CPU-GPU Hybrid Bidiagonal Reduction With Soft Error Resilience

Yulu Jia[*], Piotr Luszczek[*], George Bosilca[*], Jack J. Dongarra[*†§]

[*]University of Tennessee Knoxville
[†]Oak Ridge National Laboratory
[§]University of Manchester

ABSTRACT

Soft errors pose a real challenge to applications running on modern hardware as the feature size becomes smaller and the integration density increases for both the modern processors and the memory chips. Soft errors manifest themselves as bit-flips that alter the user value, and numerical software is a category of software that is sensitive to such data changes. In this paper, we present a design of a bidiagonal reduction algorithm that is resilient to soft errors, and we also describe its implementation on hybrid CPU-GPU architectures. Our fault-tolerant algorithm employs Algorithm Based Fault Tolerance, combined with reverse computation, to detect, locate, and correct soft errors. The tests were performed on a Sandy Bridge CPU coupled with an NVIDIA Kepler GPU. The included experiments show that our resilient bidiagonal reduction algorithm adds very little overhead compared to the error-prone code. At matrix size 10110×10110, our algorithm only has a performance overhead of 1.085% when one error occurs, and 0.354% when no errors occur.

Keywords

Bidiagonalization, Soft error, ABFT, Reverse computation, Resilient, GPU, Hybrid

1. INTRODUCTION

Bidiagonalization of a general $M \times N$ matrix A is prerequisite to computing the singular value decomposition (SVD) of A. The execution time of numerical bidiagonalization on modern computers dominates the computation of SVD. Given an $M \times N$ real matrix A, the SVD decomposition computes $A = U\Sigma V^{\top}$ where U is an $M \times M$ orthogonal matrix, Σ is an $M \times N$ diagonal matrix, and $V^{\top}$ is an $N \times N$ orthogonal matrix. The diagonal entries of matrix Σ are called the singular values of A. The numerical SVD decomposition of a matrix is usually performed in two steps. In the first step, matrix A is reduced to bidiagonal form: $A = QBP^{\top}$ where Q is an $M \times M$ orthogonal matrix, P is an $N \times N$ orthogonal matrix, and B is an $M \times N$ bidiagonal matrix. In the second step, matrix B is further reduced to diagonal form. The implementations of the steps are slow because they contain a lot of matrix-vector multiplies (GEMV), which are Level 1 BLAS operations and have a low computation intensity. As a result of the above reasons, it is time consuming to calculate either of the two stages.

ScalA '13, November 17-21, 2013, Denver, CO, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2508-0/13/11 ...$15.00
http://dx.doi.org/10.1145/2530268.2530270

Advances in Integrated Circuits (IC) manufacturing technology, described below, bring forward higher probability of soft errors in computer systems due to decreased feature size and increased complexity. A soft error is a temporary malfunction of a chip element which causes a change in the program state without any notification other than an incorrect result, but the chip element continues to function normally, and the change in program state is unnoticed by either the hardware or the software. Moore's Law states that the number of transistors on integrated circuits doubles every two years [12]. This increased transistor density on a unit silicon area provides every more prominent possibility for soft error. Also, higher transistor density requires smaller transistor feature size (the minimum size of a transistor or a wire on an IC), and causes increased heat dissipation as the circuit consumes higher power. Smaller transistors require lower voltage to operate at increasing frequencies, which makes it easier to change the transistor state unpredictably. High heat dissipation generates more thermal neutrons, which in turn cause more soft errors in the chip [20].

The computation needs to be protected so that there is no need to repeat the computation in the presence of soft errors. There are a few challenges in tolerating soft errors. First, it is difficult to detect them since a soft error changes the application state without the hardware or software noticing it. There is no permanent physical damage to the hardware, so the program proceeds normally in the presence of a soft error (assuming that the soft error does not alter the control logic). Second, it is difficult to pinpoint the error even when given the knowledge of the existence of an error. There are a large number of transistors involved in a single computation, so locating the error is analogous to finding a needle in a haystack. Third, suppose we know an error exists, and we know the exact location of the error, it is still difficult to restore the data to the correct value.

