Software Reliability Enhancements for GPU Applications

Si Li[1], Naila Farooqui[2], and Sudhakar Yalamanchili[1]

[1] School of Electrical and Computer Engineering, {sli,sudha}@gatech.edu,
[2] College of Computing, naila@cc.gatech.edu,
Georgia Institute of Technology, USA

Abstract. As the role of highly-parallel accelerators becomes more important in high performance computing, so does the need to ensure their reliable operation. In applications where precision and correctness is a necessity, bit-level reliable operation is required. While there exist mechanisms for error detection and correction, the cost-effective implementation in massively parallel accelerators is still an active area of research. In this paper we present an alternative software based approach for improving the reliability of massively parallel bulk synchronous processors such as modern GPUs. Specifically, we propose a set of software reliability enhancements via transparent code patching of GPU applications. Reliability enhancements can be applied *selectively* at runtime, *customized* by the user, and *transparent* to the application. Runtime overhead ranges from 1-737% depending on the nature of the enhancement. We provide an analysis of benefits and limitations.

Keywords: GPU, correctness checks, instrumentation, reliability, CUDA, PTX

1 Introduction

Bulk synchronous parallel (BSP) processors such as general purpose graphics processing units (GPUs) are becoming a dominant computation resource in high performance computing. Potential failures inherent in GPUs are magnified with their increased usage. Haque et. al. [1] showed two third of consumer grade GPGPUs have detectable soft errors. Such transient errors can be mitigated by mechanisms such as error correction-code (ECC) that can detect such faults and recover. Software ECC has been in use in professional-grade GPUs since the Fermi architecture, but they also introduce non-trivial runtime overhead–up to 100% [2, 3].

Fault detection and recovery mechanisms can prevent loss of precision and maintain correctness. Fault detection prevents the output of silent errors by detecting faults as they occur. Recovery mechanisms prevents the propagation of errors by rolling back the program state to a checkpoint. Together they ensure reliable execution. Such mechanisms are already implemented in some GPU accelerators, e.g., redundancy in the memory system and bus operation. Hardware

ECC enabled memory prevents transient faults in DRAM, while cyclic redundancy checks in the GDDR5 interface prevents faults from occurring during transfers across the memory bus[4]. However, these mechanisms incur extra cost and not all devices employ them and they do not cover all failure models, such as transient faults in compute or control logic.

Other approaches [5–8] demonstrated the increased reliability of software-based fault-detection techniques in CPU architectures. However they are not flexible in practical situations and require manual code insertion. The growing dominance of GPUs in compute-heavy infrastructures demands such features to be more amenable to deployment.

Therefore we seek an approach that is customizable, extensible, and transparent to the application program. In this paper we present a set of *transparent* and *automatic* software reliability enhancements to GPU applications using the Lynx dynamic instrumentation framework [9]. Lynx was originally developed to insert instrumentation code into a GPU executable, operating on the low level virtual instruction set, just above the assembly language level. We use this capability to enhance the reliability of a GPU kernel. This instrumentation capability affords two features: transparency and automation. Transparency is the ability to implement software error-detection without the need for manual program modification. Since Lynx is a dynamic library, user programs can invoke instrumentation *transparently* and *selectively* each time the application executes *automatically*. These two capabilities allows flexibility in how it is used by the end-user. Additionally, enhancements can be customized by the end user to tailor to the specific needs of an individual application. We document three initial enhancements and their usages, as well as benefits and limitations. The three enhancements are: alignment check, array bounds check, and control flow check. We show the performance overhead of these enhancements, ranging from 1-737%.

The rest of this paper is organized into the following sections. In Section 2, we cover the background of GPU computing, Lynx, and its underlying infrastructure. In Section 3, we illustrate the mechanisms behind the reliability enhancements in general and present individual examples. In Section 4, we demonstrate the overhead of the enhancements in several applications from the CUDA SDK [10] . In Section 5, we note potential enhancements in the future and characterize their benefits to fault detection.

2 Background

This section offers an overview of the GPU computing model and the context within which GPU Lynx and Ocelot [11] operate.

