On Testing GPU Memory for Hard and Soft Errors

Guochun Shi
Jeremy Enos
Michael Showerman
Volodymyr Kindratenko

Innovative Systems Laboratory

National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
Presentation Outline

• Motivation
• Test Methodology
  • Hard errors
  • Soft errors
• Findings to date
  • Hard errors
  • Soft errors
  • Some additional observations
• Related work
Motivation

- GPU (device) memory is not ECC protected
  - Not a problem with graphics computing
  - Could be a major issue for scientific computing
    - Some results may not be correct

- Questions to answer
  - Are there memory cells in the GPU memory chips that are permanently damaged (hard errors)?
  - If a cosmic ray causes a bit to flip, how often does it happen (soft errors)?
Methodology for Detecting Hard Errors

- Test methodology and test patterns are adopted from `memtest86/memtest86+`
  - 10 memory tests for hard errors detection

- Different stages are implemented as separate GPU kernels
  - To ensure the data is written to memory

- If an error is detected, all details are reported to the user
  - GPU ID, memory address, original data, erroneous data, etc.

Write patterns to memory

Read patterns, verify, write complements to memory

Read and verify
## Test Patterns for Detecting Hard Errors

<table>
<thead>
<tr>
<th>index</th>
<th>Test pattern</th>
<th>Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Walking 1 bit</td>
<td>This test changes one bit a time in memory address to see it goes to a different memory location. It is designed to test the address wires.</td>
<td>address bus test</td>
</tr>
<tr>
<td>1</td>
<td>Own physical address test</td>
<td>Each Memory location is filled with its own address. The next kernel checks if the value in each memory location still agrees with the address.</td>
<td>address bus test</td>
</tr>
<tr>
<td>2</td>
<td>Moving inversions (ones and zeros)</td>
<td>This test uses the moving inversions algorithm with patterns of all ones and zeros.</td>
<td>stuck-at faults errors</td>
</tr>
<tr>
<td>3</td>
<td>Moving inversions (8-bit pattern)</td>
<td>This test is the same as test 2, but it uses an 8 bit wide pattern of &quot;walking&quot; ones and zeros. This test will better detect subtle errors in &quot;wide&quot; memory chips.</td>
<td>stuck-at faults errors</td>
</tr>
<tr>
<td>4</td>
<td>Moving inversions (random pattern)</td>
<td>This test uses the same algorithm as test 2, but the data pattern is a random number and it's complement. This test is particularly effective in finding difficult to detect data sensitive errors. The random number sequence is different with each pass so multiple passes increase effectiveness.</td>
<td>data sensitive errors</td>
</tr>
</tbody>
</table>

See [https://sourceforge.net/projects/cudagpumemtest](https://sourceforge.net/projects/cudagpumemtest) for more details
## Test Patterns for Detecting Hard Errors

<table>
<thead>
<tr>
<th>index</th>
<th>Test pattern</th>
<th>Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Block move, 64 moves</td>
<td>This test stresses memory by moving block memories. Memory is initialized with shifting patterns that are inverted every 8 bytes. Then blocks of memory are moved around. After the moves are completed, the data patterns are checked. Because the data is checked only after the memory moves are completed, it is not possible to know where the error occurred.</td>
<td>data sensitive errors</td>
</tr>
<tr>
<td>6</td>
<td>Moving inversions, 32 bit pat</td>
<td>This is a variation of the moving inversions algorithm that shifts the data pattern left one bit for each successive address. The starting bit position is shifted left for each pass. To use all possible data patterns, 32 passes are required. This test is effective at detecting data sensitive errors, but the execution time is long.</td>
<td>data sensitive errors</td>
</tr>
<tr>
<td>7</td>
<td>Random number sequence</td>
<td>This test writes a series of random numbers into memory. A block (1 MB) of memory is initialized with random patterns. These patterns and their complements are used in moving inversions test with rest of memory.</td>
<td>data sensitive errors</td>
</tr>
<tr>
<td>8</td>
<td>Modulo 20, random pattern</td>
<td>A random pattern is generated. This pattern is used to set every 20th memory location in memory. The rest of the memory location is set to the compliment of the pattern. Repeat this for 20 times and each time the memory location to set the pattern is shifted right.</td>
<td>data sensitive errors</td>
</tr>
<tr>
<td>9</td>
<td>Bit fade test, 90 min, 2 patterns</td>
<td>The bit fade test initializes all of memory with a pattern and then sleeps for 90 minutes. Then memory is examined to see if any memory bits have changed. All ones and all zero patterns are used. This test takes 3 hours to complete. The Bit Fade test is disabled by default.</td>
<td>data retention error</td>
</tr>
</tbody>
</table>

See https://sourceforge.net/projects/cudagpumemtest for more details
Code Snippet for Moving Inversion Test

- Host function example

```c
void move_inv_test(char* ptr, unsigned int tot_num_blocks,
                    unsigned int p1, unsigned p2)
{
    unsigned int i;
    char* end_ptr = ptr + tot_num_blocks* BLOCKSIZE;
    for (i = 0; i < tot_num_blocks; i+= GRIDSIZE) {
        dim3 grid; grid.x = GRIDSIZE;
        _move_inv_write<<<grid, 1>>>(ptr + i*BLOCKSIZE, end_ptr, p1);
    }

    for (i = 0; i < tot_num_blocks; i+= GRIDSIZE) {
        dim3 grid; grid.x = GRIDSIZE;
        _move_inv_readwrite<<<grid, 1>>>(ptr + i*BLOCKSIZE,
                                         end_ptr, p1, p2, err_count, err_addr, err_expect, err_current);
        error_checking("move_inv_readwrite", i);
    }

    for (i = 0; i < tot_num_blocks; i+= GRIDSIZE) {
        dim3 grid; grid.x = GRIDSIZE;
        _move_inv_read<<<grid, 1>>>(ptr + i*BLOCKSIZE, end_ptr, p2,
                                    err_count, err_addr, err_expect, err_current);
        error_checking("move_inv_read", i);
    }
}
```