In this paper, we propose an effective and efficient algorithm to detect, locate, and correct soft errors in the numerical bidiagonal reduction of a real matrix. We employ the Algorithm Based Fault Tolerance (ABFT) technique and reverse computation to achieve fault tolerance. Our fault tolerant bidiagonal reduction algorithm is very efficient in that it introduces very low performance overhead compared with the non-fault tolerant counterpart. The overhead tends asymptotically to 0 as the matrix size scales up. We show the effectiveness and efficiency of our algorithm through an implementation based on the MAGMA library [18, 19]. Experiments show that our algorithm has a low overhead of 0.354% at matrix size about 10000. The overhead exhibits a decreasing trend as the matrix size increases.

The rest of this paper is organized as follows: Section 2 reviews related work. Section 3 introduces the block bidiagonal reduction algorithm as implemented in the MAGMA project. Section 4 discusses the soft error propagation pattern. Section 5 explains our fault tolerant algorithm. Section 6 reports the experimental results. Finally, section 7 presents our conclusions.

2. RELATED WORK

Research and reports about the existence and impact of soft errors on GPUs show that soft errors are a real problem for scientific applications [7,13,15,16]. There are efforts to tolerate these errors using both software-based approaches [3,10,11] and hardware-based approaches [14,17].

Du et al. [5] proposed a soft error resilient QR factorization algorithm using a post processing approach. In their scheme, the input matrix is encoded with two extra checksum columns. These two extra columns are maintained as the regular QR factorization proceeds. After the factorization has finished, the two extra columns are used to detect the existence of a soft error and locate the column index where the error occurred. The error is then projected to a rank-1 perturbation of the original input matrix. Then the correct factorization result is obtained using the QR update technique [6]. This post-processing scheme can successfully tolerate, at most, one soft error, no matter what point in time the error has occurred.

Kim et al. [9] designed a general scheme for fault tolerant matrix operations including matrix multiplication, Cholesky factorization, LU factorization, QR factorization, and Hessenberg reduction. This scheme tackles hard errors (process failures). The method uses a checksum to encode the input matrix, and the checksum is generated at certain intervals and serves as a checkpoint of the application state. In the case of a hard error, a roll back is performed to bring the program data back to the state at which the last checksum was generated.

The MAGMA project [19] is an effort to take advantage of the latest development of GPU accelerators to boost the performance of linear algebra operations. The project redesigned the block algorithms in LAPACK [1] to better suit GPU-enabled hybrid platforms. The methodology is called hybridization, by which computational tasks are split according to their characteristics and scheduled to the CPUs and GPUs accordingly. The `magma_dgebrd` routine in MAGMA implements the hybrid block bidiagonal reduction algorithm.

3. BLOCK BIDIAGONAL REDUCTION IN MAGMA

In order to understand our bidiagonal reduction algorithm with fault tolerant features, it is essential to understand the non-fault tolerant algorithm first. In this section, we describe the standard block bidiagonal reduction algorithm.

Suppose A is an $M \times N$ matrix, the block algorithm logically partitions A into $M \times nb$ block columns and $nb \times N$ block rows. The reduction is an iterative process, whereby every iteration reduces the leftmost nb matrix columns and the uppermost nb matrix rows into the bidiagonal form. In every iteration, the following operation is performed on the unreduced trailing matrix [2,4]:

$$A^{(i+1)} = A^{(i)} - VY^{\top} - XU^{\top}$$

where $A^{(i)}$ is the unreduced trailing matrix at the beginning of the i-th iteration, and $A^{(i+1)}$ is the resulting matrix of the i-th iteration (also the input for the $(i+1)$-th iteration). V is an $M \times nb$ matrix which consists of the Householder vectors to annihilate columns of A; this matrix is used to update the trailing matrix from the left. U is the matrix used to transform the input matrix from the right; this matrix contains the Householder vectors used to annihilate rows of A. Y and X are intermediate matrices, and each of them is generated through an iterative process. Denoting the k-th column of Y as y_k, the k-th column of X as x_k, the calculation of Y and X are given by [2,4]:

$$\begin{aligned} y_{k+1} &= \tau_{v_{k+1}}(A^{(i)^{\top}} v_{k+1} - Y_k V_k^{\top} v_{k+1} - U_k X_k^{\top} v_{k+1}) \\ x_{k+1} &= \tau_{u_{k+1}}(A^{(i)} u_{k+1} - X_k U_k^{\top} u_{k+1} - V_{k+1} Y_{k+1}^{\top} u_{k+1}) \end{aligned}$$

In MAGMA, the bidiagonal reduction routine for a double precision real matrix is `magma_dgehrd`. In every iteration, the algorithm performs the following operations:

1. Call `magma_dlahrd_gpu`. This call reduces the i-th block column and the i-th block row of A to bidiagonal form, and generates V, U, X, and Y. The two largest tasks in this routine (two GEMVs) are offloaded to the GPU.

