Software-Based ECC for GPUs

Naoya Maruyama,
Akira Nukada,
Satoshi Matsuoka

Tokyo Institute of Technology

July 2009, SAAHPC'09 @ NCSA

Objectives

- Study software approaches for DRAM reliability in GPUs
- Understand performance implications of software-implemented ECC in GPUs

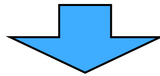
Contributions

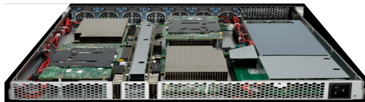
- First study of software ECC in GPUs
- Negligible to 70% performance overheads depending on memory-access intensity
  - ECC code generation is the most biggest performance bottleneck in the software approach

Background

Traditional GPUs

- Specially designed for 3-D graphics apps
- *Performance is much more important than reliability*



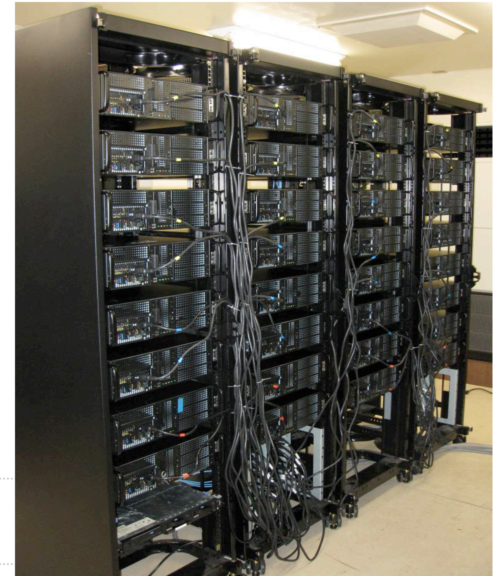


GPGPU

- New applications: bioinformatics, medical processing, etc
- Large-scale computing: Tokyo Tech TSUBAME (680 Tesla GPUs)
- *Reliability is very important*

- *Problem: Very limited reliability mechanisms in the current GPUs*
  - Transmission error detection in GDDR5 and PCIe
  - No error tolerance for bit-flips of data on DRAM, on-chip memory, and registers (i.e., no ECC for DRAM and shared memory)
- Example: DRAM errors
  - One of the most vulnerable components [Schroeder and Gibson, 2006]
  - More than 8% of DIMMs at Google exhibited bit-flip errors in a year [Schroeder et al., 2009]

Example: Transient DRAM Errors



- 72-hour run of our Memtest for CUDA on 60 GeForce 8800 GTS 512 GPUs
- All errors are silent; No visible abnormal behavior except for bit-flips

No errors on 52 GPUs

< 10 errors on 7 GPUs

57 errors!

Numb

0 10 20 30 40 50 60

0 10 20 30 40 50 60

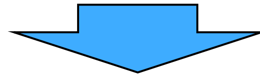
Number of Errors

Existing Techniques

- Spatial and temporal redundancy
  - Executes the same program multiple times using different devices or sequentially.
  - Can’t avoid >2x time/resource overhead in general ([Dimitrov et al., 2009])
- Information redundancy: EDAC coding
  - E.g., (72, 64) SEC-DED Hamming ECC
    - Add 8 bits per 64 bits for correcting a single-bit errors and detecting a double-bit error
    - Usually implemented by **hardware** in the ECC DRAM
  - Costs
    - Space overhead: one byte per 8 bytes (12.5%)
    - Time overhead: very cheap when implemented in hardware
  - More efficient than spatial and temporal redundancy when implemented in hardware

Our Approach

GPUs have massive processing performance



Software-Based ECC

– Check data on each memory access by using error-checking codes (ECC in particular, parity being evaluated)

– Overlap data accesses and data checking thanks to the highly multithreaded GPU architecture

– Enhance the DRAM reliability of commodity GPUs with no special hardware extensions

Performance?

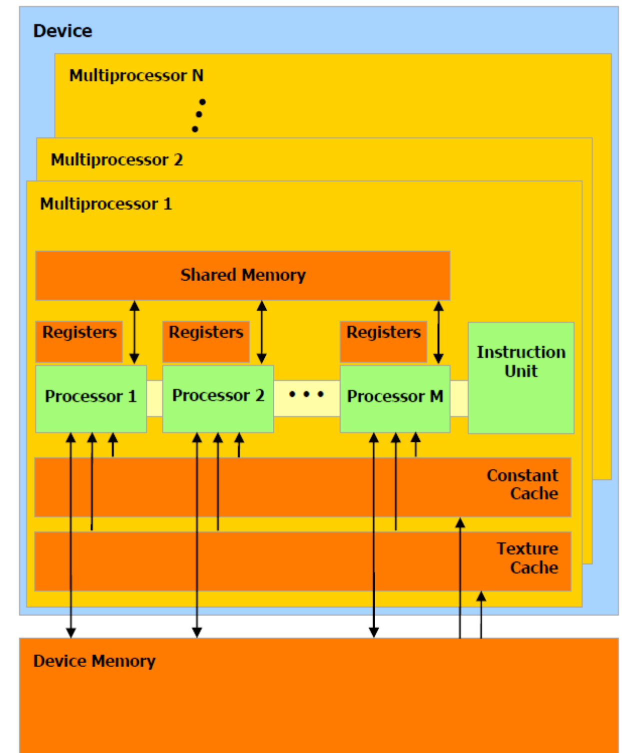
- Performance overheads
  - Bandwidth overheads → 1/8
  - Processing overheads → ECC coding may be too expensive

Hypothesis

1. *Compute intensive kernels*
   - Little impact by increase of memory access costs
2. *Bandwidth intensive kernels*
   - Unavoidable memory overheads as in the hardware ECC, but still only 1/8
   - Can hide the processing overheads by exploiting idle processing units?

