Software-Based ECC for GPUs

Naoya Maruyama and Akira Nukada
Tokyo Institute of Technology
JST CREST

Satoshi Matsuoka
Tokyo Institute of Technology
National Institute of Informatics
JST CREST

***Abstract*—Commodity off-the-shelf GPUs lack error checking mechanisms for graphics memory, whereas conventional HPC platforms have used hardware-based ECC for DRAMs. To alleviate this reliability concern, we propose a software-based ECC for GPGPU applications. We add small program codes to normal CUDA programs that compute ECCs for data residing in graphics memory so that transient bit-flips can be detected or masked. Preliminary performance studies with 3-D FFT and the N-body problem show that error checking using ECC can take 200% and 7% of overhead, respectively. We discuss that performance overheads are derived from the cost of ECC computation on GPUs.**

I. Introduction

General-purpose computing on graphics processing units (GPGPU) has rapidly been recognized as a promising HPC technology because of GPUs' much higher peak floating-point processing power than conventional CPUs. However, since GPUs have originally been developed for graphics applications, such as 3-D games, where transient errors can be tolerable in many cases, its reliability has not been given as much consideration as in HPC communities. One notable example is the lack of error checking in graphics memory systems. DRAMs in conventional HPC platforms are usually equipped with hardware-based error checking mechanisms, such as Error Correcting Codes (ECC) that can detect double-bit errors and correct single-bit errors. Since radiation-induced soft errors, while the likelihood of occurrences is very small, can manifest themselves as large deviation in final execution results, such error protection is essential in HPC applications, especially long-running ones using a large number of GPUs. However, as far as we know, none of the currently available GPUs manufactured by both NVIDIA and ATI provides any variants of error checking for graphics memory.

To safely exploit GPUs for accelerating HPC application performance, we propose a software-based approach to error checking for GPU memory. Although extreme high-end cards for the professional market might adopt hardware ECC in the future, it is very likely that mainstream GPGPUs will not adopt ECC in any foreseeable future, just as commodity PCs do not as well despite their exponential growth in their performance and memory capacity. This extended abstract presents an overview of our software-based approach with preliminary performance evaluation.

II. Our Approach

Memory errors such as bit flips can be detected and/or corrected with *error-checking codes*, such as parity bits and Hamming codes [1]. The most common case is Error-Correcting Codes (ECC), which use both parity bits and Hamming codes. ECC allows single-bit errors to be corrected and double-bit errors to be detected (SECDED). For example, typical ECC DRAMs reserve an 8-bit code region 64 bits to store the code, where seven bits are consumed by Hamming code and one bit by parity. When a block of 64 bits being written to DRAMs, the memory system writes its 8-bit ECC to the code region as well; When the data being read, its code is again generated and checked against the previously generated code. When the two codes do not agree, there must be bit flips in the data, which may or may not be corrected depending on the number of flipped bits.

As an initial feasibility study, we realize the same memory fault tolerance in GPGPUs by implementing this error-checking scheme by software, thus allowing commodity hardware with no built-in error checking mechanisms to be reliably used for scientific computing. More specifically, for each memory write, we embed a small additional program code that calculates and saves error-checking code of the value written. The additional program code for read accesses uses the code to check the correctness of reads from the memory.

Our current prototype implements this error checking scheme as a library for NVIDIA CUDA [2]. CUDA exposes several memory components to applications, such as *shared memory*, *global memory*, and *texture memory*. This preliminary study focuses on protecting applications from errors in the global memory, which typically has the largest capacity backed by off-chip graphics memory (hundreds of mega bytes to a few giga bytes in the current generation of GPUs). Errors occurring in other areas remain a subject of further work.

Since the current approach is library-based, the process of adding error checking is not completely automatic, but rather requires manual modifications of user programs. Specifically, a user program has to be modified so that it performs the following three additional

tasks. First, it needs to allocate separate memory for storing ECCs of global memory. This is a relatively simple task since global memory in CUDA has to be explicitly allocated by calling `cudaMalloc`; the user can simply add an additional call to `cudaMalloc` for each global memory allocation site. Second, for each access to global memory, it needs to compute the ECC of the value accessed and to detect or correct errors if possible. Our library provides functions for computing ECCs for several data types, including integers and floating points of various sizes. The user program needs to be modified so that every memory read is followed by a call to its corresponding error-checking function. Similarly, every write access is modified to be followed by a library call that generates and saves the code of the written data. Third, since global memory can be copied to and from CPU memory, it needs to generate codes of the data on the CPU memory as well. For example, a typical scenario of using global memory is that the user program first allocates an area and initializes it with the data residing in CPU memory. In that case, the CPU first needs to generate the codes before transferring it to the global memory. Our library provides several functions for this purpose so that the user program can simply call one of them to implement code generation on the CPU.

