Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications

Matrix Multiplication on GPUs with On-line Fault Tolerance

Chong Ding, Christer Karlsson, Hui Liu, Teresa Davies, and Zizhong Chen
Mathematical and Computer Sciences
Colorado School of Mines
Golden, CO, USA
Email: {cding, fckarlsso, huliu, tdavies, zchen}@mines.edu

***Abstract*—Commercial graphics processing units (GPUs) prove their attractive, inexpensive in high performance scientific applications. However, a recent research [1] through Folding@home demonstrates that two-thirds of tested GPUs on Folding@home exhibit a detectable, pattern-sensitive rate of memory soft errors for GPGPU. Fault tolerance has been viewed as critical to the effective use of these GPUs. In this paper, we present an on-line GPU error detection, location, and correction method to incorporate fault tolerance into matrix multiplication. The main contribution of the paper is to extend the traditional algorithm-based fault tolerance (ABFT) from offline to online and apply it to matrix multiplication on GPUs. The proposed on-line fault tolerance mechanism detects soft errors in the middle of the computation so that better reliability can be achieved by correcting corrupted computations in time. Experimental results demonstrate that the proposed method is highly efficient.**

Keywords-Soft Errors; Fault Tolerance; GPUs; Matrix Multiplication.

I. INTRODUCTION

Massive graphics processors have been successfully demonstrated to accelerate a wide variety of HPC applications in several domains, such as physical simulations [2, 3], bioinformatics [4], and medical analysis [5]. A lack of error checking and correcting (ECC) capability in the memory subsystems of many graphics cards is cited as a hindrance to the acceptance of GPUs as high performance coprocessors. The paper published by Stanford University on 2010 presented the first large-scale study of error rates in GPGPU hardware that approximately two-thirds of tested cards exhibited a pattern-sensitive susceptibility to soft errors in GPU memory or logic, confirming concerns about the reliability of the installed base of GPUs for GPGPU computation [1]. Most of the time, the traditional dominant applications of GPUs focus on video or image processing such as 3-D graphics games, favor performance over reliability [6].Fault tolerance seems not that essential for those applications, because it's only a pixel that appears for 1/30 of a second, nobody cares about the bit error. However, the scientific applications such as bioinformatics and medical analysis standing on massive data processing requires more reliability even with overheads to implement. The error detecting and correcting mechanism of the current generation of GPUs is so limited that no such mechanisms have been integrated in GPU memory systems [6] until the latest generation of GPU is released in 2011 by NVIDIA called Fermi which has the ECC mechanism.

Soft error can be hazardous to scientific computation on CPU or GPU and therefore needs to be well addressed by designing an efficient mechanism. This paper addresses the reliability issue of GPGPU and focuses on matrix multiplication. In the particular mechanism discussed in this paper, a software-only technique can detect corrupted data such as DRAM bit-flip error in the middle of computation and recover it. This kind of error doesn't fail the device which can continue to compute, however the error is kept in the result or is even propagated to make more incorrect data in result.

We apply our technique mainly on matrix multiplication in this paper, but it's not limited to this, it can also be extended to other matrix operation such as LU factorization. In our approach, we construct column/row checksum matrix for two input matrices and get product matrix with full checksum. The outer product version matrix multiplication is used which gains us several advantages that can't be reached by other existing approaches tolerating continue failures. During computation, the partial product matrix is scanned frequently so that corrupted data can be detected and recovered immediately in case it propagates. It's the first time to demonstrate that, for the blocked outer product version matrix multiplication algorithm, it is possible to maintain the checksum relationship in input checksum matrices and the accumulated partial product for every step of computation, not only in the end of multiplication. Based on this maintained checksum relationship during computation, we demonstrate that it is possible to tolerate intermediate errors in the blocked outer product version matrix multiplication, which increases reliability with low overhead.

Despite the fact that there has been much research on algorithm-based fault tolerance [16] in which applications operate on encoded data to determine the correctness of some matrix operation, to the best of our knowledge, this is the first time applications operate on encoded data during computation to maintain the correctness of partial product. Periodical checking mechanism helps avoid multiple errors to accumulate. A working fault tolerant implementation of the matrix product based on CUBALS 3.1 (a BLAS library ported to CUDA) library's matrix multiplication (cublasSGEMM [21, 22]) is discussed in this paper. This implementation is with better fault tolerance capability compared to TMR and traditional ABFT.