Overview of NVIDIA GPUs

- Highly multithreaded architecture
  - 8 SPs per SM, up to 30 SMs per card (Tesla S1070 and GeForce GTX 285)
  - 355 giga operations per second (GOPS) in GTX 285
- Memory system
  - Up to 4GB of external DRAM
    - Optimized for non-random, block accesses
    - Accessible through several abstractions: global memory (RW), constant memory (R-only), texture memory (R-only)
    - Limited hardware cache
  - Small on-chip memory: shared memory (16KB per SM), registers (-16K per SM), constant and texture cache (6-8 KB per SM)



Fault Model

- Focuses on bit-flip errors in CUDA global memory
- Global memory
  - Located in external DRAM
  - The most general-purpose memory in GPUs
  - Theoretical peak bandwidth → 159GB/s (GTX 285)
  - Large latency (400 to 600 cycles)
    - Can be effectively hidden by multithreading
  - Not cached by hardware
- Assumes transient behavior
  - Errors are eliminated at the next write cycle
- Mostly 1-bit errors with rare 2-bit errors
  - 97% → 1-bit errors, 3% → 2-bit errors [Schroeder et al., 2009]

Error Detection and Correction for Global Memory

- Always have objects in global memory associated with EDAC codes
  - Store EDAC codes in global memory as well
  - Codes themselves are vulnerable to errors
  - True in the hardware implementation too
- Extends both host and device parts in CUDA apps

EDAC Codes [Fujiwara, 2006]

Faster → Safer

- Parity
  - 1 if number of set bits is odd
  - 2 * log(n) bitwise operations (n: data length)
  - Can detect 1-bit errors
- Hamming
  - 32-bit datum requires 6 bits; 64-bit datum requires 7 bits
  - Can correct 1-bit errors
- Error-Correcting Code (ECC)
  - Concatenation of parity and Hamming
  - Can detect 2-bit errors and correct 1-bit errors

Computing 64-bit ECC

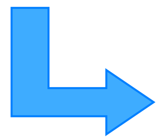
- 7-bit Hamming code + 1-bit parity
- Hamming code
  - Select 7 sub-bits from 64 bits
  - Compute parities for the sub-bits
- Error checking algorithm

```
if Hamming does not match then
    if parity matches then
        correct 1-bit error
    else
        detect 2-bit error
```

- Can be implemented with 63 32-bit int. ops

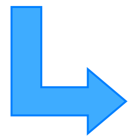
Host Part Extension: Pre Kernel Launch

1. Allocate global memory



Allocate code space for the data object
Size → 1/8 with (72, 64) ECC

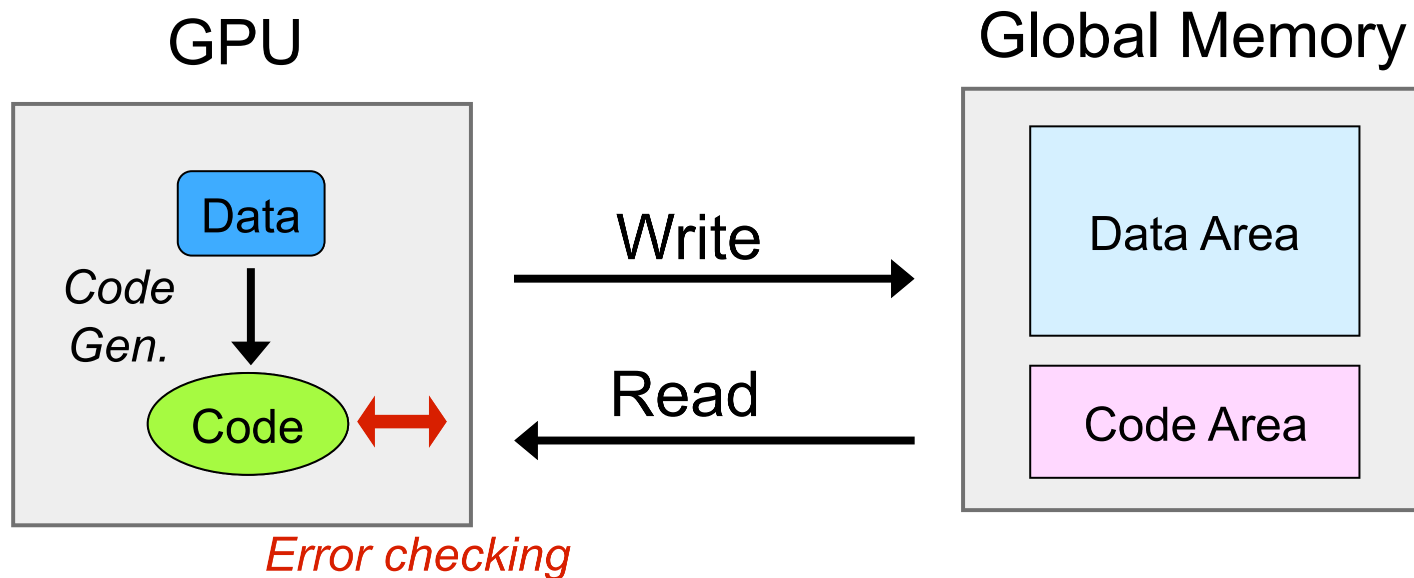
2. Transfer input data to the global memory



