A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs

Naoya Maruyama, Akira Nukada,
Satoshi Matsuoka

Tokyo Institute of Technology

April 2010, IPDPS 2010 @ Atlanta

Background

Traditional GPUs

- Designed for 3-D graphics apps
- *Performance is the most important*

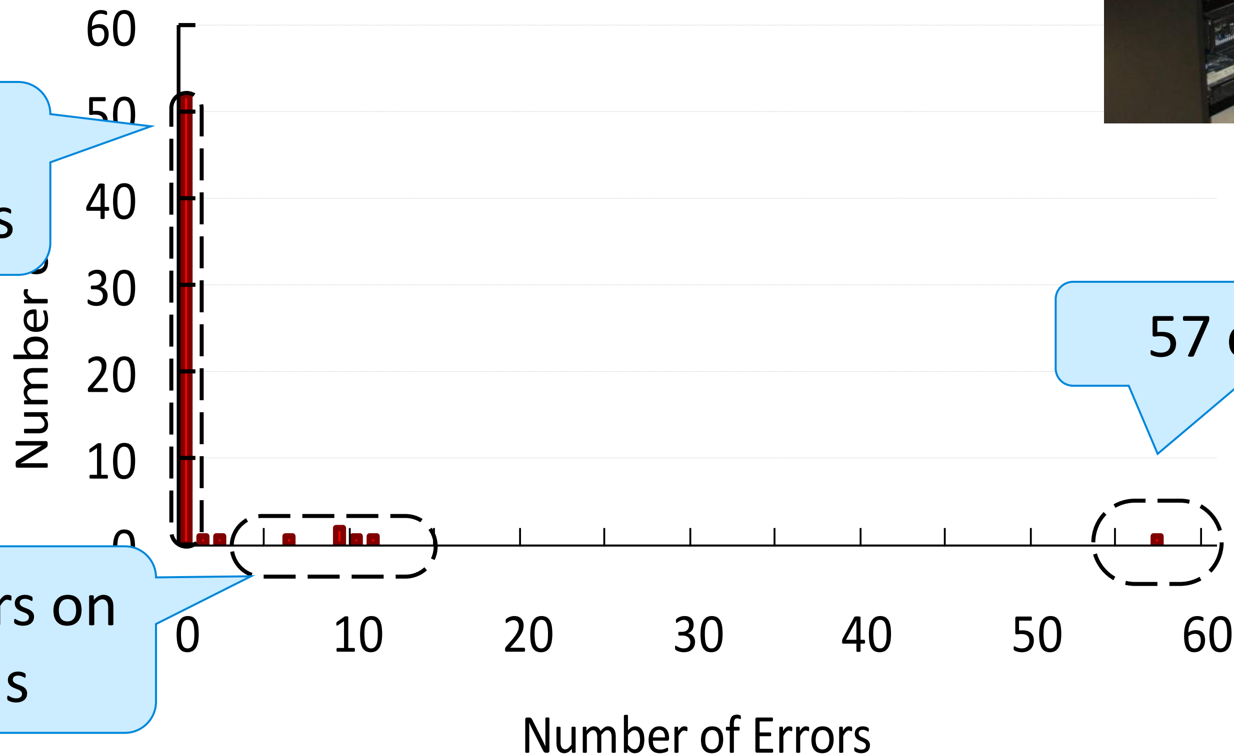
HPC GPGPU

- Scientific applications
- Large-scale computing: Tokyo Tech TSUBAME (680 Tesla GPUs)
- *Reliability is very important*

- *Problem: Very limited reliability features in the current GPUs*
  - No ECC for GPU DRAM except for the new Tesla GPUs (Fermi)
- DRAM errors are one of the most vulnerable components [Schroeder and Gibson, 2006]

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



Proposal

A software framework for DRAM bit-flip errors

- Lightweight runtime error checking by exploiting the latency hiding capability of GPU

Performance Results:

- < 5% in matrix multiplication
- 35% in 3D FFT

Talk Outline

1. Background
2. Proposal
3. Coding
4. Fault Tolerant Framework
5. Case studies
6. Conclusions

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)
- 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]

Fault Tolerance against Memory Errors

1. Temporal or spatial redundancy
   – Requires more than 2x resources

2. Information redundancy
   – E.g., ECC in standard server DRAM
   – Costs of ECC
      - Memory capacity → 1/8
      - Coding time → ???