978-0-7695-4428-1/11 $26.00 © 2011 IEEE
DOI 10.1109/ISPA.2011.50

II. RLATED WORK

Wherever Times is specified, this section briefly reviews fault tolerance work for GPU computing errors.

A. Triple modular redundancy (TMR)

Triple modular redundancy (TMR) is a classical hardware redundancy technique for fault tolerance. The process simply executes a given computation three times on different devices and checks and votes to confirm that the majority of the executions yielded the same result. TMR needs at least a factor of 2 or 3 in additional hardware redundancy to tolerate single module failures [16] and this mechanism is only capable of detecting transient faults, but not permanent faults as both executions take place on the same hardware units.

[10] examines several techniques for adding software-implemented hardware fault tolerance (SIHFT) to extend fault tolerance to GPUs and compare the relative performance overhead of each technique. This approach has such a high overhead, although it is the most intuitive and simple to implement and requires almost no changes in the existing hardware and software setups.

B. Checkpoint and Restart

Checkpoint is probably the most typical approach to tolerate failures in parallel and distributed systems. Even in fault tolerance for GPU transient error, this idea is still used, because this class of approaches is very general and able to tolerate the failure of the whole system. However, there are some limitations to this approach. There is often a trade-off between transparency and performance. Transparent system level approach usually introduces a higher performance overhead than non-transparent application-level approach, especially for GPU which cannot access the storage system directly and the overhead to send the large scale data back to CPU is large. Also, it generally needs stable storage to save a globally consistent state periodically.

[6] demonstrates a high-performance software framework to enhance commodity off-the-shelf GPUs with DRAM fault tolerance. It combines data coding for detecting bit-flip errors and checkpointing for recovering computations when such errors are detected.

C. Algorithm based fault tolerance

Algorithm based fault tolerance is an approach which is tolerant of fail-continue failure, in which, GPU continues to work but produce incorrect calculations due to some problems such as bit-flip error. In this approach, applications are modified to operate on encoded data to determine the correctness of some mathematical calculations. It's proved by [16] that fault tolerant capability can be added to many matrix operations by detecting and correcting corrupted data based on maintained checksum relationship in a low overhead. In CPU HPC field, this approach can be applied to linear algebraic computations and with a very low overhead. This should work for GPGPU as well, but so far such methods haven't been tried.

[20] illustrates another approach which is closely related to methods described in [16] to implement fault-tolerant high-performance matrix multiplication . They show how a comparison of $\upsilon^T C$ with $\upsilon^T AB$ and $C\omega$ with $AB\omega$ can detect and correct errors introduced in matrix C (where υ^T and ω are checksum vectors). The method is based on inner product version matrix multiplication whose drawback in fault-tolerance capability is that even the matrices A, B and C are partitioned and with partitioned checksum. Once the sub-matrices are done with calculation with any corrupted data occur, there is no mechanism existing to detect and correct them.

III. FAILURE MODEL

Bit-flip errors in the GPU DRAM are the main problem to be taken care of. So far no Error Detection and Correction Codes (EDAC) mechanisms have been adopted to do so.

Being inherently transient is a feature of most of the bit-flip errors because they can be removed at the next write cycle and their occurrence probability should be independent from previous occurrences. Those fail-stop errors which are usually duo to hardware violation can't be recoverable by our mechanism. Although a permanent, unrecoverable error such as a disk read error can also happen to GPU, we focus on the fault tolerant capability to deal with fail-continue errors. Note that our mechanism can detect hard errors as well, but it doesn't help recover from hard errors [6].

Another important assumption of errors is that they are mostly single-bit errors [6]. DRAM is the main component of GPU memory architecture. Thus protection of DRAM is an important step toward the overall reliability of GPU computing. Not only DRAM needs protection, on-chip components, such as shared memory, registers, and arithmetic units, should also be protected. But, our work only deals with GPU DRAM transient errors which are reported as the most common reason of system failures in many HPC systems [12] and we assume error-free operations within GPU chips.