Generate and transfer the ECC of the input data

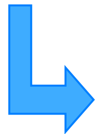
Kernel Extension

- Data stores → Save codes as well
- Data loads → Check the data using saved code



Host Part Extension: Post Kernel Launch

1. Output transfer to host memory

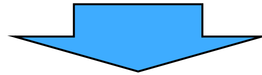


Scrub the output on global memory after it's transferred to host

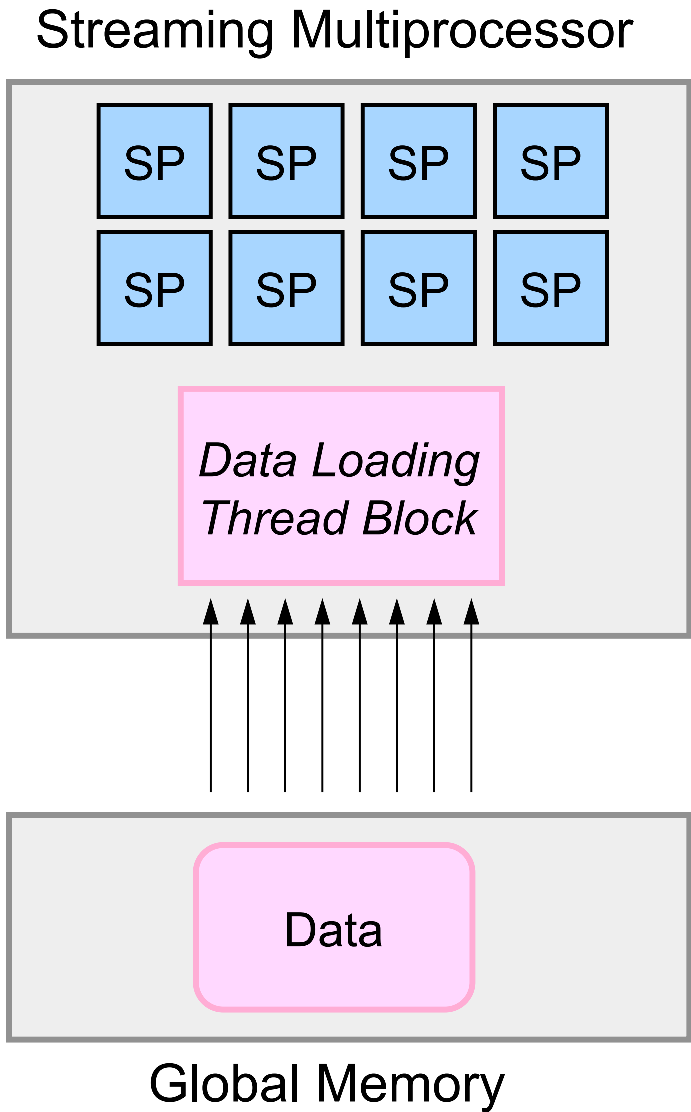
- Host error checking can be very costly
- Relies on PCIe's error detection