2.1 GPU Computing

A CUDA application is composed of a series of multithreaded, data-parallel kernels. These kernels consist of a grid of parallel work-units called *Cooperative Thread Arrays (CTAs)*, which in turn consist of an array of threads that may

synchronize at CTA-wide barriers. Each CTA is data and control-flow independent of other CTAs. This property enables CTAs to be executed concurrently and in parallel. This means they can be executed in any order without affecting the correctness of the program. In NVIDIA processors, threads within a CTA are grouped together into logical units known as *warps* that are mapped to single instruction stream multiple data stream (SIMD) units called stream multiprocessors (SMs). While derived from NVIDIAs CUDA, this model closely matches the execution model of the industry standard OpenCL.

All threads within a warp execute in lockstep. The execution of warps, however, are interleaved based on a hardware scheduler such that one warp may run-ahead of another. Each thread has access to global memory within the device. There is also a faster, hardware-managed, shared memory accessible by threads within each SM. While there is a synchronization mechanism between warps of a CTA, there exists no such mechanism across CTAs except for kernel boundaries.

In the accelerator model of execution, the host allocates and initializes corresponding data on the device side, and launches the kernel on the device by making function calls to the CUDA runtime.

2.2 Lynx and GPU Ocelot

Lynx [9] is a dynamic instrumentation engine for data-parallel applications on GPU architectures. Specifically, Lynx allows the creation of customized, user-defined instrumentation routines that can be applied transparently at run-time for a variety of purposes, including performance debugging and correctness checking. Lynx can be built as a library, where it can be linked with any runtime, or as a Lynx runtime, where it provides a default implementation of the CUDA runtime to directly support the execution of CUDA applications on NVIDIA GPU devices. In both cases, Lynx relies on GPU Ocelot's [12] Parser and IR/-Analyses classes for extracting and generating NVIDIA's parallel thread execution (PTX) intermediate representation from the compiled CUDA fat binary. GPU Ocelot is a dynamic compilation framework for executing CUDA applications on multiple backends. While the Lynx runtime only supports execution on NVIDIA GPU backends, the Lynx library can be linked with GPU Ocelot to take advantage of Ocelot's support for additional backend targets, such as x86 [12], AMD GPU [13], or the built-in emulator.

2.3 Instrumenting with Lynx

Lynx [9] enables insertion of user-defined instrumentations into CUDA applications. The general procedure for executing CUDA applications includes extracting the compute kernel source in the NVIDIA's PTX instruction set from the compiled CUDA fat binary, generating an intermediate representation (IR) for this PTX kernel, applying the relevant PTX transformations based on the transformation passes defined (performed by the PTX-PTX Transformations Pass Manager), and finally executing the modified kernels on the backend target.

These transformations can be used to characterize application performance or implement runtime correctness checks and other reliability enchancements. Figure 1 shows the overview of how reliability enhancement transformation passes fit into the instrumentation workflow of Lynx. For more information on the transformation pass framework, please refer to [14].

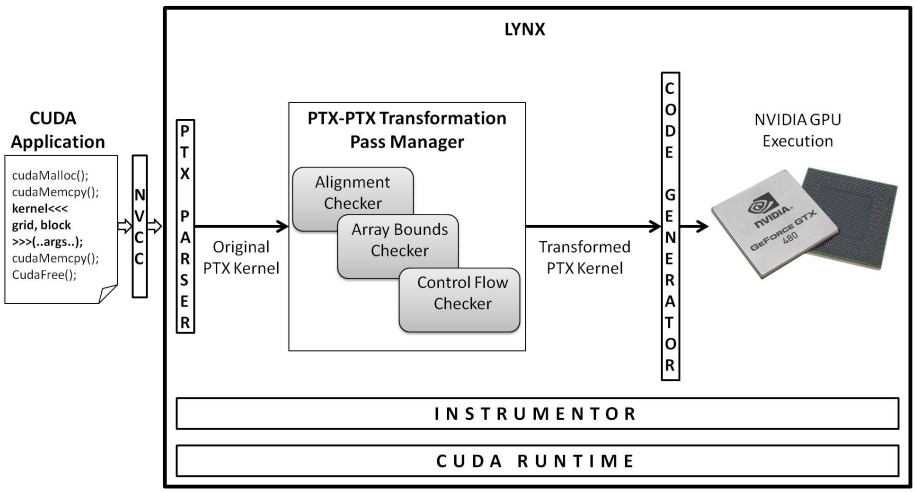

Fig. 1. Reliability enhancement passes within Lynx

3 Reliability and Correctness Enhancements

The contribution of this paper lies in the reliability and correctness checking enhancements implemented as instrumentations in Lynx.