IV. ERROR DETECTION, LOCATION AND CORRECTION

If there is one erroneous element in a full checksum matrix, exactly one row and one column will have an incorrect checksum. The intersection of this row and column locates the erroneous matrix element [16]. A procedure used to detect, locate and correct a single erroneous element in a full checksum matrix is listed in the following.

A. Failure Detection and Location

Row and column checksums vectors are calculated for this N×N sub-matrix. If there are no errors, the checksum vectors of the extracted matrix should match the sum of elements on corresponding rows and columns. If they don't, errors are present. Specific index for the error element can be located by scanning all rows and columns to find the intersection of the inconsistent row and column. This scheme can be used to detect the erroneous elements occurring on no more than one column or one row of any checksum matrix and correct at least one error or multiple errors on one column intersected with rows or one row intersected with columns. For simplification, we assume there is only a single error each time of error is detected.

B. Single Failure Recovery

Here, we demonstrate the simple case in which there is only one error in the matrix C. The computation of product C can be done in several steps, which will be discussed later, and appropriate number of error checks is done between the steps. During error-check if we assume that data on the position (i,j) of matrix C satisfies:

$$\sum_{k=1}^{N} C_{i,k} = C_{i,N+1} \quad (1) \quad Or \quad \sum_{k=1}^{N} C_{k,j} = C_{N+1,j} \quad (2)$$

Where N is the original size of the square matrix A and B which are used for the computation of product C. $(N+1)_{th}$ row and column are the row and column checksums. Then the lost data can be recovered from (1) or (2). Assuming $C_{i,j}$ is detected as a junk value, it will be recovered from

$$C_{i,j} += C_{i,N+1} - \sum_{k=1}^{N} C_{i,k} \quad (3) \quad Or \quad C_{i,j} += C_{N+1,j} - \sum_{k=1}^{N} C_{k,j} \quad (4)$$

In each step to get partial product, although the values of all elements changed, but the relationship (1) and (2) will be maintained. If any element's value is error, the data lost can be reconstructed through solving this linear equation with one unknown.

C. Multiple Failure Recovery

The checksum relationship in the last sub-section can only reconstruct one erroneous element on matrix C. However, in GPU memory system, double-bit errors do occur when a single memory word is affected by two radiation events, or when a single event affects multiple adjacent bits. In this section, we will discuss a scheme to recover multiple simultaneous failures. There are two situations for multiple errors on a matrix.

If we assume there are m errors and all the errors occur on the same column/row, they can be recovered from their own column/row checksum relationships.

If the m errors are on random position of the matrix, and we assume that the maximum number of errors on the same row or column is k, in order to be able to reconstruct the lost data from the remaining elements, the checksum rows and columns are dedicated to hold at least k weighted checksums of all the elements on its row or column at the beginning of application. The weighted checksum relationships actually establish k equalities between the data on corresponding row or column on computation processes and the encoding data on the encoding processes. The process of getting the lost data back reduces to finding the solution of linear equations.

V. Fault-Tolerance Matrix Multiplication

In this paper, we implemented the technique by utilizing checksum matrices. We used the Column Checksum matrix ($A^{c\_checksum}$) and multiplied it with the Row Checksum matrix ($B^{r\_checksum}$). If A and B are both N×N matrices, the product matrix is a (N+1)-by-(N+1) matrix. The extracted N-by-N sub-matrix is exactly the result of computation of A×B.

For simplicity, we only discuss the case where there is only one element error on matrix C (C = C+A×B).

In this section, *b* denotes number of columns of A or rows of B used for computing partial product of C. *fq* denotes the frequency of error detection.

DEFINITION 5.1: The original matrix A, B and C are N-by-N matrix. The element on the i_{th} row and j_{th} column in matrix A, B or C can be written as $A_{i,j}$, $B_{i,j}$ or $C_{i,j}$.