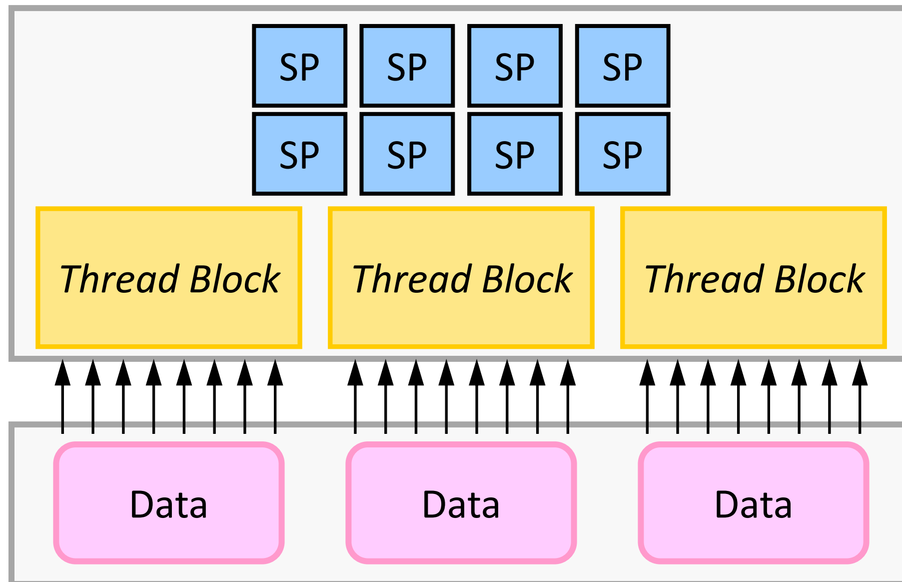
Approach: *Poor Man's ECC*

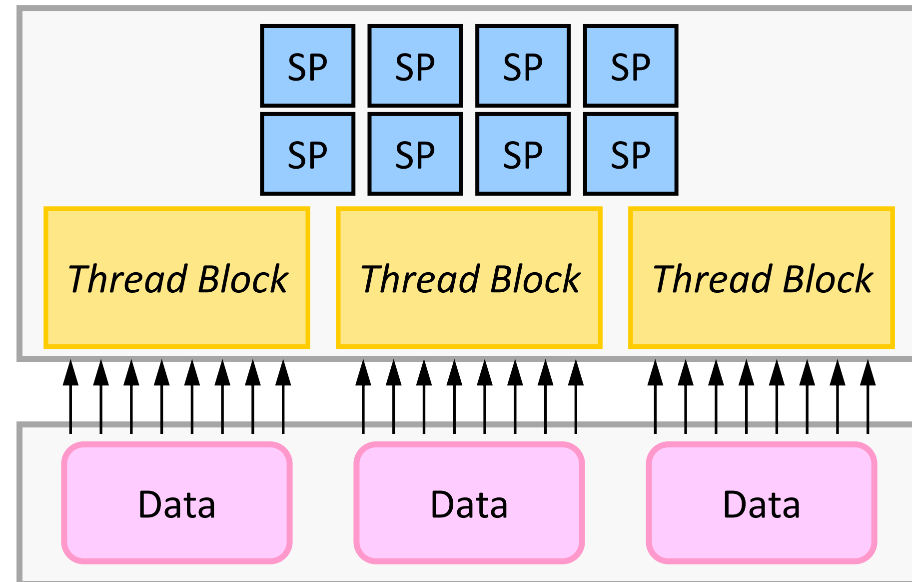
- Modify program code:
  - To generate error checking code on memory writes
  - To check data on memory reads by using error-checking code
- Potentially degrade memory latency and bandwidth!

- Exploit the latency hiding capability of GPU memory system
  - GPU memory system
    - Highly tolerant to latency by multiplexing a large number of concurrent threads
    - One of the key architectural features of the GPU
  - Optimization by overlapping data accesses and data checking thanks to the highly multithreaded GPU architecture

Exploiting Latency Hiding Capability

- Issues lots of loads and stores from a large number of threads to maximize throughput
- **Idle** compute units during memory accesses in bandwidth-bound kernels
  - Post-processing for free!