3.1 Anatomy of an Instrumentation

Each instrumentation is broken into two classes: instrumentation and transformation. The *instrumentation class* performs data allocation on the device, creates an object of the transformation class, and transfers the analysis data on the device back to the host after kernel execution.

The transformation class takes the device kernel and insert, modify, or replace instructions based on user-defined criteria. The *transformation class* modifies the kernel by applying a procedural transformation pass over one or more of three levels: module, kernel, and basic block. The transformation pass can look for a specific class of instructions, such as memory instructions, and inject enhancement code for a variety of purposes, such as to count the number of dynamic executions or to check the validity of its operands.

GPU instrumentation enables a diverse set of software reliability enhancements spanning correctness checking and failure detection. Correctness checking functions include memory alignment detection, array bound access check, and potentially uninitialized shared memory usage detection. Possible fault detection instrumentations include transient fault checks (implemented as automatic redundant kernel execution), illegal control flow detection, and software based ECC. While not all of these reliability enhancements have been implemented in the form of instrumentations, this list shows the potential of GPU instrumentation.

3.2 Software Reliability Enhancements (SRE)

The general algorithm of each SRE is composed of three phases: detect a transient fault, flag the occurrence of such a fault, and resume execution. It is up to a higher level runtime monitor to determine the necessary course of action once an error has been detected, such as ignore the error, re-execute the kernel, or initiate an application specific error handling procedure. SRE's are not inherently immune to faults within the inserted code, but they can be resilient to some faults. Each SRE targets a different fault model. The alignment checker targets errors in memory address operands in load/store instructions. These errors can be the result of transient bit-flips or missed algorithmic or logic errors. The array bounds checker is similar to the alignment checker in that it targets errors occurring in memory address operands in load/store instructions, but it also checks the address against the boundaries of all valid global memory allocations. The control flow checker targets faults in the branch address operand of control-flow instructions and detects illegal control flows.

Alignment Checker This SRE checks each memory instruction and ensures correct alignment of its memory address operands. An address is aligned when it is a multiple of the target data size. Upon detecting an unaligned address, it will write diagnostic information to global memory where, upon kernel execution completion, it will be transferred to the host. To make room for diagnostic information, a data structure is allocated on the device before kernel launch.

The transform class iterates over each instruction in the kernel. At each memory instruction, it finds the address operand and calculates the final address using the offset and the base address. A bit-mask is generated at compile time based on the data type of the memory instruction. The final address is then masked to reveal the lower order bits. An error is detected if this result is non-zero and the error flag is written back. The instructions for write-back are predicated on the comparison between the result register and zero, as shown in Listing 1.

The PTX instructions inserted before each memory operation is shown in Listing 1. Pseudocode is added in the comments to illustrate the purpose of each code segment. If a bit were to flip in one of the registers during the alignment check, there is a non-zero chance that a silent error will occur, though this is very unlikely. When an error is detected, the diagnostic data will be copied to the host upon kernel execution completion.

Listing 1. Alignment Checker PTX instrumentation

```
//original instruction is: a = array[i]
//check if address is aligned
add.u64 %MemAddr, %array, %i
and.b64 %result, %rMemAddr, 3;
setp.ne.u64 %unAligned, %result, 0;

//write back to log in global memory
@%unAligned st.global.u64 [%log], 1;    //fault occured
@%unAligned st.global.u64 [%log + 8], %tid;
@%unAligned st.global.u64 [%log + 16], %MemAddr;
@%unAligned st.global.u64 [%log + 32], sourceLineNumber;

//original load instruction
ld.global.f32 %r9, [%array + %i];
```

Array Bounds Checker This SRE compares each global memory reference to a list of known global allocations to determine its legality. A sparse allocation table is generated by the instrumentation object at compile time and inserted into device memory via CUDA API calls. This table contains the first and last address of each global memory allocation made by the original program. In the transformation pass, memory instructions referencing the global address space is checked against this table in a *for* loop. If the address of the memory reference does not reside in the table, it is deemed illegal and relevant data is written to global memory.