The preliminary prototype has several limitations and assumptions. First, it does not support race-free concurrent writes, but simply assumes that all the writes touches separate regions. The CUDA API provides atomic operations on global memory, allowing mutual exclusion among concurrent multiple accesses to the same region. However, its access granularity is limited to only 32-bit data, so writes of original 32-bit data and its code cannot be performed atomically, but must be done by two separate transactions. Second, the current prototype relies on the user so that she correctly manages the mapping of protected data and its ECC. This can be a trivial task when an application is simple and small scale, such as matrix multiplication; however, in general, indirect memory accesses through pointers could make the management of mapping rather difficult and tedious. We are now considering automating the mapping by compiler-based analyses.

III. Preliminary Evaluation

As preliminary performance studies, we first evaluate the basic throughputs of memory reads with the software ECC to understand the overhead derived from the software ECC. Next, we extend two application kernels, an N-body problem and FFT [3], with ECC to show application-level performance overheads. We select the two kernels as representatives of compute-intensive and bandwidth-intensive applications. All of the measurements are done on three Nvidia GPUs:

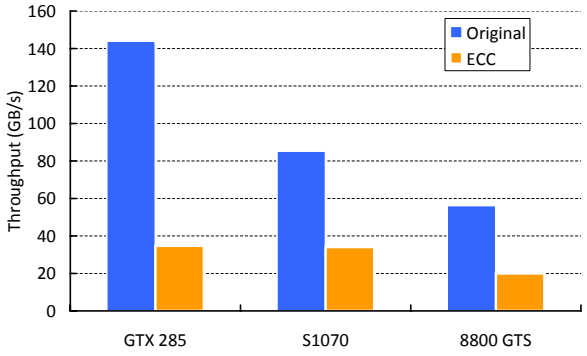

Fig. 1. Throughputs of memory reads w/ and w/o ECC.

GeForce GTX 285, Tesla S1070, and GeForce 8800 GTS 512 (abbreviated as GTX 285, S1070, and 8800 GTS). The former two GPUs are both based on the same latest Nvidia GPU architecture (G200), but have different memory capacities and speeds. The 8800 GTS is derived from the older generation (G92). We use CUDA version 2.1 for the GTX 285 and GTS 8800 GPUs, and 2.0 for the Telsa GPU.

Figure 1 compares the throughputs of memory reads with and without ECC. The throughputs decreased to 24%, 40%, and 35% with GTX 285, S1070, and 8800 GTS, respectively. We speculate that this throughput degradation can be explained by the ECC computation cost. Our prototype takes 63 integer 32-bit logical operations to generate an ECC for a 64-bit datum, which means that one byte of data approximately requires eight integer operation. As shown in the blue bars in the graph, the GTX 285 GPU achieves more than 140 GB/s. To keep up the memory throughput, the GPU need to process 1120 giga operations per second (GOPS), which is far beyond of its theoretical limit of 355 GOPS. In other words, the GPU can at most afford 44 GB/s throughputs. In addition, read accesses incur other instructions, including the comparison of codes, so the actual ECC throughputs is lower than the limit. Thus, we believe that the performance of software ECC is computation bottleneck. This analysis is consistent with the fact that the ECC throughput of S1070 is approximately the same as that of GTX 285: While the latter achieves much higher throughput without ECC, they have very similar processing power (355 GOPS and 345 GOPS). The 8800 GTS GPU, whose processing speed is 208 GOPS, shows the similar behavior.

As an example of computation-intensive kernels, Figure 2 compares the performance of the N-body problem on the GTX 285 GPU with varying numbers of bodies. Here, we use the sample N-body code that is shipped with the CUDA SDK, which includes both a CPU version and a GPU version. As a reference, we also measure the CPU performance using a quad-core AMD Phenom

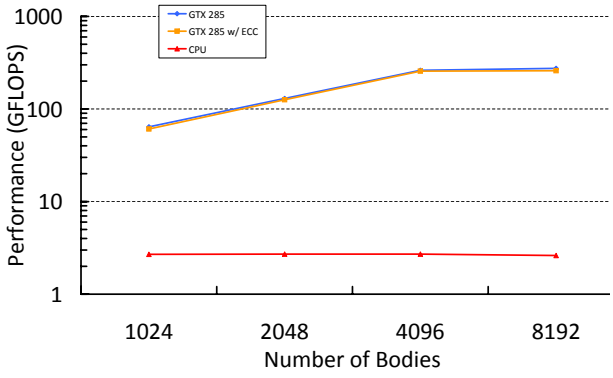

Fig. 2. Performance comparison of the N-body problem.