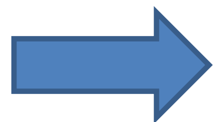


Performance Model of Error Checking

- Assumption

  Computation and DRAM accesses can be completely overlapped

c	#ops/byte for coding	M	DRAM throughput
f	storage overhead factor	M'	DRAM throughput with error check
P	#integer ops/s		



$$M' < \min\{\frac{M}{f}, \frac{P}{c}\}$$

Coding Throughput ($\frac{P}{c}$)

Data access throughput ($\frac{M}{f}$)

Performance Model of (72, 64) SEC-DED ECC

$$M' < \min\{\frac{M}{f}, \frac{P}{c}\} = \min\{\frac{8M}{9}, \frac{P}{12}\}$$

c: 12

f: 9/8

GeForce 285 GTX: $M = 159\text{GB/s}$

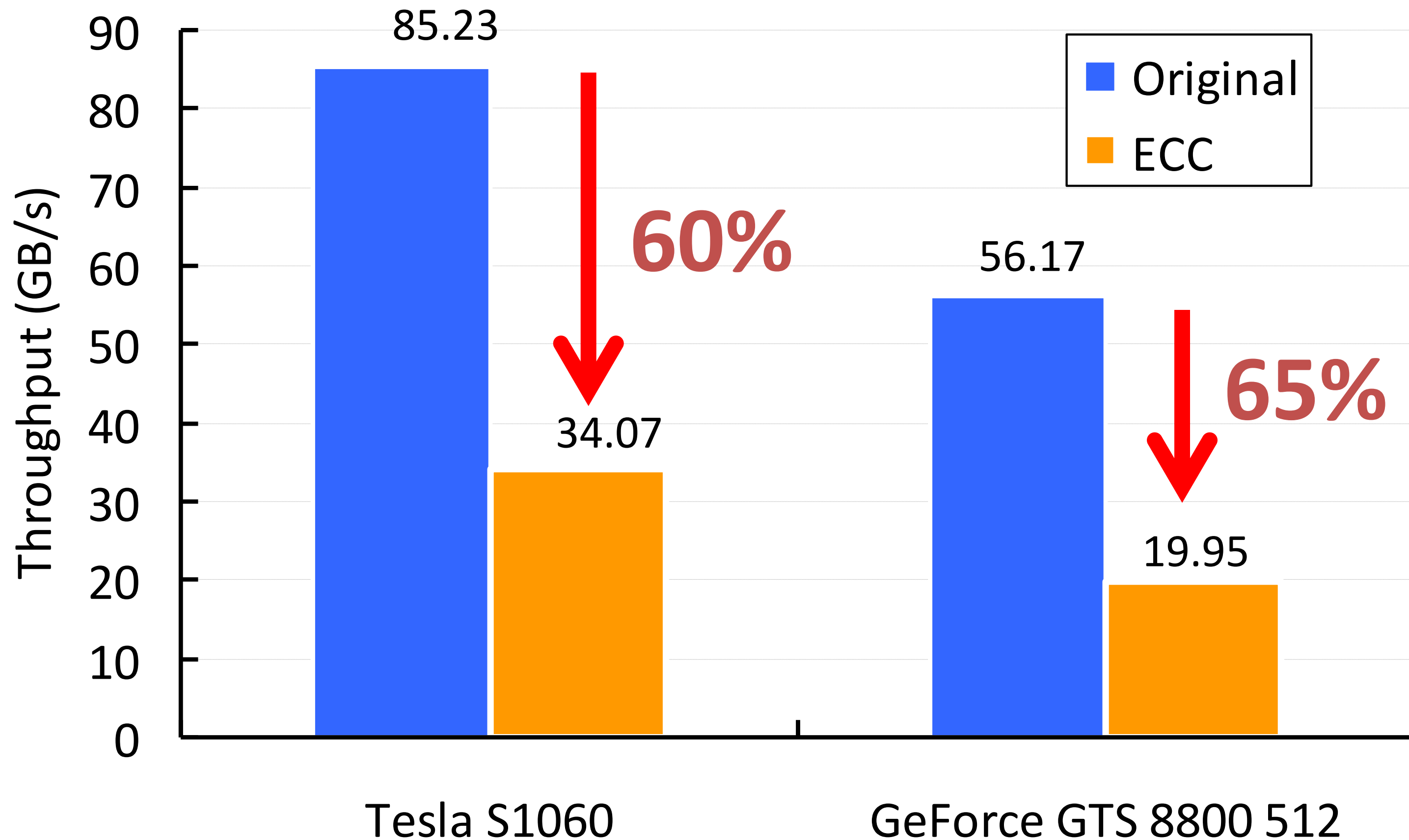
$P = 350\text{GOPS}$

Estimated

$$M' < \min\{141, 29\} = 29\text{GB/s}$$

ECC Coding Throughput

- Validates 64-bit data by (72, 64) Hamming ECC
- Overheads: ECC coding + Code loading