Listing 2 shows the code inserted for every global memory instruction. Future improvements can be made to compartmentalize this into a function call. This instrumentation requires creation of new basic block and modification to the control flow graph. Depending on the frequency of global memory accesses, this enhancement pass can incur high overhead due to the execution of a *for* loop on each memory access. The backward edge and conditional exit out of this for loop has a high impact on bulk-synchronous compute architectures due to branch redirection and divergence behavior, which stalls the in-order pipeline of the Stream Multiprocessors (SM) and reduces the number of available warps in the hardware scheduler.

This method of illegal memory reference detection does not catch all incorrect addresses, however. Because the address checking mechanism does not keep track of each memory instruction and their designated allocation, it is possible for a corrupted operand to point to a valid memory allocation other than the intended and proceed undetected. A simple extension to this is to statically track all array references to their respective memory allocation and insert corresponding checks. While this approach would not cover indirect accesses via arbitrary pointers, it would reduce the instrumentation overhead depending on the number of global memory allocations in the original program. This enhancement is planned in future work.

Listing 2. Array Bounds Checker PTX instrumentation

```
//original code: x = *memAddr
//load allocation map pointer and find end of map
ld.global.u64 %mapPtr, [__global_allocation_map];
ld.global.u64 %numElem, [%mapPtr];
add.u64 %mapEnd, %mapPtr, %numElem;

Loop_Start:
//if end of map reached, no valid alloc found
setp.lt.u64 %pred, %mapEnd, %mapPtr;
@%pred bra Error;

Check_Base_Address:
//is memAddr less than base address?
//if so, compare against next allocation
ld.global.u64 %baseAddr, [%mapPtr + 8];
setp.lt.u64 %pred, %memAddr, %baseAddr;
@%pred add.u64 %mapPtr, %mapPtr, 16;
@%pred bra Loop_Start;

Check_End_Address:
//is memAddr is less than last address of allocation?
//if so, memAddr falls within this allocation
ld.global.u64 %endAddr, [%mapPtr + 16];
setp.lt.u64 %pred, %memAddr, %endAddr;
@%pred bra No_Error;

End_loop:
//increment mapPtr to next allocation
add.u64 %mapPtr, %mapPtr, 16;
bra Loop_start;

Error:
//report diagnostic information
cvt.u64.u32 %r43, %tid.x;
st.global.u64 [%r36], %r43;
st.global.u64 [%r36 + 8], %memAddr;

No_Error:
//original code
ld.global.f32 %x, [%memAddr];
```

Control Flow Checker This SRE is the same as that proposed in [5]. It enhances every basic block with a unique signature. With the exception of the entry block, the beginning of each basic block is instrumented with a signature check. This check hashes the signature of the previous block (source block) in a dynamic control flow with that of the current block (target block). A comparison checks if the result of this hash matches the precomputed value. In the event of a mismatch, an error flag will be written back and signaled to the host upon completion of the kernel. In the case of a control-flow fan-in, multiple basic blocks can legally branch to the same block. Without additional mechanisms, multiple blocks will share the same signature in order to match the target block signature. Instead, a Signature Difference Register (SDR) is used to distinguish the source basic blocks from one another. The SDR will be populated by the source basic block and only checked by the target block in the case of a fan-in.