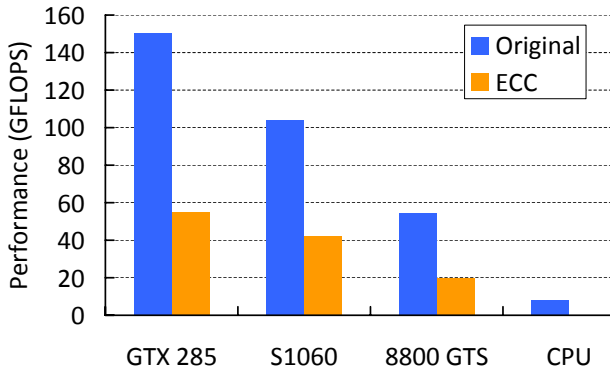

Fig. 3. Performance comparison of 3-D FFT of complex 256^3 points.

9850 processor (2.5 GHz) equipped with 4 GB of memory. The graph shows that the software ECC did not incur much overhead: approximately 7% compared to the original performance. This relatively small overhead is because the N-body problem is mostly computation dominant, so even if the memory throughputs degrade as in the above results, the overall performance should not be affected much.

As an example of bandwidth-intensive kernels, Figure 3 shows the performance overhead of software ECC in 3-D FFT of 256^3 complex numbers. The evaluation uses a bandwidth-dominant kernel, which is up to 3-times faster than the standard implementation in CUFFT [3]. We also show performance of FFTW executed on the CPU as in the N-body study. Each of the ECC versions decreases its performance to approximately 36% (GTX 285 and 8800 GTS) or 40% (S1070). These results reflect the throughput reduction results since the performance of the underlying kernel is bandwidth intensive.

In summary, the performance studies suggest that the software ECC would be feasible with little performance overhead when application kernels are compute intensive, whereas bandwidth-intensive kernels can suffer significant overheads by up to four times. Therefore, while the current approach could be effective for compute intensive kernels, lighter-weight techniques would be necessary for the others.

IV. Related Work

Sheaffer et al. proposed a hardware extension for fault tolerance in GPGPU [4]. The extension implements this error checking for GPUs by executing pixel shading multiple times. Note that while their technique requires hardware changes, ours can be used with current off-the-shelf GPUs. Also, the two techniques can be considered complementary because Sheaffer et al. assumes that graphics memory is protected by ECC-based error checking, for which our proposed technique can be used.

While we only consider errors in external DRAM in GPUs, Dimitrov et al. presented cost evaluations of complete redundant executions in GPUs, which indicate two times performance overhead [5]. Specific protection methods can be chosen depending on application sensitivities to errors and error rates of GPU components.

V. Summary

We presented a brief overview and preliminary performance results of our software-based ECC for memory-fault tolerance in GPGPU. Our current approach realizes ECC for CUDA global memory by requiring small modifications to user programs so that each memory access site is followed by a call to an appropriate error-checking function in our library. Preliminary performance evaluation indicated that software-based approach can be performed with acceptable performance overheads for compute-intensive applications such as the N-body problem; however, bandwidth-intensive applications require more efficient coding schemes. Our future work includes more performance studies, investigation of lighter-weight approaches, and fault-injection-based recovery studies.

Acknowledgment

This work is supported in part by Microsoft Technical Computing Initiative entitled “HPC-GPGPU”, and in part by a JST CREST research program entitled “Ultra-Low-Power HPC”.

References

[1] E. Fujiwara, *Code Design for Dependable Systems: Theory and Practical Applications*. Wiley Interscience, 2006.

[2] “NVIDIA CUDA Programming Guide,” http://developer.nvidia.com/object/cuda.html.

[3] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, “Bandwidth intensive 3-D FFT kernel for GPUs using CUDA,” in *SC’08*, Nov 2008.

[4] J. W. Sheaffer, D. P. Luebke, and K. Skadron, “A hardware redundancy and recovery mechanism for reliable gpgpu,” in *Graphics Hardware 2007*, Aug 2007, pp. 55–64.

[5] M. Dimitrov, M. Mantor, and H. Zhou, “Understanding software approaches for gpgpu reliability,” in *Proceedings of GPGPU-2*, 2009, pp. 94–104.