Coding Optimization 1

- Error detection with checkpoint/recovery
  - Correction is costly
  - Only detection by coding
  - Retrying by keeping input data on host memory
- Detection code
  - Parity → Can detect a one-bit error
  - *c*: 1.5 → M’ = min{141, 233} →No coding bottleneck
  - No two-bit errors are detected

Coding Optimization 2

- *Observation*
  - Typical DRAM access granularity >> 8 bytes
- *Idea*: Blocking
  - Encodes in parallel a block of data accessed by all threads in a thread block
  - A block size is typically larger than 64B in the current CUDA GPU because of the coalescing
- *Implementation:* Modified Cross Parity
  - Consists of vertical and diagonal code
  - Can detect one-bit errors in a word and two-bit errors in a block of words

Modified Cross Parity: Vertical Code

- Parity of each N-th bit of words
- Detects a 1-bit error in N-bit words

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

0	1	2	3	4	5	6	7

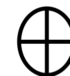
xor of eight words

Modified Cross Parity: Diagonal Code

- Parity of rotated N-bit words
- More efficient than horizontal parity
- Detects a 2-bit error in a NxN block

0	1	2	3	4	5	6	7
7	0	1	2	3	4	5	6
6	7	0	1	2	3	4	5
5	6	7	0	1	2	3	4
4	5	6	7	0	1	2	3
3	4	5	6	7	0	1	2
2	3	4	5	6	7	0	1
1	2	3	4	5	6	7	0