Listing 3 shows the enhancement code inserted in each basic block. The entry block only requires an initial signature. In the beginning of subsequent

blocks, the signature register is hashed with a value to derive the signature of the current block. The resultant signature is compared against a precomputed value, and the error handling is predicated on this comparison. BasicBlock2 & 3 are source blocks to a control-flow fan-in, which requires the initialization of the SDR. In BasicBlock4, the SDR is first hashed with the signature register before the actual hash to determine the current signature. In this way BasicBlock2 & 3 possess unique signatures but can still resolve to the same signature for control-flow fan-ins. More information regarding this method can be found in [5]. If a transient bit flip were to occur in the checker code, a false positive may be reported as this will likely modify the signature and result in a mismatch.

Listing 3. Control Flow Checker PTX instrumentation

```
EntryBlock:
mov.u64 %signature, 55062;
add.u64 %a, %b, %c;  //body

BasicBlock2:
xor.b64 %signature, %signature, 46048;
setp.ne.u64 %pred, %signature, 25846;
@%pred st.u64 [%error], 1;
@%pred exit;
mov.u64 %difference, 73939;
bra BasicBlock4; //body

BasicBlock3:
xor.b64 %signature, %signature, 14893;
sept.ne.u64 %pred, %signature, 24283;
@%pred st.u64 [%error], 1;
@%pred exit;
mov.u64 %difference, 72446;
add.u64 %a, %b, %c;  //body

BasicBlock4: //control flow fan-in
xor.b64 %signature, %signature, %difference;
xor.b64 %signature, signature, 94732;
sept.ne.u64 %pred, %signature, 13865;
@%pred st.u64 [%error], 1;
ret;    //body
```

4 Evaluation

We evaluated the software reliability enhancements (SRE) over 12 CUDA SDK benchmarks [10] on an NVIDIA, Kepler architecture GPU, the GTX 660 Ti. The total runtime of each kernel, without memory transfer overhead, was measured with the command line cudaprof [15]. As shown in Figure 2, runtime overhead of these benchmarks ranged from below 1% to 737%, depending on the benchmark and the enhancement. Each SRE has unique characteristic in terms of overhead. The Control Flow Checker (CFC) showed minimal overhead, ranging from less than 1% to 14%. Since this SRE inserts only several instructions per basic block, its impact on performance is the smallest out of the sampled SRE. The Bounds Checker (BC) exhibits the highest overhead overall. This overhead has a strong correlation with the normalized dynamic branches executed in Figure 3.

