SASSIFI: EVALUATING RESILIENCE OF GPU APPLICATIONS

Siva Hari, Timothy Tsai, Mark Stephenson,
Joel Emer, Steve Keckler
NVIDIA Research
Motivation

• Transient errors are a rising concern
• Need to evaluate resilience of applications
  - Silent Data Corruption (SDC), Detected Unrecoverable Error (DUE) rates
  - Identify vulnerable program sections
    - Key for developing low-cost mitigation schemes
• Application-level resilience evaluation is challenging
  - Traditional low-level error injection experiments are slow
  - Low visibility into application behavior
• Need quicker GPU application resilience evaluation scheme
Our Approach: Architecture-level Error Injections

- Inject error at architecture level
  - Fast
  - Visibility into application

- Leverages SASSI
  - A low-level assembly-language instrumentation tool

Advantages:
- Analyze and study SDCs in detail
  - Magnitude of SDCs and which errors produce SDCs
- Ability to correlate program properties with program vulnerability
  - Key to develop low cost error mitigation schemes
- Ability to quantify application level error derating factors
Overview of SASSI [ISCA 2015]

SASSI is a compiler-based instrumentation framework that allows us to inject code at specific points in a program.

Example: Identify all SASS memory ops and inject code needed to pass op’s address to a user-defined function

```
.L_8:
    ISCADD R7, R5, R3, 0x2;
    STS [R7], R2;
    BAR.SYNC 0x0;
    MOV R0, c[0x0][0x28];
    SHF.R R0, R0, 0x1, RZ;
    ISETP.EQ.AND P0, PT, ...
    @P0 BRA `(.L_12);
```

```
.L_8:
    ISCADD R7, R5, R3, 0x2;
    IADD R1, R1, -0x4;
    STL [R1], R4;
    IADD R4, R7, 0x0;
    JCAL `(_users_function);
    LDL R4, [R1];
    IADD R1, R1, 0x4;
    STS [R7], R2;
    BAR.SYNC 0x0;
```

1. Create extra stack space
2. Save live registers
3. Pass parameters of interest to user-defined function
4. Call user-defined function
5. Restore live registers
6. Restore stack
7. Execute instrumented instruction

User writes a handler function, _users_function, in CUDA

“Flexible Software Profiling of GPU Architectures,”
Mark Stephenson, Siva Hari, Yunsup Lee, Eiman Ebrahimi, Daniel Johnson, Mike O’Connor, Dave Nellans, and Steve Keckler, ISCA 2015
SASSIFI: SASSI based Fault Injector

- Leveraged SASSI for error injections
- Instrumented kernels for profiling and error injections
SASSIFI Methodology

CPU Code -> GPU Kernels

Output
SASSIFI Methodology

- Profile: Identify possible injection sites
- Instrumented kernels execute on the GPU
SASSIFI Methodology

- **Profile:** Identify possible injection sites
  - Instrumented kernels execute on the GPU
- **Statistically select injection sites**
  - Error model: Single-bit flip in one of the destination values of executing instructions
  - <kernel name, kernel invocation id, thread id, instruction id, destination reg id, bit id>
SASSIFI Methodology

- **Profile:** Identify possible injection sites
  - Instrumented kernels execute on the GPU

- **Statistically select injection sites**
  - Error model: Single-bit flip in one of the destination values of executing instructions
  - `<kernel name, kernel invocation id, thread id, instruction id, destination reg id, bit id>`

- **Injection runs:** inject one error at a time
  - Uses instrument-after, mem info, reg info
  - Start application, inject error at the selected site
  - Continue execution until a crash or the output

CPU Code  GPU Kernels

Output
Error Model

Our model: Single-bit flips in destination values of executing instructions
- Injection instruction is randomly selected among all dynamic instructions
- In a selected instruction, inject errors in:
  - One of the destination registers
  - Store value
  - Control and predicate registers (only the bits that are being written by the instruction)

Future work:
- Unbias injection site selection
  - All instructions are given equal weight in current model
- Correlate micro-architectural bit flips with architectural bit flips to tune rates of architectural injections
  - How to accurately model low-level errors at architecture level - single/multi-bits in single/different values?
## Outcome Categories

<table>
<thead>
<tr>
<th>Outcome Categories</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Potential DUEs</strong></td>
<td></td>
</tr>
<tr>
<td>Crashes</td>
<td>App exits with non-zero status</td>
</tr>
<tr>
<td>Hangs</td>
<td>App does not terminate in allocated time (10x of normal runtime)</td>
</tr>
</tbody>
</table>
| **Failure Symptoms** | Error messages in `stdout`, `stderr`  
Output files are missing |
| **Potential SDCs**  |            |
| Stdout only different | Only `stdout` is different |
| Output file difference | Output file generated by the program is different |
| **Masked**         | All output files (`stdout`, `output`, `stderr`) match |
Preliminary Results: Overview

1,000 error injections per application
Preliminary Results: Which kernel is more vulnerable?
Preliminary Results: How SDC rate changes with kernel invocations?

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

0 2 4 6 8 10 12 14

Kernel2 Kernel Fan2 kernel dynproc_kernel

Output file different
stdout only different

Potential SDCs

Failure symptoms
Hangs
Crashes
Masked

Potential DUEs

bfs gaussian heartwall pathfinder

0-127 256-383 512-639 768-895
Preliminary Results: How masking rate changes with injection location?

- General Purpose Registers
- Store values
- Condition Codes
- Predicate Registers

Potential SDCs:
- Output file different
- Stdout only different

Potential DUEs:
- Failure symptoms
- Hangs
- Crashes

Masked
Preliminary Results: Slowdowns

- Average slowdowns for Rodinia benchmarks is about 4.8x
  - Maximum of 138x and minimum of 1.1x
  - Kernel level slowdowns are higher (max 780x, min 54x)
- Significantly faster than architecture-level simulators
Summary and Future Work

- **SASSIFI**: Architecture-level error injection framework for GPUs
  - Fast
  - Visibility into application
- Ability to analyze and study SDCs in detail
  - Key to develop low cost error mitigation schemes
- Primary results show sensitivity of different kernels, invocations
- **Future work:**
  - Error models: correlate micro-architectural bit flips with architectural bit flips to tune rates of architectural injections
  - Different error site selection criteria, single vs. multi bit errors, etc.
  - Correlate application properties with SDC rates
  - More applications