0	1	2	3	4	5	6	7



xor of rotated eight words

Performance Model of Cross Parity

$$M' < \min\{\frac{M}{f}, \frac{P}{c}\} = \min\{\frac{8M}{9}, \frac{P}{1.7}\}$$

c: 1.7

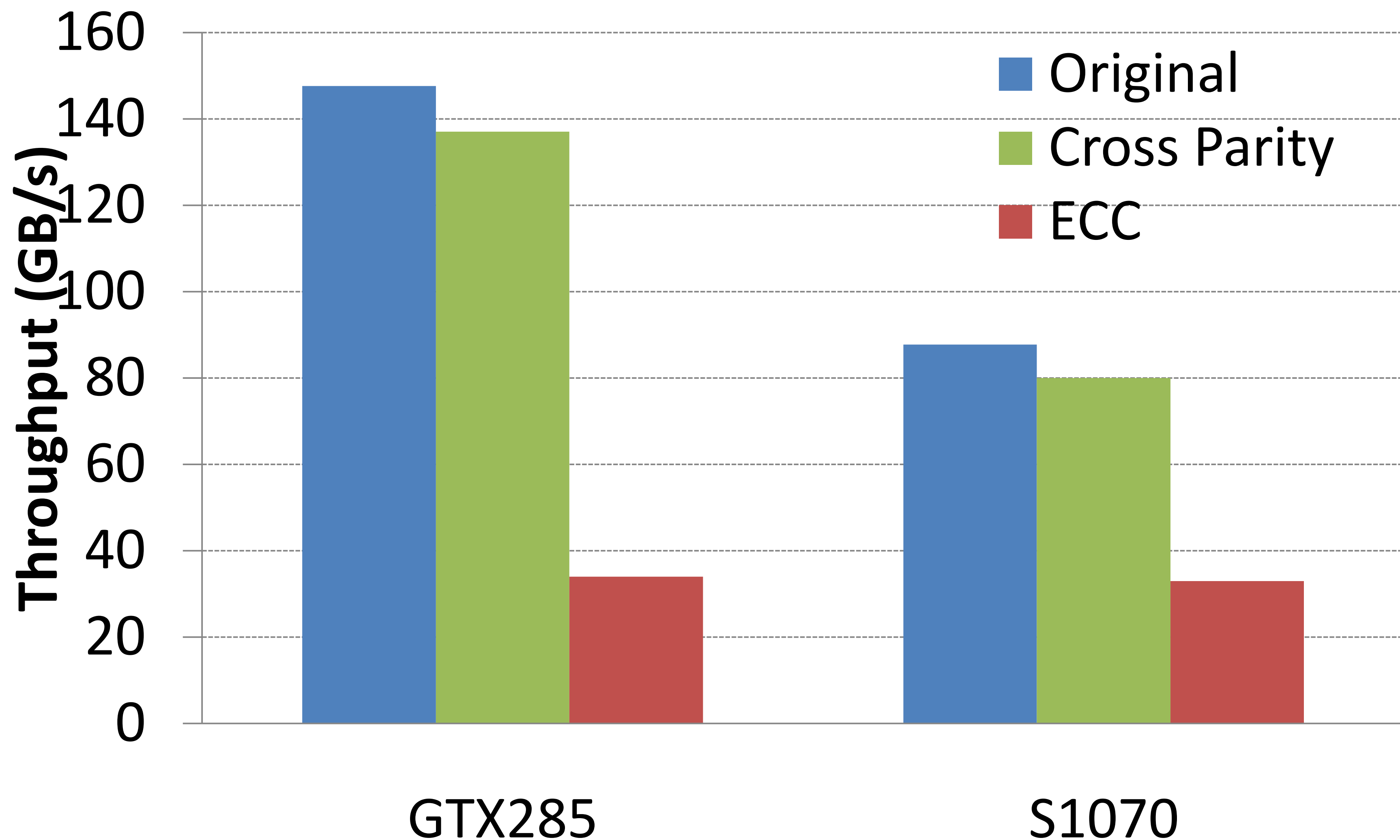
f: 9/8

GeForce 285 GTX: $M = 159\text{GB/s}$

$P = 350\text{GOPS}$

No Coding Bottleneck

Improved Coding Throughput

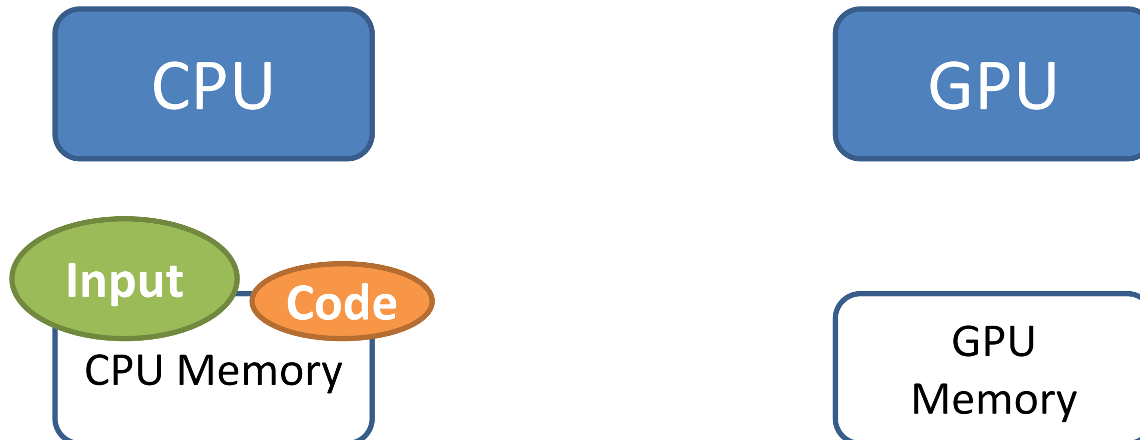


Realizing Memory Error Tolerance

- Framework to apply the cross parity and GPU checkpointing
  - Error detection by the cross parity (N=32)
  - Kernel recovery by checkpointing
- Divides CUDA apps into three major stages
  1. Input transfer
  2. Kernel execution
  3. Result transfer