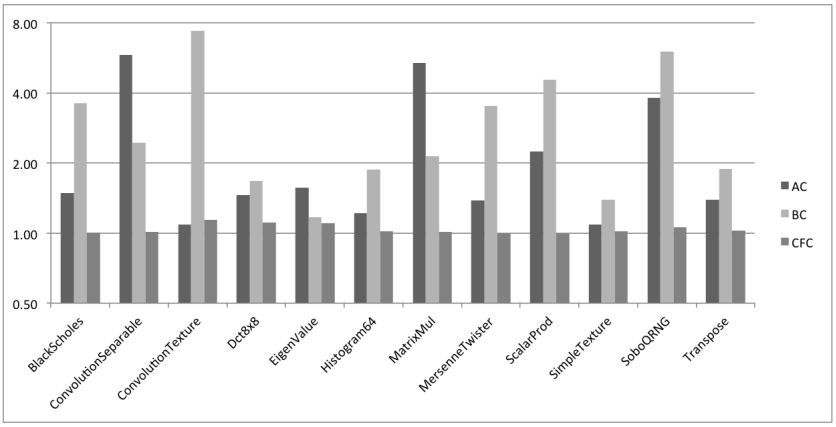

Fig. 2. Normalized runtime

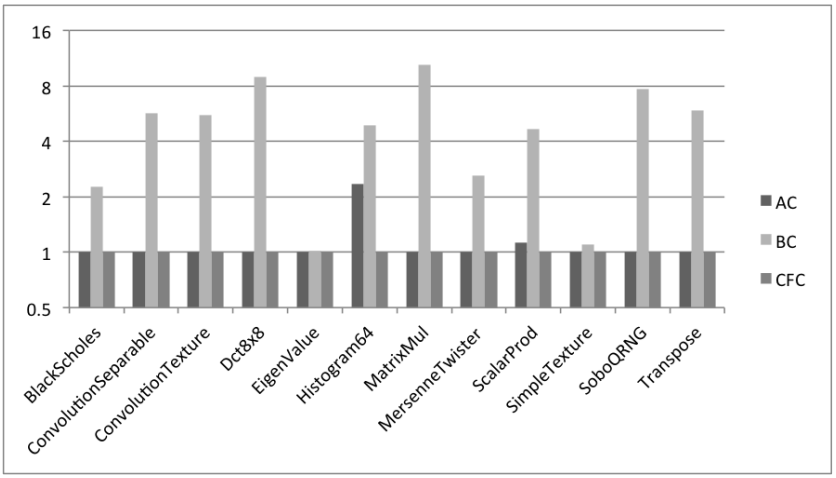

Fig. 3. Normalized dynamic branches

The Alignment Checker (AC) inserts less instructions per memory reference than BC but is more intrusive than the block level instrumentation of the CFC. While AC does not introduce comparably as many branches as CFC, it does induce similar amount of L2 cache misses as CFC in ConvolutionSeparable, Eigenvalue, and MatrixMul, as shown in Figure 4.

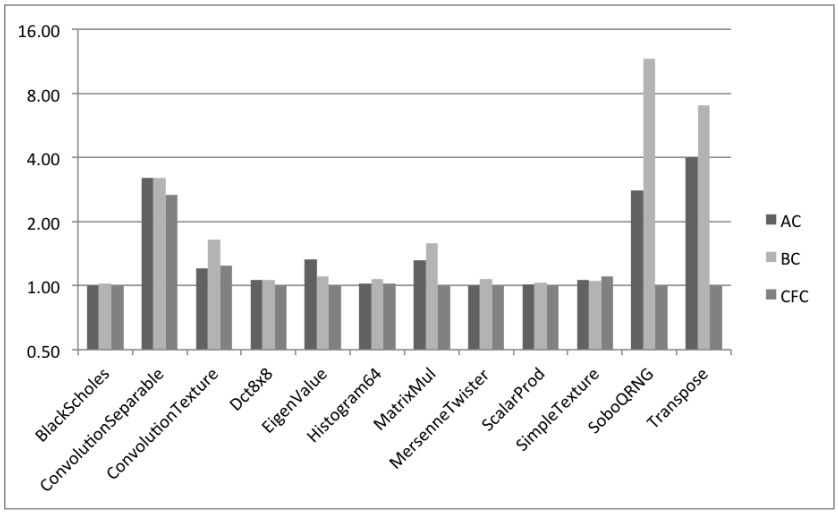

Fig. 4. Normalized L2 misses

5 Future Work

Future work includes a more fine grained examination of the performance characteristics of each SRE, randomized fault injection, and their efficacy at detecting and improving their own resilience to such faults. We plan to develop more resilience specific instrumentation, including redundant kernel execution, selective software-based ECC, and enhancements to control flow checking such as detecting legal but wrong branches by dynamically assign signatures based on condition variable.

6 Related Work

Enhancing reliability using software patching is not a novel concept. Yau, et. al. [8] proposed the notion of self-checking software, citing useful checks such as function call, control sequence, and data integrity. Annelid [7] performs bounds-checks by recording array pointer variables and the legitimate range of memory they can access. Annelid is a plugin to Valgrind [16], a dynamic instrumentation platform that translates x86 machine code to a virtual ISA, where instrumentation occurs, and the result is translated back into machine language. Oh, et. al. [5] propose control flow checking using software signatures. This technique is implemented as SRE in this paper. In branching-fault injection experiments, this method lowered the percentage of undetected incorrect output from 33.7% to 3.1%. SWIFT [6] enhances control flow checking by dynamically assigning block signatures based on the branch condition using redundant execution, but