- Kernel function example

```c
__global__ void _move_inv_readwrite(char* _ptr, char* end_ptr,
                                     unsigned int p1, unsigned int p2, unsigned int* err,
                                     unsigned long* err_addr, unsigned long* err_expect, unsigned long* err_current)
{
    unsigned int i;
    unsigned int* ptr = (unsigned int*) (_ptr + blockIdx.x*BLOCKSIZE);
    if (ptr >= (unsigned int*) end_ptr) return;
    for (i = 0; i < BLOCKSIZE/sizeof(unsigned int); i++) {
        if (ptr[i] != p1) { RECORD_ERR(err, i, p1, ptr[i]); }
        ptr[i] = p2;
    }
    return;
}
```
Methodology for Detecting Soft Errors

• It has been reported that read-write intensive memory access test can detect soft errors more frequently (*)
• We designed a memory access intensive test to stress the memory system
  • Read/write at the maximum memory bandwidth

<table>
<thead>
<tr>
<th>index</th>
<th>Test pattern</th>
<th>Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Memory stress test</td>
<td>Device memory is set to a random pattern using a high-bandwidth write kernel. A read-write kernel is launched immediately to check if there are any errors and to replace the values stored in memory with their compliments. The process is repeated 1,000 times for one pattern.</td>
<td>Soft errors</td>
</tr>
</tbody>
</table>

(*) Soft Errors in Electronic Memory – A White Paper, Tezzaron Semiconductor, Jan 2004
Cluster Environment Setup for Memory Test

- **AC** cluster (32 nodes, 128 Tesla C1060 GPUs)
  - GPU memory test runs on idle GPUs
    - The batch system and scheduler are configured to allow preemptible jobs
    - By default, any job submitted is a preemptor
    - GPU memory test jobs are submitted with the preemptee flag set

- **Lincoln** cluster (192 nodes, 384 Tesla C1060 GPUs)
  - GPU memory test runs as a regular job
Findings: Hard Errors

- 4 out of 128 GPUs on AC cluster tested with hard errors
  - Node 3, GPU 3: multiple tests found errors at different addresses
  - Node 5, GPU 1: multiple tests found errors at different addresses
  - Node 25, GPU 1: multiple tests found errors at different addresses
  - Node 28, GPU 1: multiple tests found errors at the same address (0xe7fbc990) with the same bit (bit is stuck at ‘1’)
    - expected value=0xe7fbc990, current value=0xe7fbc9d0, diff=0x40

- 5 out of 378 GPUs on Lincoln cluster tested with hard errors
Findings: Hard Errors (Cont.)

- Our GPU clusters were built using early Tesla products
- NVIDIA confirmed that some Tesla units that were in our clusters were not properly tested with the manufacturing tests and thus could have had bad memory
  - Our tests found memory failures on systems that should not have been shipped to customers if the manufacturing tests were run properly
- Thus, our findings with regards to hard errors are not representative of what one might find when purchasing new hardware from NVIDIA
Findings: Soft Errors

- The rate at which soft errors occur is measured in the number of failures in time (FIT)
  - 1 FIT equals 1 soft error per billion hours of device operation
  - This rate is also sometimes reported as FIT per Mbit
- Modern memory devices are believed to have a 1,000 to 5,000 FIT per Mbit error rate, or about one bit error per month per gigabyte of memory(*)
  - We expected to see error rates at the order of 4 bit errors per GPU card per month of continuous operation

(*) Soft Errors in Electronic Memory – A White Paper, Tezzaron Semiconductor, Jan 2004
Findings: Soft Errors (Cont.)

- Two test runs on AC cluster
  - 7,222 hours of testing per ~4 GB of memory, idle GPU tests
  - 10,080 hours of testing per ~4 GB of memory, dedicated GPU tests
  - We run the tests for over 24 months combined, and expected to detect on the order of 96 errors

- Soft errors found: 0

- Possible reasons
  - Our error detection methodology is flawed
  - Soft errors are less frequent than previously stated and we have not tested for them long enough
  - The GPU memory chips are very robust with regards to soft errors
Some Additional Observations

• The memory stress test could leave the GPU in a “broken” state (CUDA 2.2/2.3 only)
  • In the GPU program that runs after this test, CUDA API calls start to fail, e.g.,
    • cudaMalloc() hangs
    • order of kernel execution in the stream seems to be broken
  • Results from the GPU in this state cannot be trusted
  • Reproducible by others, reported to NVIDIA in multiple ways

• Temporary fix: reload NVIDIA driver module
  • On AC cluster, we test GPU state after each job and reload the NVIDIA driver module if the GPU is detected to be in this “broken” state
Related Work

• memtestG80 from Stanford University
  • Aimed for GPGPUs, based on CUDA
  • Derived from memtest86/86+
  • No stress test, but there are tests for shared memory
  • Two versions: closed source and open source

• Using Stanford’s test, we were able to verify 3 out of 4 hard errors detected on the AC cluster
Conclusions and Future Work

- The developed GPU memory test suite is effective in detecting hard errors
- But we have yet to catch a soft error
  - Need to develop a better test methodology
- Future work includes adding tests for shared and constant memory and making the software to support a variety of GPU cards
- Software is available at SourceForge
  - [http://sourceforge.net/projects/cudagpumemtest](http://sourceforge.net/projects/cudagpumemtest)