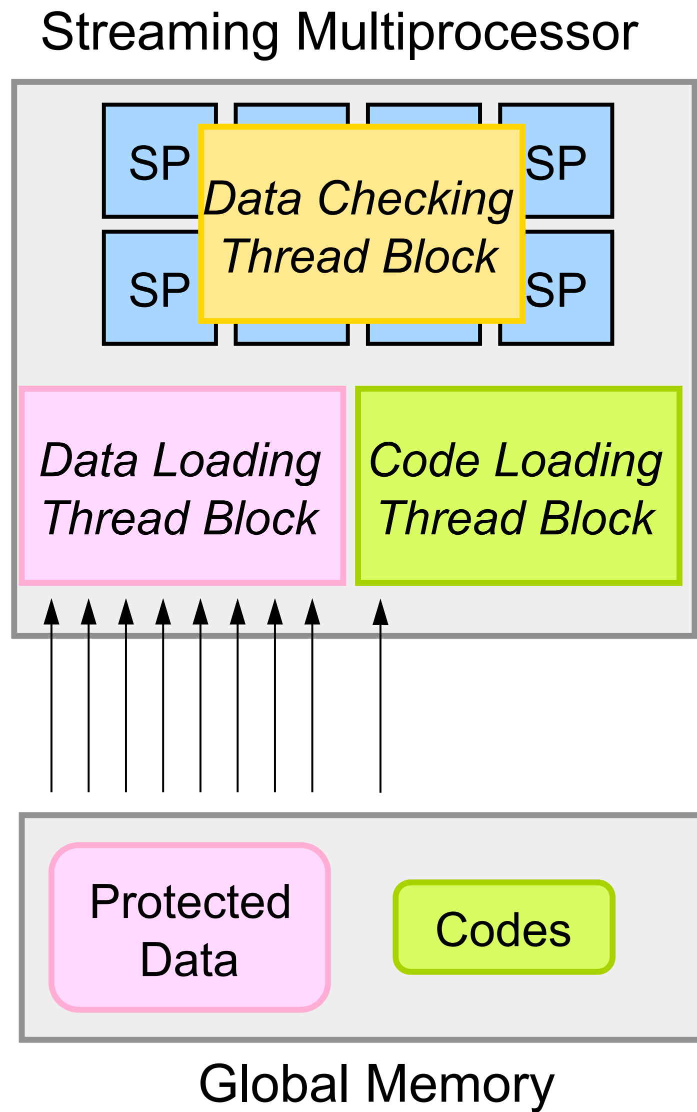
Optimization: Reducing Memory Transactions

- High latency of global memory accesses (400-600 cycles)
- Sub-word accesses are not efficient in CUDA



- Bundling multiple codes to a single word
  - A 32-bit word contains 4 8-bit ECC codes
- Only part of threads in a thread block load and store memory accesses for ECC codes
  - Arrange codes so that the accesses are coaleced


Original Data Loads
Streaming Multiprocessor
SP SP SP SP
SP SP SP SP
Data Loading Thread Block
Data
Global Memory


ECC-Protected Data Loads
Streaming Multiprocessor
SP SP
SP SP
Data Checking Thread Block
Data Loading Thread Block
Code Loading Thread Block
Protected Data
Codes
Global Memory

Prototype Implementation

- Possible Approaches
  - ECC library
    - Encapsulates ECC checking as library APIs
    - Programmers manually insert appropriate library functions.
    - Simple to implement
    - Needs manual program modification
  - Automatic translation of unsafe to safe codes
    - e.g., normal ptx → ptx w/ ECC checking
    - Not so simple to implement
    - Simple to use

Case Study : Matrix Multiplication

- Based on the CUDA SDK sample code
  - A x B = C
  - Each thread computes a single element of matrix C
- Modification
  - Error checking upon loading elements form matrix A and B
  - Saves ECC as well as the computed result

```
matMul(float *A, unsigned *A_code,
      float *B, unsigned *B_code,
      float *C, unsigned *C_code){
  shared sA[][];
  shared sB[][];
  float sum = 0.0;

  for (i, j in BLOCK) {
    sA[i][j] = A[i][j];
    check_float32(sA[i][j]);
    sB[i][j] = B[i][j];
    check_float32(sB[i][j]);
    for (k, l in block) {
      sum += sA[k][l] * sB[k][l];
    }
  }
  C[i][j] = sum;
  write_check_code32(sum);
}
```

Case Study： N-Body Problem

- Based on the CUDA SDK sample code
  - Each thread updates a single body
  - Each body consists of 4 floats for each of the position and velocity
- Modification
  - Error checking upon loading the current values of bodies
  - Generates ECC when updating the location and velocity of each body