2. Call `magma_dgemm` to compute $A = A - VY^{\top}$. This matrix-matrix multiply and the following one are offloaded to the GPU.

3. Call `magma_dgemm` to compute $A = A - XU^{\top}$

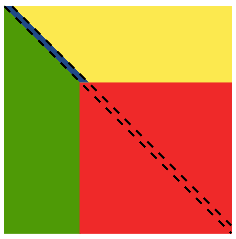
Figure 1 shows one iteration of the `magma_dgehrd` routine.

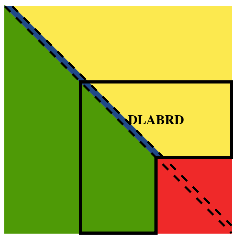
4. ERROR PROPAGATION

In this work we target soft errors specifically (as opposed to hard errors). A single bit flip is sufficient to completely invalidate the factorization result. Figure 2 shows the impact of a soft error in the course of the factorization. This example uses a 158×158 matrix with the block size $nb = 32$. The soft error occurs in the second iteration at location $(72, 79)$, which is marked by a cross in Figure 2(a). Figure 2(b) shows the heat map of the difference matrix between the correct factorization result and the factorization result affected by one soft error. Black color indicates that the difference is 0, and any other color indicates a difference of a magnitude proportionate to the color. We can observe that the rectangular matrix at the bottom right corner contains wrong values.

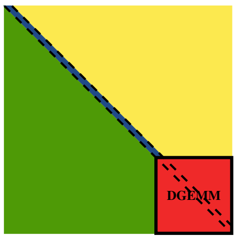
5. FAULT TOLERANT DGEBRD

In this section, we describe the fault tolerant bidiagonal reduction algorithm. The algorithm is inspired by the ABFT concept [8]. The basic idea is to add redundant information to the original data. A soft error means that some information in the original matrix is corrupted. After the detection of the soft error, the algorithm uses the redundancy to recover the corrupted information. The redundant information of the input matrix is provided by a checksum column and a checksum row. Algorithm 1 shows the details of our approach.

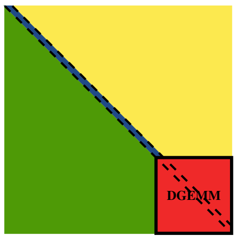

(a) Beginning of iteration



(b) Factorize the panel



(c) $A = A - VY^\top$



(d) $A = A - XU^\top$

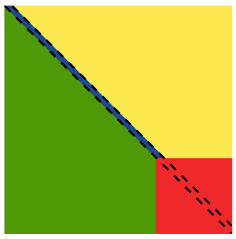
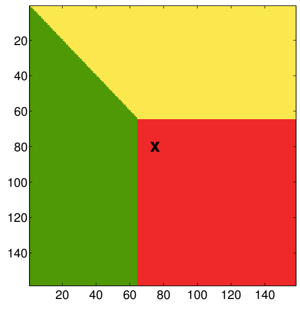
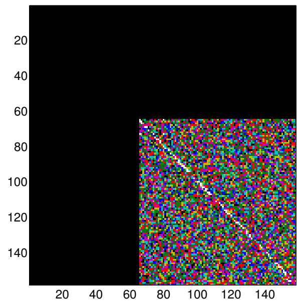

(e) End of iteration

Figure 1: One iteration of `magma_dgebrd`



(a) Error location $(72, 79)$.



(b) Partition

Figure 2: Propagation pattern of an error.

Algorithm 1 Fault Tolerant Hybrid Bidiagonal Reduction

```
1:  Transfer matrix: $A$ on the host → d_$A$ on the GPU
2:  Encode the input matrix, expand it with a checksum column
    and a checksum row.
3:  **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
4:      Transfer the leftmost $nb$ columns and uppermost $nb$ rows of
        the trailing matrix to the host.
5:      **FT_MAGMA_DLABRD_GPU**, return $V, U, X$ and $Y$
6:      Compute $X_{ce}, Y_{ce}, V_{ce}, U_{ce}$
7:      **DGEMM**: $A_{fe} = A_{fe} - V_{ce}Y_{ce}^\top$
8:      **DGEMM**: $A_{fe} = A_{fe} - X_{ce}U_{ce}^\top$
9:      Compute $S_{re} = \sum A_{re}(i)$ and $S_{ce} = \sum A_{ce}(i)$
10:     **if** $|S_{re} - S_{ce}| >$ *threshold* **then**
11:         Reverse the last left update and right update.
              $A_{fe} = A_{fe} + V_{ce}Y_{ce}^\top$
              $A_{fe} = A_{fe} + X_{ce}U_{ce}^\top$
12:         Correct the error
13:     **end if**
14: **end for**
```