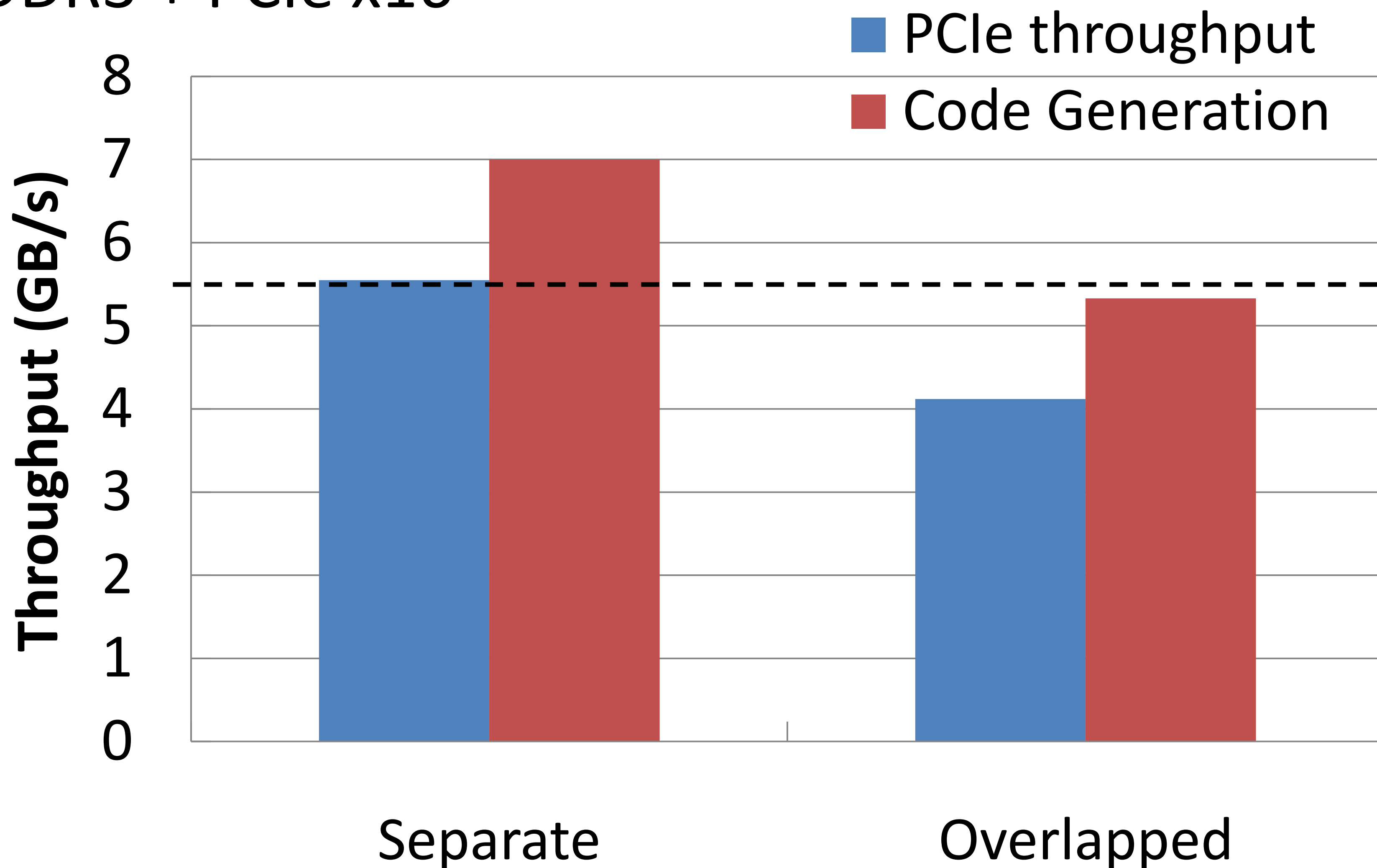
Input Transfer

- Generate code for input data on host CPUs
  - Transferred data to GPU gets susceptible to errors
- Performance cost
  - Decreased PCI throughput by 1/8
  - Code generation on host CPUs (limited by memory BW)
    - Mostly irrelevant by overlapping code generation with the transfer of the original data



Transfer Performance

- DDR3 + PCIe x16



Kernel Execution

- Embeds error checking at memory access sites in CUDA source code
  - Currently manually modified to call appropriate API routines
  - Could be automated with compiler-based source translation (now in progress)

Example: Matrix multiplication (CUDA SDK)

```
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);
}
```