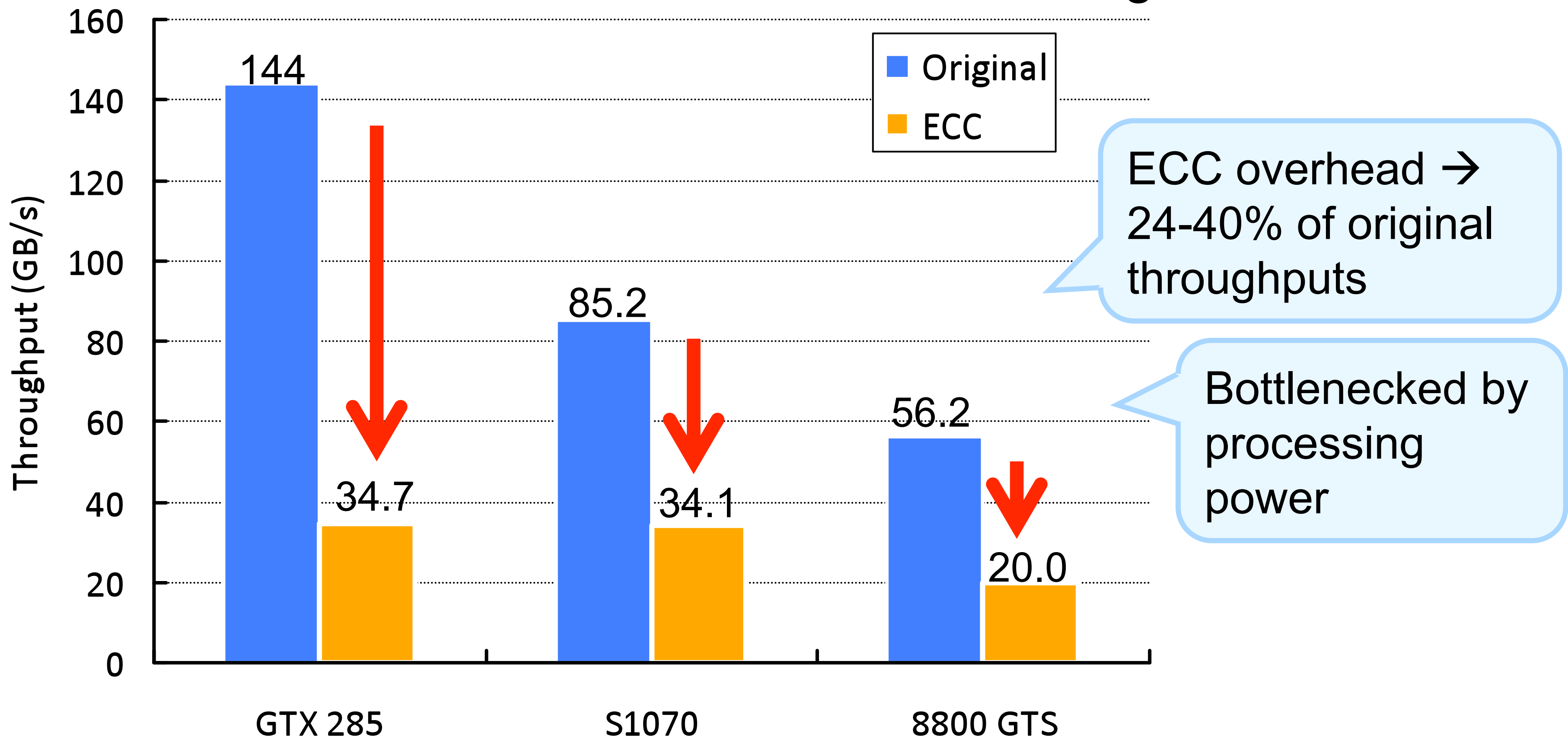
```
nbody (float *pos, unsigned *pos_code,
       float *vel, unsigned *vel_code) {
  float4 *mypos = pos[i];
  check_float4(mypos, i);
  for (j in BLOCK) {
    shared float4 block[];
    block[k] = pos[k];
    check_float4(block[k], k);
    for (pt in block_size) {
      acc+=compute_acc(mypos,block[pt]);
    }
  }
  update_pos(pos, vel);
  write_check_code_float4(pos);
  update_vel(vel, acc);
  write_check_code_float4(vel);
}
```

ECC Performance Evaluation

- Experimental Setting
  - GeForce 8800 GTS 512 + CUDA 2.1
  - Tesla S1070 + CUDA 2.0
  - GeForce GTX 285 + CUDA 2.1
- Benchmarks
  - Memory read throughput
  - N-body problem
  - 3D FFT of 256^3

Memory Throughputs w/ ECC

- All threads read 32-bit data from global memory to registers
- ECC for each 64-bit datum
- Overhead: ECC calculation + ECC loading