A. Blocked outer product matrix multiplication algorithm

To achieve high performance, we use is a blocked outer product version of the matrix multiplication algorithm. We initialize matrices A and B with random number. Let A_j denote the j_{th} column block of the matrix A and B^T_j denote the j_{th} row block of the matrix B. The following Figure 1 is the algorithm to perform the matrix multiplication. Figure 2 shows the j_{th} step of the matrix multiplication algorithm.

for j = 0, 1…N/b
C = C + A_j ×B^T_j ;
end

Figure 1. Blockly Outer Product Matrix Multiplication Algorithm.

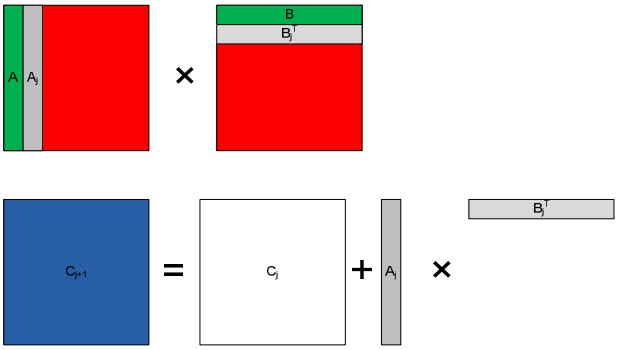

Figure 2. The jth step of the matrix multiplication algorithm.

B. Checksum relationship maintained during computation

THEOREM 1: The result of a column checksum matrix $A^{c\_checksum}$ multiplied by a row checksum matrix $B^{r\_checksum}$ is a full checksum matrix $C^{checksum}$. Shown as

$$C^{checksum} = A^{c\_checksum} \times B^{r\_checksum} = AB^{checksum}$$

THEOREM 2: Checksum relationship is maintained in the result of a sub-matrix of $A^{c\_checksum}$ multiplied by a sub-matrix of $B^{r\_checksum}$. A^b is a block set of columns of Matrix A. B^b is a block set of rows of Matrix B. C^b is the product of $A^b \times B^b$.

Proof:

$$A^b = \begin{pmatrix} A^b \\ v^T A^b \end{pmatrix}, B^b = \begin{pmatrix} B^b & B^b\omega \end{pmatrix}$$

$$\text{and } C^b = \begin{pmatrix} C^b & C^b\omega \\ \vartheta^T C^b & \vartheta^T C^b \omega \end{pmatrix}$$

Here, both ϑ^T and ω are checksum vectors.

$$C^b = \begin{pmatrix} C^b & C^b\omega \\ \vartheta^T C^b & \vartheta^T C^b \omega \end{pmatrix} = \begin{pmatrix} A^b \\ v^T A^b \end{pmatrix} \begin{pmatrix} B^b & B^b\omega \end{pmatrix}$$

$$= \begin{pmatrix} A^b B^b & A^b B^b \omega \\ \vartheta^T A^b B^b & \vartheta^T A^b B^b \omega \end{pmatrix}$$

C. Fault tolerance for single error in matrix multiplication

Here we explain how the algorithm-based fault tolerance works for matrix multiplication in order to handle single error in a full checksum matrix.

Following is the Algorithm of matrix multiplication with fault tolerance on matrix C

```
build checksum matrix from A, B and C
for j = 0, 1…N/b
  if (mod(j,fq)==0)
    comparison between all sum of elements in the &
    row/column and all row/column checksum of C
    if doesn't match
      locate the error
      recover lost date in C^checksum
    end if
  end if
  C^checksum = C^checksum + A_j^c_checksum ×(B_j^r_checksum)^T ;
end
```

Figure 3. Blockly Matrix Multiplication Algorithm with fault tolerance using checksum relationship (tolerate matrix C only).

For the case that there is only one erroneous element in the full checksum matrix $C^{checksum}$, exactly one row and one column will have incorrect checksum. The intersection of this row and column locates the erroneous matrix element. The procedure used to detect, locate and correct a single erroneous element in a full checksum matrix $C^{checksum}$ is listed below:

Step 1: Compute the sum of elements same as that in the original matrix C for each column and row in matrix $C^{checksum}$. Step 2: Compare the sum with the checksum (because of round-off errors, a small tolerance should be allowed in comparison). Step 3: The intersection of row and column which doesn't match the condition is the index of erroneous element. Step 4: If error occurs in original elements, it can be corrected by adding the difference of the newly calculated sum and updated checksum. If the erroneous element occurs at the encoded data, it should be replaced by the newly calculated summation.

