such that only n/2 threads are running each time and the other n/2 lanes are used for the redundant threads. However, in this case there exist significant opportunity costs since the theoretical maximum performance ceiling of the chip would be halved.

Summary: Though some of these prior mechanisms overlap with Argus-G, most of them have obstacles preventing them from ported to Argus efficiently. The key among these obstacles are performance and power impact and the increase in hardware.

6. Future Work

We plan to improve upon the evaluation of Argus-G by obtaining the results for error coverage and the area and power costs. We are currently working on an FPGA-based implementation of a GPGPU in order to thoroughly evaluate these attributes.

7. Conclusion

The goal of this research was to develop a low-cost mechanism to detect errors in GPGPUs. We expect the popularity of GPGPUs to increase with time since they can be used for a wide variety of scientific applications. Therefore, it is important that we detect errors in GPGPUs in order to not distort results of simulations.

The Argus-G implementation shows the potential of employing the Argus methodology in the SIMT cores present in GPGPUs. The Argus-G performance costs are low (on average 4% overhead), but we believe that with additional tuning, we can further decrease the performance costs.

8. References

[1] Al-Kiswany, S., Gharaibeh, A., Santos-Neto, E., Yuan, G., and Ripeanu, M,. 2008. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proc. 17th Int'l Symp. on High Performance Distributed Computing*, pages 165–174, 2008.

[2] Austin, T. M. 1999. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual ACM/IEEE international Symposium on Microarchitecture*.

[3] Austin, T.M. 2000. DIVA: A Dynamic Approach to Microprocessor Verification. In *Journal of Instruction-Level Parallelism*, 2, May 2000.

[4] Bakhoda A., Yuan, G. L., Fung W. W. L., Wong H., and Aamodt T. M. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium*

on Performance Analysis of Systems and Software, April 2009.

[5] Billconan and Kavinguy. A Neural Network on GPU. http://www.codeproject.com/KB/graphics/GPUNN.aspx.

[6] Borkar, S. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. Micro, IEEE **25**(6): 10-16.

[7] Delord, X. and Saucier, G. 1991. Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors. In *Proc. of Int'l Test Conf.*, 1991.

[8] Eibl, P. J., Cook, A. D., and Sorin, D. J. 2009. Reduced Precision Checking for a Floating Point Adder. In *Proceedings of the 2009 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*.

[9] Giles, M. Jacobi Iteration for a Laplace Discretisation on a 3D Structured Grid. http://people.maths.ox.ac.uk/˜gilesm/hpc/NVIDIA/laplace3d.pdf.

[10] Giles, M., and Xiaoke, S. Notes on Using the NVIDIA 8800 GTX graphics card. http://people.maths.ox.ac.uk/˜gilesm/hpc/.

[11] Harish, P. and Narayanan, P. J. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, pages 197–208, 2007.

[12] Kim, S. and Somani, A.K. 2001. On-Line Integrity Monitoring of Microprocessor Control Logic. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors* (September 23 - 26, 2001).

[13] Manavski, S.A. 2007. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *ICSPC 2007: Proc. of IEEE Int'l Conf. on Signal Processing and Communication*, pages 65–68, 2007.

[14] Maxime. Ray tracing. http://www.nvidia.com/cuda.

[15] Meixner, A. and Sorin, D. J. 2007. Error Detection Using Dynamic Dataflow Verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (September 15 - 19, 2007).

[16] Meixner, A., Bauer, M. E., and Sorin, D. 2007. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (December 01 - 05, 2007).

[17] Mukherjee, S. S, et al. 2002. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *Proc. of the 29th Annual Int'l Symp. On Computer Architecture*, p. 99–110, May 2002.

[18] Nicolaidis, M. 1993. Efficient Implementations of Self-Checking Adders and ALUs. In *Proc. of the 23rd Int'l Symp. on Fault-Tolerant Computing Systems*, p. 586–595, June 1993.

[19] Nvidia. 2007. CUDA: Compute Unified Device Architecture, Version 1.0. June 2007.

[20] Nvidia. 2009. Compute PTX: Parallel Thread Execution, Version 1.4. March, 2009.

[21] Nvidia. 2009. Whitepaper. Nvidia's Next Generation CUDA Compute Architecture: Fermi.

[22] Patel, J. H. and Fung, L. Y. 1983. Concurrent Error Detection in Multiply and Divide Arrays. In *IEEE Trans. Comput.* 32, 4 (Apr. 1983), 417-422.

[23] Patel, J.H. and Fung, L. Y. 1982. Concurrent Error Detection in ALUs by Recomputing with Shifted Operands. In *IEEE Trans. on Computers*, C-31(7):589–595, July 1982.

[24] Reinhardt, S. K., and Mukherjee, S. S. 2000. Transient Fault Detection via Simultaneous Multithreading. In *Proc. Of the 27th Annual Int'l Symp. on Computer Architecture*, p. 25–36, June 2000.

[25] Rotenberg, E. 1999. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. of the 29th Int'l Symp. On Fault-Tolerant Computing Systems*, p. 84–91, June 1999.

[26] Schatz, M., Trapnell, C., Delcher, A., and Varshney, A. 2007. High-Throughput Sequence Alignment using Graphics Processing Units. In *BMC Bioinformatics*, 8(1):474, 2007.

[27] Sellers, F.F. et al. 1968. *Error Detecting Logic for Digital Computers*. McGraw Hill Book Company.

[28] Warter, N.J and Hwu, W.-M. W. 1990. A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking. In *Proc. of the 20th Int'l Symp. on Fault-Tolerant Computing Systems*, p.442–449, June 1990.

[29] Weaver, C. and Austin, T. M. 2001. A Fault Tolerant Approach to Microprocessor Design. In *Proceedings of the 2001 international Conference on Dependable Systems and Networks (Formerly: Ftcs)* (July 01 - 04, 2001).

[30] Yilmaz, M., Meixner, A., Ozev, S., and Sorin, D. J. 2007. Lazy Error Detection for Microprocessor Functional Units. In *Proceedings of the 22nd IEEE international Symposium on Defect and Fault-Tolerance in VLSI Systems* (September 26 - 28, 2007).