Result Transfer

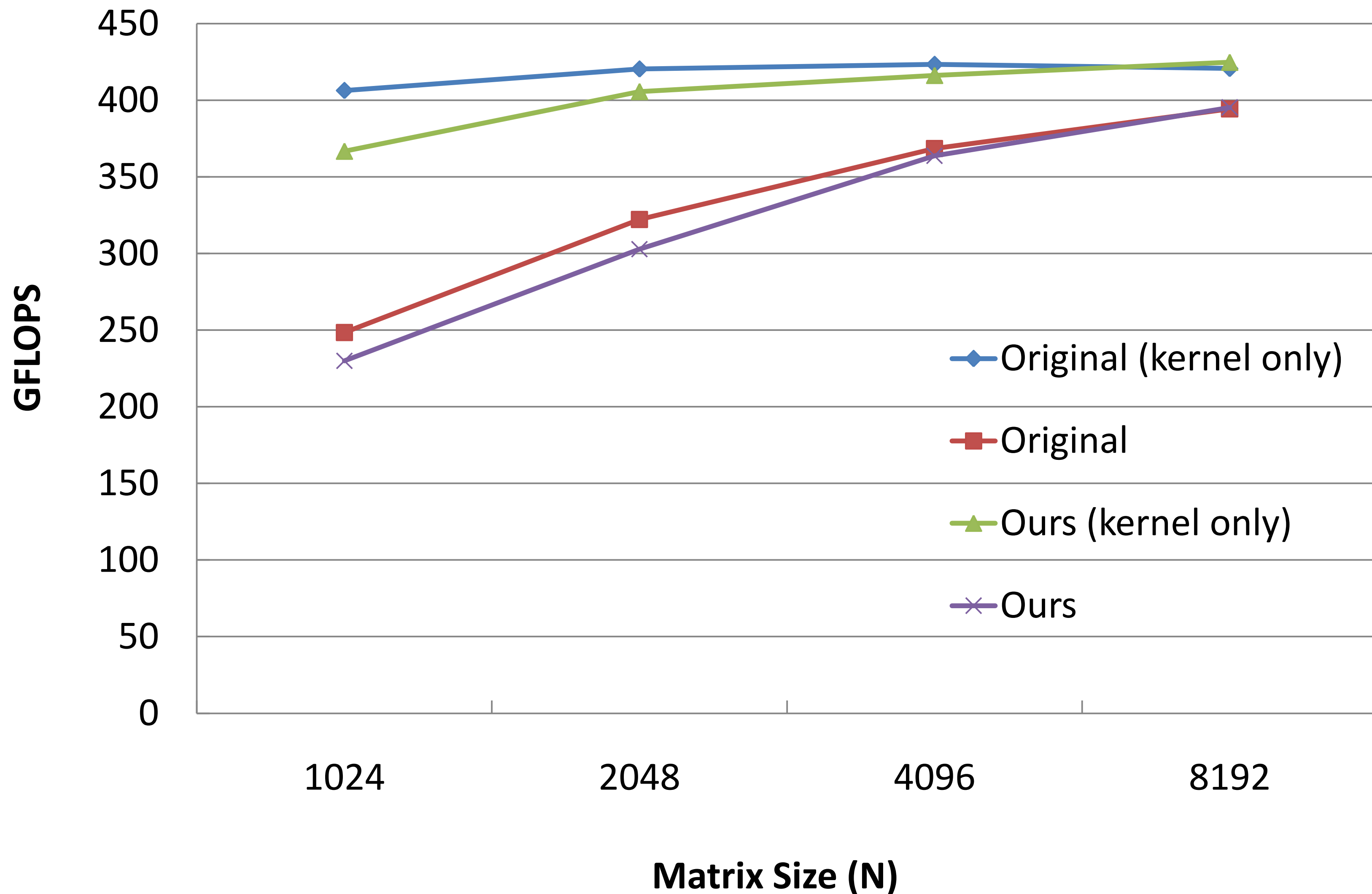
- Final data are usually transferred back to host memory
- Potential errors during the transfer on GPU memory
- Solution: Run another kernel to check the correctness of the data on GPU memory
  - Adds another performance overhead
  - But much smaller than the result transfer time
    - Error checking throughput → 100 GB/s
    - PCIe throughput → < 8 GB/s