In order to tolerate some error in matrix A and B as well, and not just in C, we can build full checksum matrices for both A and B. Detecting, locating and correcting error is done as discussed in 5.3.

D. Overhead Analysis

We analyze the overhead introduced by the checksum fault tolerance for matrix multiplication in this section. For simplicity of presentation, we assume that all three matrices A, B, C are N-by-N square matrices. Let γ denote the time the CPU takes to perform one floating-point arithmetic operation.

1) Overhead for Encoding and Detecting Errors

To make matrix multiplication fault tolerant, the first type of overhead introduced is (1) constructing the column checksum matrix $A^{c\_checksum}$ from A; (2) constructing the row checksum matrix $B^{r\_checksum}$ from B.

The time complexity of the checksum operation for one matrix can be expressed as follow.

$$T_{encode} = N^2\gamma \quad (5)$$

[19] shows the overhead to construct a full checksum matrix.

$$T_{encode\_fullchecksum} = 2N^2\gamma \quad (6)$$

The procedure to detect errors in encoded matrices is a process to check whether the checksum is still equal to the sum of elements in corresponding row or column. The process to scan a whole (N+1)-by-(N+1) matrix with full checksum once needs $2N^2$ addition operations and 2N branch operations. If the program can tolerate m errors, then the overhead to detect a full checksum matrix is:

$$T_{detect} = 2mN^2\gamma \quad (7)$$

The procedure of detecting errors in matrices A and B from checksum matrices is similar to that in matrix C if full checksum is constructed for both A and B.

2) Overhead for Recovery

Matrices A, B and C can be recovered from either the row checksum or the column checksum relationship. The overhead recovering data depends on how many errors the fault tolerant matrix multiplication can handle and the size of target matrix as well. Assuming it handles *m* errors at one time error detection can be before running the program, the time complexity is

$$T_{recovery} = mN\gamma \quad (8)$$

3) Overhead for communication

Another overhead for computing on GPU is copying data from CPU to device and back. Assuming 1/β (GB/s) is the bandwidth of the communication between CPU and GPU. 32 bit = 3.7252903×10^{-9} GB.

$$T_{comm} = 2 \times 3.72\beta \quad (9)$$

VI. PERFORMANCE

In this section, we experimentally test the performance of our fault tolerance technique applied to matrix multiplication on GPU. We performed four sets of tests to show the advantage of our approach.

- Performance of our approach with different error detection ability.
- Overhead of our approach compared to CUBLAS matrix multiplication without fault tolerance and overhead of our approach with different error detection ability.
- Comparison between our approach and TMR.
- Comparison between our approach and traditional ABFT.

All the tests are done on the machine provided by Colorado School of Mines managed by GECO. The GPU device is NVIDIA Tesla S1070 system. This 1U rack mountable system contains 4 of the NVIDIA Quadro FX 5600 GPU cards and has a peak computing performance of 4 trillion floating point operations per second, or 4 Teraflops. Each of the 4 graphics processing units (GPU) on the Tesla has 240 processing cores and 4 Gbytes of memory for a total of 960 cores and 16 Gbytes. The individual GPUs are

connected to the front-end node of Mio (mio.mines.edu) via a PCI connector.

A routine called cublasSGEMM in CUBALS 3.1 is used to implement the online fault-tolerant matrix multiplication.

Results are shown in following tables and figures.

A. *Performance of our technique tolerating different number of errors*

The first set of data includes the execution time of the program tolerating different errors using our technique. Figure 5 shows the performance of our approach tolerating 2 errors, 4 errors, 8 errors and 16 errors respectively. From the figure, we can see that for a fixed number of errors which the program can tolerate, as the size of the matrix increases, the execution time increases exponentially. This is because time complexity of matrix multiplication is $O(N^3)$. A fault tolerance matrix multiplication's time complexity must include the overhead to encode, detect and recover errors. It's estimated to be $O(mN^2)$. Where m is the number of error the program can tolerate. So the total execution time of fault tolerance matrix multiplication is dominated by the operation to computer the product.