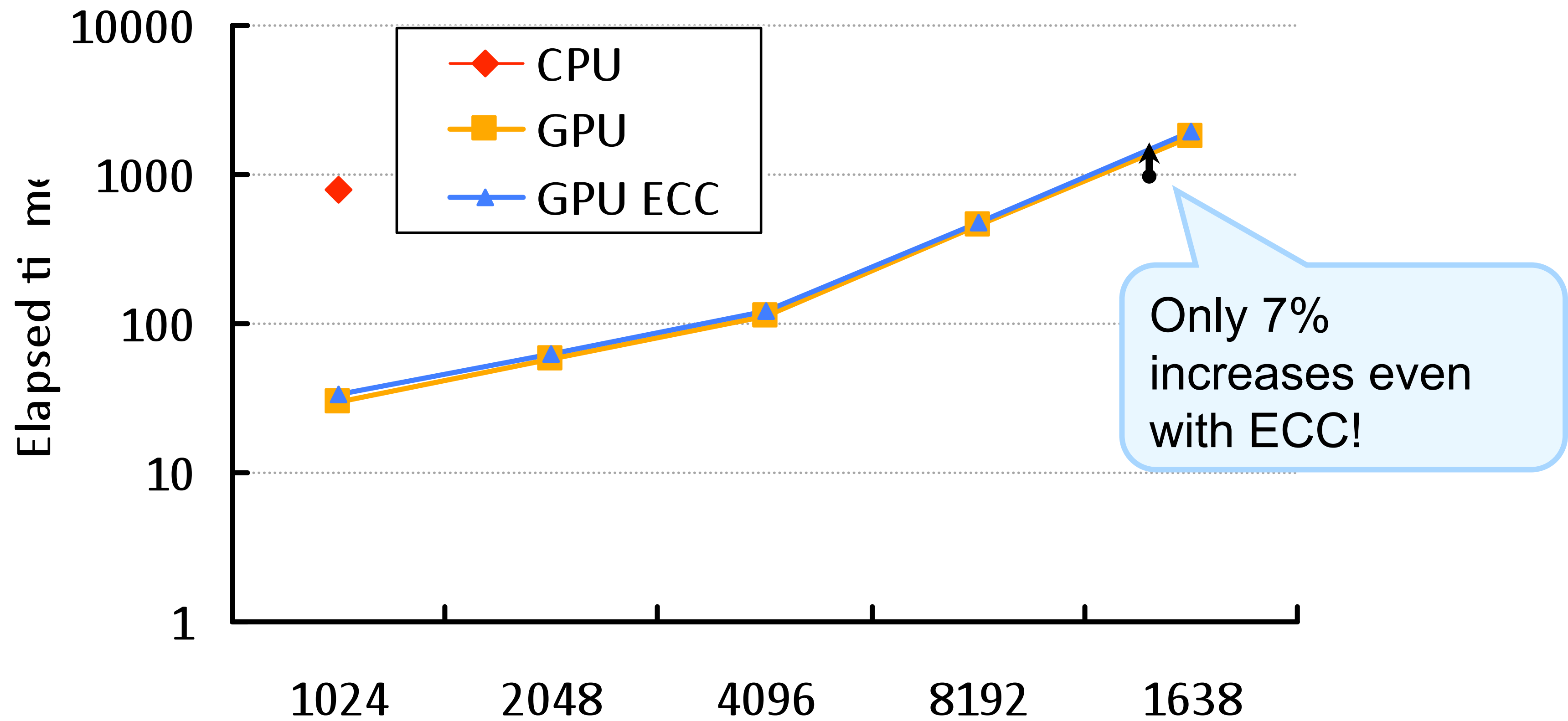
Throughput Modeling

- ECC for 64-bit datum → 63 32-bit integer ops → 8 ops/byte
  → ECC protection will be sustained if compute/ bandwidth is greater than 8
- Tesla S1060 (1GPU)
  – 360 G Int. Ops/s & 100 GB/s → 45 GB/s (= 360 / 8) w/ ECC
- GeForce GTS 512
  – 200 G Int. Ops/s & 62 GB/s → 25 GB/s (= 200 / 8) w/ ECC?

→Throughput would be around 40% w/ ECC

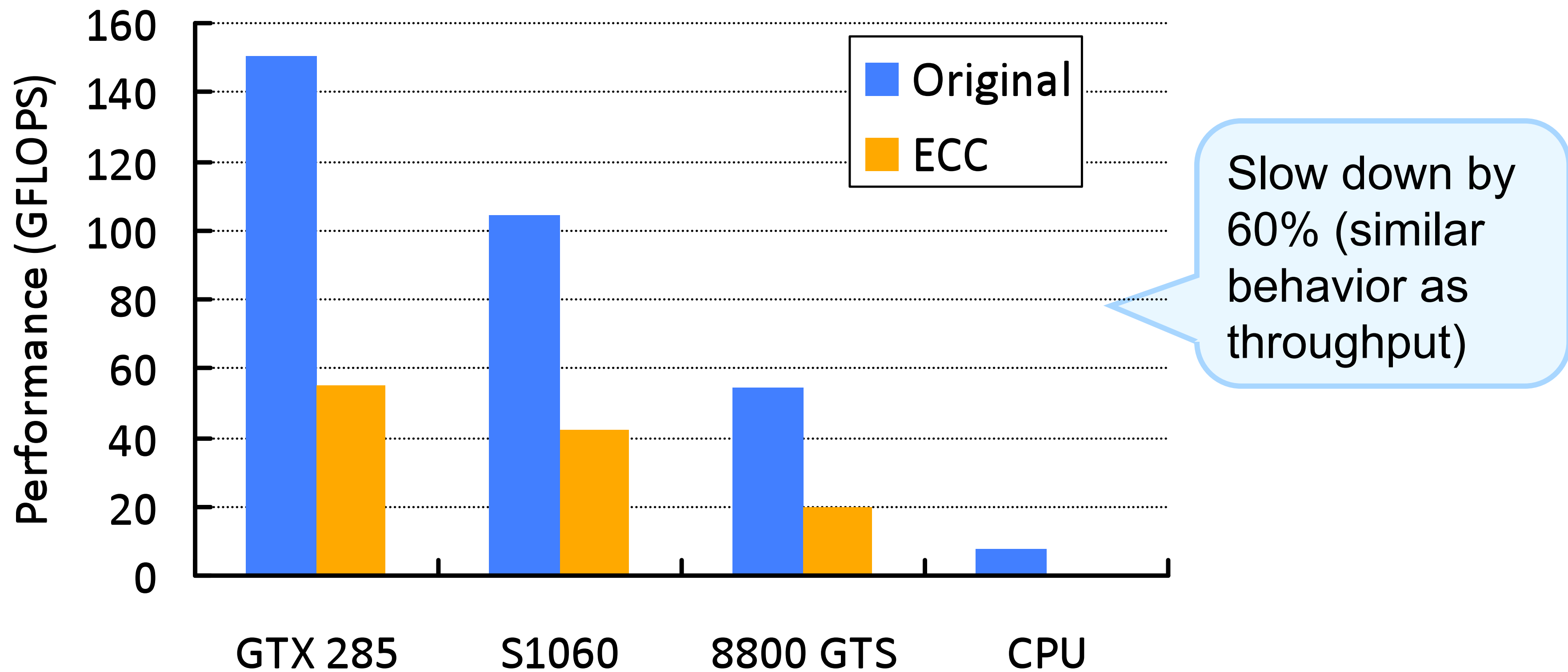
N-Body Problem

- CPU vs GPU vs GPU+ECC
- Extends the sample implementation in the CUDA SDK



3-D FFT (256^3)