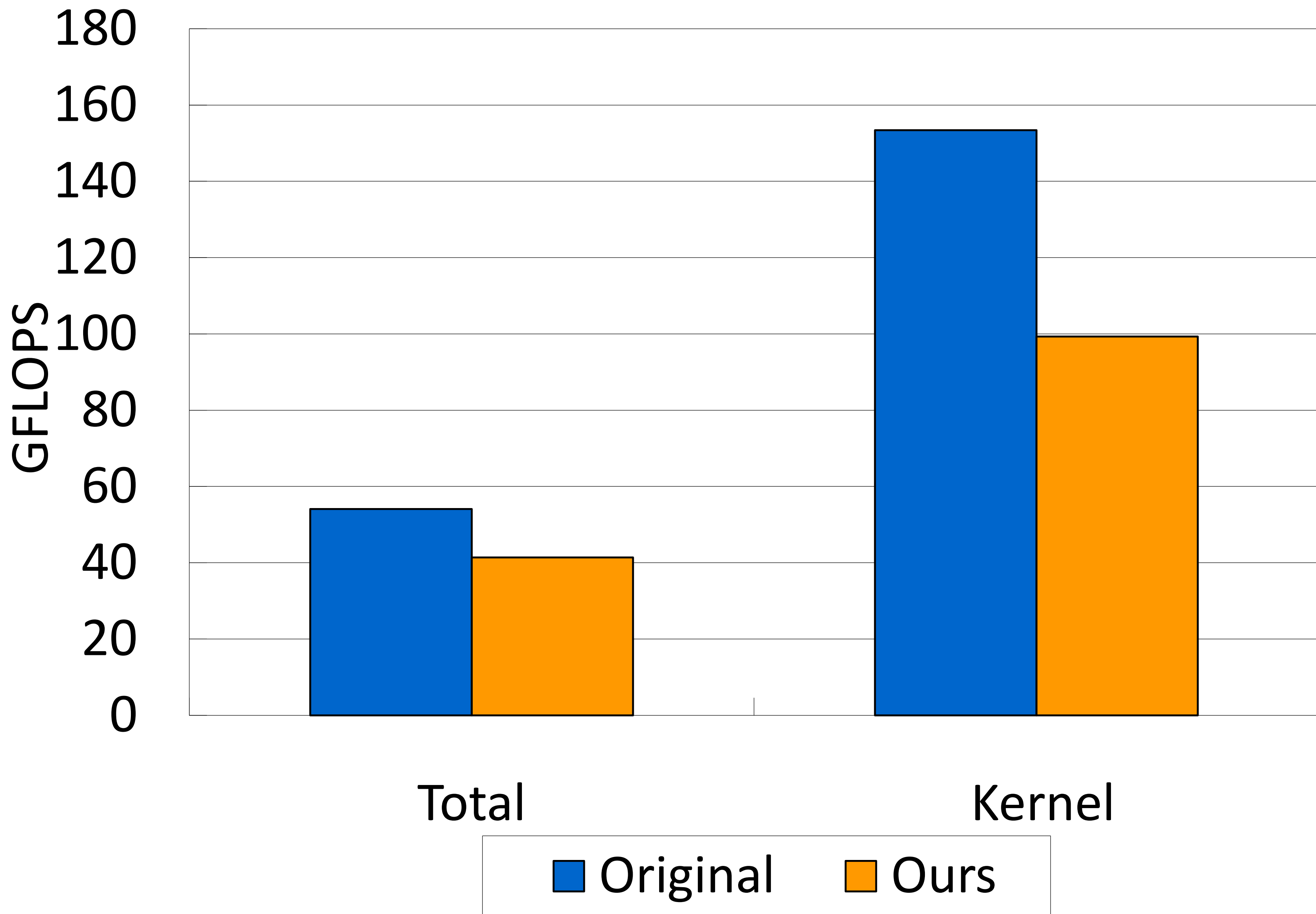
Cost < 10%

Performance Evaluation

- AMD Phenom II X4 955, 790FX, 8GB DDR3 memory
- GeForce GTX 285
- Fedora 10, NVIDIA Driver 190.18, GCC 4.3.2, CUDA 2.3

Matrix Multiplication





3D FFT (256³)
GFLOPS
180
160
140
120
100
80
60
40
20
0
Total
Kernel
Original
Ours

Conclusion

- Proposed a software framework for DRAM bit-flip errors
- Performance results
  - < 5% in matrix multiplication
  - 35% in 3D FFT
- Other applications of the latency hiding capability
  - Compression?
  - Encryption?

Extra Slides

GPU Memory System

- Highly tolerant to latency by multiplexing a large number of concurrent threads
  - Exploits massively multithreaded architectures
  - One of the key architectural features of the GPU
- Works great for bandwidth-intensive applications with regular data access patterns

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