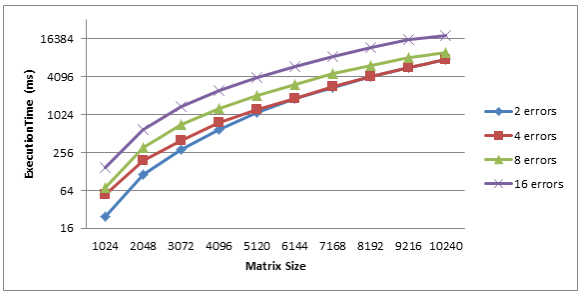

Figure 4. Performance of our technique tolerating different number of errors

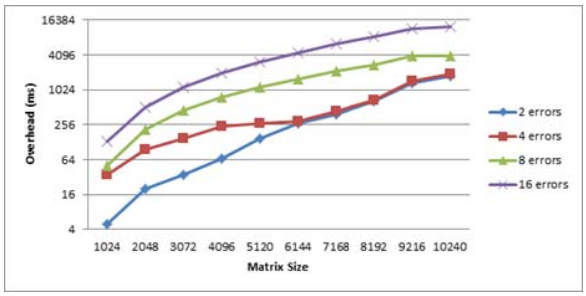

Figure 5. Overhead of our technique tolerating different number of errors

TABLE I. OVERHEAD OF OUR FAULT-TOLERANT MATRIX MULTIPLICATION MECHANISM

MATRIX_SIZE	1024	2048	3072	4096	5120	6144	7168	8192	9216	10240
Our technique [only C] (ms)	25	115	285	595	1095	1820	2770	4060	5650	7590
Original CUBLAS (ms)	20	95	250	530	950	1550	2380	3420	4300	5820
Overhead (ms)	5	20	35	65	145	270	390	640	1350	1770
Overhead (%)	25.00%	21.05%	14.00%	12.26%	15.26%	17.42%	16.39%	18.71%	31.40%	30.41%

TABLE II. PERFORMANCE COMPARISON BETWEEN OUR APPORACH AND TMR

MATRIX_SIZE	1024	2048	3072	4096	5120	6144	7168	8192	9216	10240
Our technique [only C] (ms)	25	115	285	595	1095	1820	2770	4060	5650	7590
TMR(ms)	60	280	730	1560	2830	4610	7020	10220	12550	17400
Speed Up	2.40	2.43	2.56	2.62	2.58	2.53	2.53	2.52	2.22	2.29

B. *Overhead of our fault-tolerant matrix multiplication mechanism*

This set of data shows overhead of our fault tolerant matrix multiplication mechanism compared with CUBALS version without fault tolerance. Also, comparison between overhead of our technique tolerating different number of errors is shown in a figure.

Table I clearly reports the overhead of our technique. The execution time of the code with fault tolerance is based on the program tolerating two errors. From the section 5.4, we demonstrate that overhead for our technique mainly consists of three parts: Encoding information, detecting errors and recovering corrupted data. Based on formula 5-9, the estimated total overhead is $2\gamma(mN^2+N^2+N)$. So, basically, the time complexity can be expressed as $O(mN^2)$. The overall overhead introduced increases quadratically as the size of the matrix increases. Since the time taken by the GPU to perform one floating-point arithmetic operation is different from that of the CPU and also the total execution time by the GPU depends on how the algorithm is parallelized and whether it fully uses the cache and shared memory on GPU, the ratio of overhead to GPU matrix multiplication without fault tolerance can't be derived directly. The ratio is derived from the experiments and is kept in the range from 12.26% to 31.4%. The Average overhead is roughly 20%.

Figure 6 shows the overhead of programs tolerating different number of errors. As the number of errors increases by two times more, the overhead doubles, which is consistent with formula $O(mN^2)$ theoretically about the overhead for our technique.

C. *Performance comparison between our approach and TMR*

The third set of data in table II indicates performance comparison between our fault-tolerant mechanism and TMR which is another important method that could tolerate continued failures. The real TMR is a fault-tolerant mechanism in which three systems perform a process and the result is processed by a voting system to produce a single output. In our emulated TMR, we run the same program three times to tolerate faults during computation which results in incorrect data instead of fail of the device.