- Original Code: NUFFT
  - Bandwidth-intensive algorithm by Nukada
  - Much faster than CUFFT (<= 2.2) [Nukada et al., SC'08]



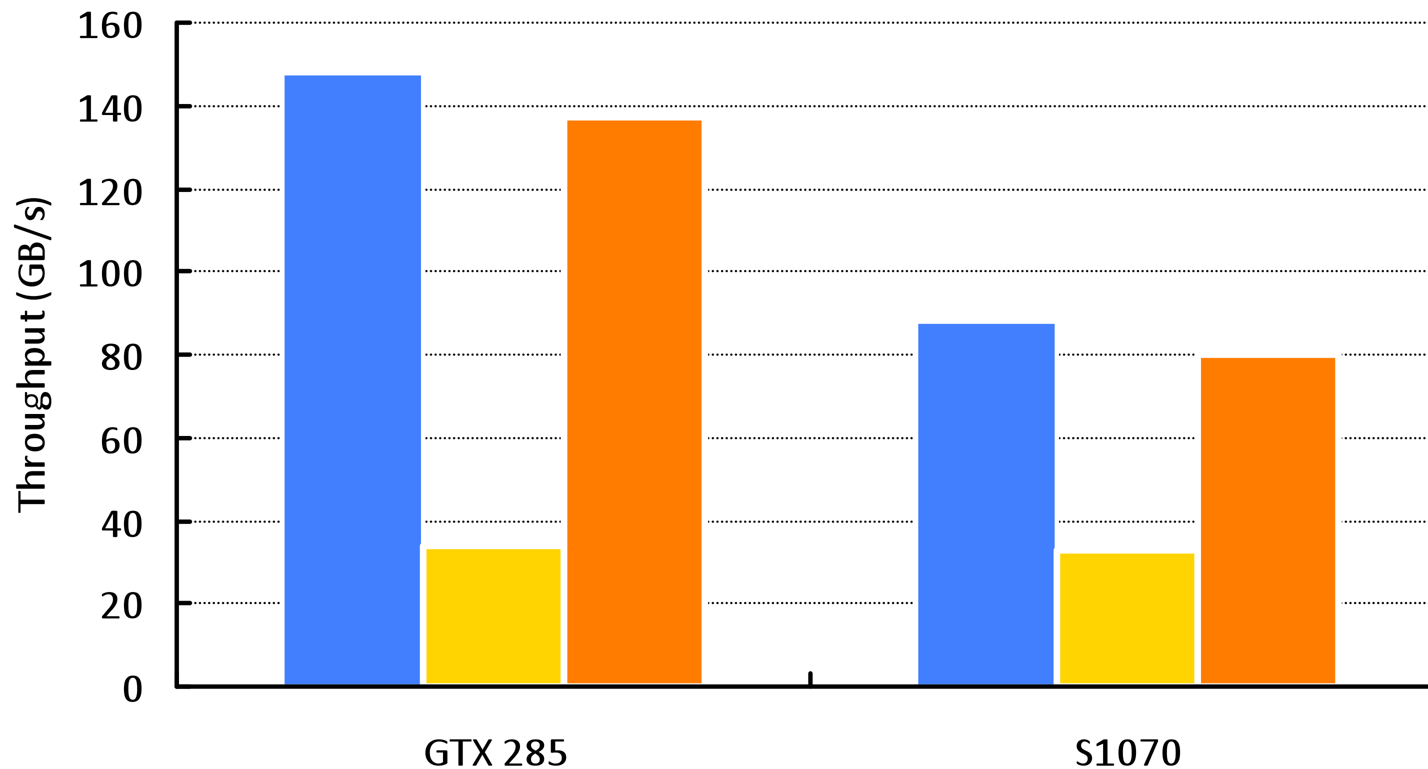
Related Work

- Shirvani et al., 2000
  – Software ECC for CPU
  – In-kernel process scrubs memory pages at idle times
- Sheaffer et. al, 2007
  – Redundant execution by a small hardware extension
  – DRAM is not protected
- Dimitrov et al., 2009
  – Performance evaluation of software-based redundant execution
  – 2x overhead in NVIDIA GPUS

Conclusions

- Preliminary studies of software ECC for GPUs
  - 24%-40% of throughputs
  - Significant slowdown with bandwidth-intensive kernels (e.g., 3-D FFT)
  - Only a few percent of overheads in compute-intensive kernels (e.g., the N-Body problem)
- Future work
  - Develop lighter-weight coding
  - GPU checkpointing for more comprehensive fault tolerance in large-scale GPGPU