limits to branches affecting stores. This technique target Itanium processor using a modified OpenIMPACT compiler. Erez et. al. [17] proposed a fault tolerance technique using redundant execution, checkpoints at control flows governing write backs, and control flow checking only at checkpoints. This technique focuses on Merrimac architecture. Sheaffer et. al. [18] proposed redundant hardware execution resources to provide resilience in the face of transient faults in computation logic. Dimitrov et. al. [19] proposed three methodologies of redundant execution to achieve software reliability in GPU applications with approximately 100% overhead. One was duplicate kernel execution and the other two utilized instruction-level and thread-level parallelism. They also argued ECC protected memory does not significantly reduce execution time in redundant computation scenarios, as input data are not always read-only and as such requires duplication. Maruyama et. al. [4] demonstrated a low overhead software-based ECC for GPU applications by offloading parity computation to the CPU. Their implementation is in the form of a library, which requires manual insertion of library calls into the target application, where as reliability enhancements using Lynx can be performed transparently at runtime. Chung et. al. [20] proposed the concept of Containment Domains (CD) for resilient computing in a scalable and efficient manner. CDs are hierarchical in nature and failures in nested CD are localized and do not propagate outward. Reliability enhancements proposed in this paper can be used to detect failures where a more efficient algorithmic failure detection does not exist.

7 Conclusion

This paper presents a set of reliability enhancements that has the flexibility to run selectively, be customized for a specific purpose, and applied transparently to the application without modifying the original source. Runtime overheads ranged from less than 1% to less than 8x runtime. This broad range is dependent on the behavior of the original application as well as the intrusiveness of the enhancement. Ultimately, this reliability mechanism can be extended based on the specific needs of the user for critical applications as well as for operating on consumer devices that lack the required built-in reliability mechanisms. Future iterations of these enhancements will aim to increase the robustness and extend to cover other fault models such as memory faults.

8 Acknowledgement

This research was supported in part by the National Science Foundation under grands CCF 0905459 and award OCI-0910735, by equipment grants from NVIDIA Corporation.

References

1. Imran S Haque and Vijay S Pande. Hard Data on Soft Errors : A Large-Scale Assessment of Real-World Error Rates in GPGPU. pages 1–10.

2. M. Bernaschi, G. Parisi, and L. Parisi. Benchmarking GPU and CPU codes for Heisenberg spin glass over-relaxation. *Computer Physics Communications*, 182(6):1265–1271, June 2011.
3. Timo Stich. Fermi Hardware & Performance Tips, 2011.
4. N Maruyama. A high-performance fault-tolerant software framework for memory on commodity gpus. *Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010.
5. Nahmsuk Oh, Philip P Shirvani, Edward J Mccluskey, and Life Fellow. Control-Flow Checking by Software Signatures. 51(2):111–122, 2002.
6. G.a. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: Software Implemented Fault Tolerance. *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.
7. Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. *SPACE*, 2004.
8. SS Yau and RC Cheung. Design of self-checking software. *ACM SIGPLAN Notices*, 10(6):450–455, 1975.
9. Naila Farooqui, Andrew Kerr, Greg Eisenhauer, Karsten Schwan, and Sudhakar Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 58–67, April 2012.
10. C. Nvidia. Compute unified device architecture programming guide. 2007.
11. G Diamos, A Kerr, and M Kesavan. Translating GPU binaries to tiered SIMD architectures with Ocelot. *Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-01, January*, 2009.
12. GF Diamos and AR Kerr. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, 2010.
13. R Dominguez, Dana Schaa, and David Kaeli. Caracal: Dynamic translation of runtime environments for gpus. *Proceedings of the 4th Workshop on General on General-Purpose Computation on Graphics Processing Units*, (March), 2011.
14. Naila Farooqui, Andrew Kerr, Gregory Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-4*, page 1, 2011.
15. NVIDIA. Nvidia cuda tools sdk cupti. February 2011.
16. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, pages 89–100, 2007.
17. M. Erez, N. Jayasena, T.J. Knight, and W.J. Dally. Fault Tolerance Techniques for the Merrimac Streaming Supercomputer. *ACM/IEEE SC 2005 Conference (SC'05)*, (c):29–29, 2005.
18. JW Sheaffer, DP Luebke, and Kevin Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 55–64, 2007.
19. Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for GPGPU reliability. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, pages 94–104, 2009.
20. Jinsuk Chung, Ikhwan Lee, Michael Sullivan, and JH Ryoo. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. *SC*, 2012.