Table II clearly demonstrates the speedup our approach gains. It's roughly 2.5 times speedup. No matter it is hardware based TMR or software based emulated TMR, the cost is obviously more.

D. *Performance comparison between our approach and traditional ABFT*

The last set of data indicates the performance comparison between our fault-tolerant mechanism and traditional ABFT which is a very famous technique to check the correctness of most matrix operation and recover the corrupted data.

In table III, data is collected in the situation that both programs tolerating error by our approach and traditional ABFT has only one column/row checksum on matrix A and B. Two errors occur in random position of matrix C during computation.

To tolerate two errors in our approach, only one checksum vector is needed for both matrices A and B and the program needs to be executed only once. However, this encoded information is not sufficient for traditional ABFT to tolerate two errors. Traditional ABFT can only detect one error at the end of computation by given the same coded information. If two errors occur, traditional ABFT has to rerun the program in order to get the correct result under the assumption that there is no more than one corrupted data in the second run. So the execution time of traditional ABFT is about double that of the execution time of our approach.

Our technique is as generally applicable as ABFT which presents the checksum relationship maintain at the end of most matrix operation. More corrupted data could be detected and recovered during the execution of the code instead of checking errors after the matrix operation like traditional ABFT and the mechanism described in the [20] which can handle single corruption in the end of computation or single corruption during execution on the assumption that the error doesn't propagate. However, our technique can handle more errors than the other two methods with the same amount of coded information, because the algorithm for matrix multiplication in this paper is outer product version where matrix A/B are divided into a number of blocks of columns/rows, and matrix C is updated with one (or more) products of a panel of columns of A times a panel of rows of B.

The advantages of outer product matrix multiplication used in our technique are that any corrupted data in partial product matrix can be detected then corrected in the intermediate steps, which means it tolerates more errors with the same amount of coded data compared to traditional ABFT and it reduces the possibility error propagation by check relationship between the checksum and sum of corresponding row or column in the middle of execution. However, in traditional ABFT, which is based on inner product multiplication, corrupted data in matrices A and B propagates easily and generates more wrong results in matrix C. Although this problem can be solved by more encoded data, the overhead would be larger. Also, in traditional ABFT, the sub-matrix of C where calculation has been done would only be checked at the end of computation. If several errors occur in those parts after it's done and before final product matrix C is generated, the original encoded data are not sufficient to recover errors.

TABLE III. PERFORMANCE COMPARISON BETWEEN OUR APPORACH AND TRADITIONAL ABFT

MATRIX_SIZE	1024	2048	3072	4096	5120	6144	7168	8192	9216	10240
Our technique [only C] (ms)	25	115	285	595	1095	1820	2770	4060	5650	7590
Traditional ABFT (ms)	45	195	515	1090	1978	3249	4969	7145	9096	12337
Speed Up	1.80	1.70	1.81	1.83	1.81	1.79	1.79	1.76	1.61	1.63

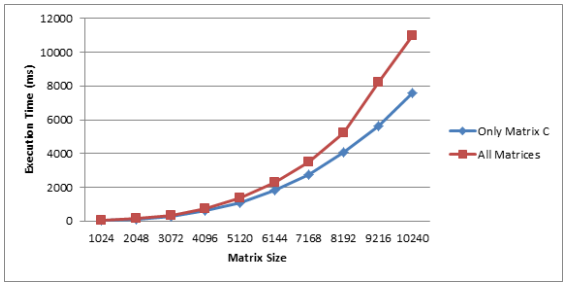

Figure 6. Performance of fault tolerance on only c and all matrces

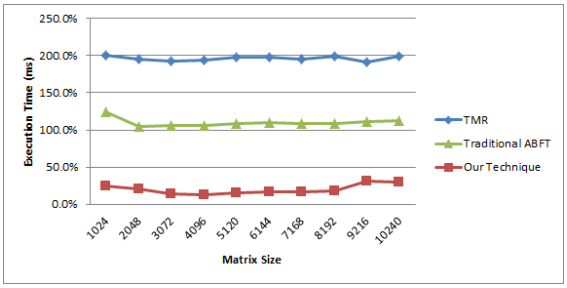

Figure 7. Overhead Comparison between our technique and other two technique (TMR and Traditioanl ABFT)