Recent Results: Memory Throughput Improvements



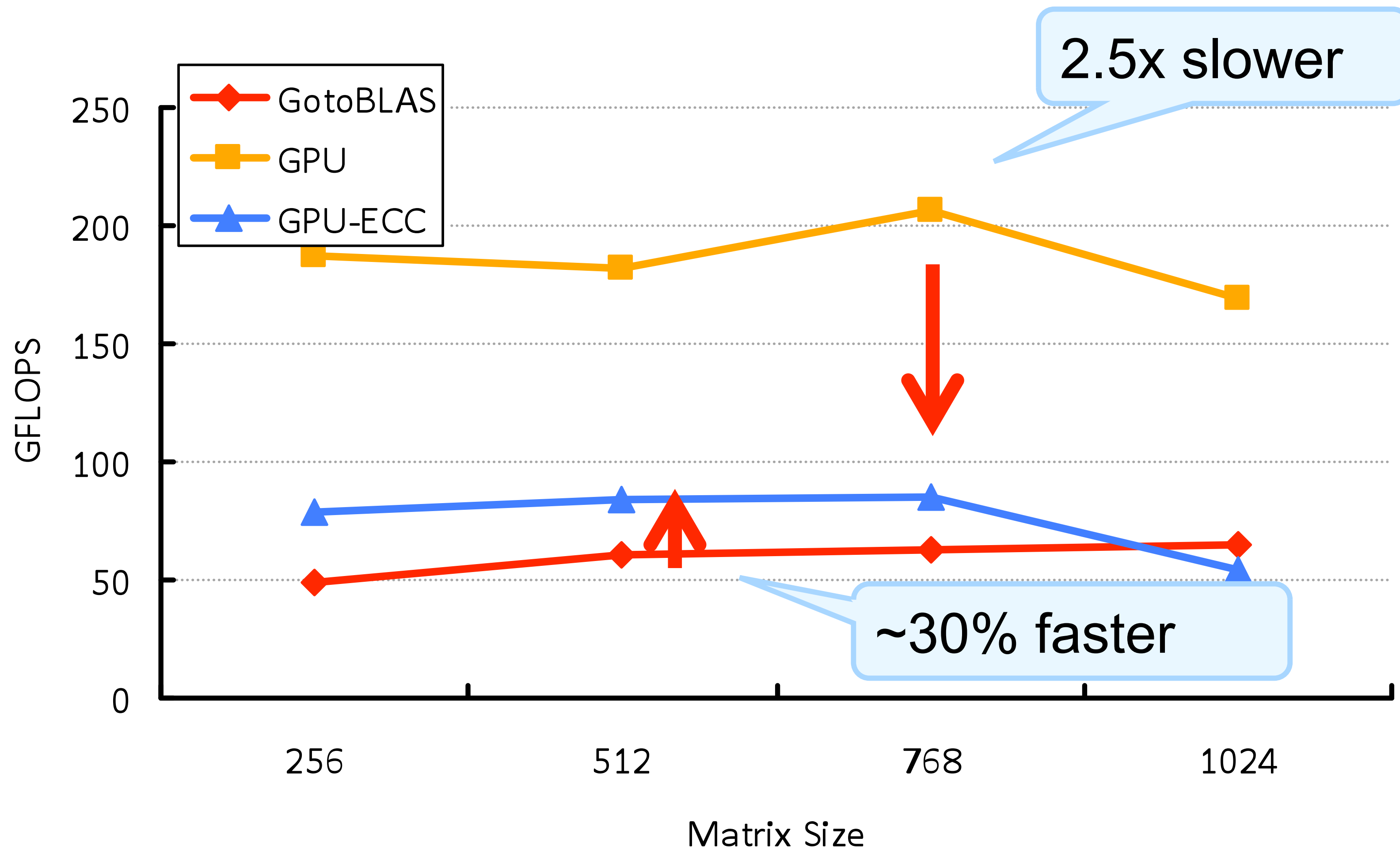
A technical report with more details will be available soon
(contact us: naoya@matsulab.is.titech.ac.jp)

Backup slides

GPU Memory Vulnerability Analysis

Matrix Multiplication

CPU (GotoBLAS) vs GPU vs GPU+ECC



CUDA Programming Model

- Typical execution flow
    1. Transfer inputs to GPU global memory
    2. Launch kernel functions working with the input data
        1. Load data from global memory to registers and shared memory
        2. Do some computation
        3. Store the results to global memory
    3. Transfer back the results to host DRAM
- Vulnerabilities in CUDA
    - Soft errors in global memory

Possible Vulnerabilities

Expectedly the most frequent

- DRAM soft errors
  - Can be caused by radiation events (neutron strikes from cosmic rays, alpha particles in packaging materials.
  - Other suspected causes: Power fluctuations, aggressively high-speed memory configurations, too high temperature, etc.
- Soft errors in on-chip memory
- Soft errors in arithmetic units
- PCIe data transfer
- Permanent (hard) errors