VII. CONCLUSION

In this paper, we extended the traditional algorithm-based fault tolerance (ABFT) from offline to online and applied it to matrix multiplication on GPUs. The proposed method is able to detect, locate, and correct soft error in matrix multiplication on GPUs in the middle of the computations. Experimental results demonstrate that the proposed method is highly efficient.

ACKNOWLEDGMENT

This research is partly supported by National Science Foundation, under grant #OCI-0905019 and Department of Energy, under grant DE FE#0000988.

We want to thank HPC group in Colorado School of Mines-Golden Energy Computing Organization (GECO) for the GPU node Cuda1 (cuda1.mines.edu) and computer cluster nodes Mio (Mio.mines.edu).

REFERENCES

[1] Haque, Imran S.; Pande, Vijay S.; , "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on* , vol., no., pp.691-696, 17-20 May 2010

[2] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with CUDA," in GPU Gems 3, H. Nguyen, Ed. Addison Wesley Professional, August 2007, ch. 31.

[3] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, no. 16, pp. 2618–2640, September 2007.

[4] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 1, pp. 474+, 2007.

[5] S. S. Stone, J. P. Haldar, S. C. Tsao, W. Mei, Z. P. Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs," in *Proceedings of the 5th conference on Computing frontiers*, 2008, pp. 261–272.

[6] Maruyama, N.; Nukada, A.; Matsuoka, S.; , "A high-performance fault-tolerant software framework for memory on commodity GPUs," *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* , vol., no., pp.1-12, 19-23 April 2010

[7] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-ofthe- art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, 1984.

[8] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q Supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, 2005.

[9] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005, pp. 243–247.

[10] Performance Cost Analysis of Software-Implemented Hardware Fault Tolerance Methods in General- Purpose GPU Computing. [Online]. http://homepages.cae.wisc.edu/~ece753/papers/Paper_4.pdf

[11] Zizhong Chen; Dongarra, J.; , "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources, " *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* , vol., no., pp.10 pp., 25-29 April 2006

[12] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance-Computing Systems," in *International Conference on Dependable Systems and Networks (DSN'06)*, June 2006, pp. 249¯C258.

[13] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *MICRO-32*, 1999, pp. 196–207.

[14] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: software implemented fault tolerance," in *Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

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

[16] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," IEEE Transactions on Computers, vol. C-33, no. 6, pp. 518–528, June 1984.

[17] B. Schroeder, E. Pinheiro, and W. D. Weber, "DRAM errors in the wild: a large-scale field study," in SIGMETRICS '09, 2009, pp. 193–204.

[18] C. N. Hadjicostis and G. C. Verghese. Coding approaches to fault tolerance in linear dynamic systems. *IEEE Transactions on Information Theory*, 2005

[19] P P. Ferreira and J. Vieira. Stable dft codes and frames. IEEE Signal Processing Letters, 10(2):50{53, 2003.C. N. Hadjicostis and G. C. Verghese. *Coding approaches to fault tolerance in linear dynamic systems. Submitted to IEEE Transactions on Information Theory*, 2004

[20] Gunnels, J.A.; Katz, D.S.; Quintana-Orti, E.S.; Van de Gejin, R.A.; , "Fault-tolerant high-performance matrix multiplication: theory and practice," *Dependable Systems and Networks, 2001. DSN 2001. International Conference on* , vol., no., pp.47-56, 1-4 July 2001

[21] Volkov, V., and Demmel, J. W. 2008. Using GPUs to accelerate linear algebra routines, *Poster at PAR Lab Winter Retreat*, January 9, 2008.

[22] Barrachina, S.; Castillo, M.; Igual, F.D.; Mayo, R.; Quintana-Orti, E.S.; , "Evaluation and tuning of the Level 3 CUBLAS for graphics processors," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* , vol., no., pp.1-8, 14-18 April 2008