Algorithm 2 Locate(i, j, k)

```
1: **DGEMV**: $\hat{A}_{chk\_r} = A_{trail} \cdot e$
2: IF
3: **DGEMV**: $\hat{A}_{chk\_r} = A_{trail} \cdot e^\top$
4: IF
```

5.1 Data Redundancy

Algorithm 3 Recover(i, j, k)

```
1: Set $A(i, j)$ to 0.
2: Compute $S_{new} = \sum_{i=k+1}^{M} A(i, j)$
3: $\hat{A}(i, j) = A_{ce}(j) - S_{new}$
```

The algorithm first encodes the input matrix with both row checksums and column checksums. The row checksums form a column vector which is appended to the right of the matrix, the column checksums form a row vector which is appended to the bottom of the input matrix. This task is accomplished in line 2. The algorithm enters the main loop in line 3. In every iteration, the next panel is transferred to the CPU to be factorized there (in Line 5). The original `magma_dlabrd_gpu` routine only computes part of X and part of Y. Assuming the trailing matrix is of size $m \times n$, we only need the lower $n \times nb$ part of Y and the lower $m \times nb$ part of X are needed to update the trailing matrix. In our fault tolerant algorithm, we need the entire X and Y to calculate their respective column checksums, so we modified the `magma_dlabrd_gpu` routine to compute the complete X and Y. The new routine is named `ft_magma_dlabrd_gpu`. Line 7 and line 8 update the trailing matrix. The row checksums and column checksums are also updated together with the trailing matrix. After the update, the row checksums remain to be the row checksums of their corresponding rows. The column checksums remain to be the column checksums of their corresponding columns. In other words, the checksum relationship is preserved throughout the algorithm.

5.2 Error Detection

Line 9 and line 10 carry out error detection. Error detection is achieved by comparing the sum of the row checksums of the trailing matrix and the sum of the column checksums of the trailing matrix. Because both checksum vectors protect the same matrix data (the trailing matrix), their sums should be equal to each other. If the difference is higher than a certain threshold, we consider an error has occurred. The comparison of the two sums is performed in line 10.

5.3 Error Location and Correction

If an error is detected at line 10, the algorithm initiates the procedure to locate and correct the error. To achieve this, the algorithm first performs a reverse update on the trailing matrix. This is accomplished in line 11. The reverse update brings the trailing matrix back to the state at the beginning of the erroneous iteration. At this point, the error only exists in one matrix entry, the contamination to other matrix entries is reversed, and now we have the correct row checksums and column checksums. The error location works as follows. We compute the new row checksums and column checksums of the actual trailing matrix, and these new checksums will encode the erroneous value. Moreover, there will be exactly one row checksum which differs from its corresponding old row checksum, and there will be exactly one column checksum which differs from its corresponding column checksum. The row index and column index of the error can be identified by comparing the new checksums and the old checksums.

Once the error location (i, j) has been determined, we can use the row checksums and the column checksums as devices to recover the lost matrix element. First we set $A(i, j)$ to zero, then we compute the checksum chk_r for the i-th row. The lost matrix element can be recovered by $A(i, j) = chk_r - old\_chk_r$. old_chk_r is the row checksum of the i-th row which the algorithm maintains since the beginning of the factorization. The algorithm resumes its normal operations after the recovery. It continues to detect, locate, and correct errors in subsequent iteration until the factorization completes.

5.4 Multiple Concurrent Errors

In previous subsections, we only considered the case in which only one soft error happens in an iteration. In fact the fault tolerant algorithm can deal with more than one soft errors in one iteration. When more than one soft error occurs, the entire trailing matrix will be contaminated as in the one-error case, so the existence of errors can always be detected. Similar to the analysis by Huang et al. [8], when the faulty elements form a rectangle, these four errors cannot be located. Other than such a situation, multiple errors can be located and then corrected.

5.5 Range of Application

The fault tolerant algorithm stated above is for bit-flips in the data matrix. The fault tolerant algorithm does not deal with soft errors in the control logic. It does not consider persistent errors either. Persistent errors are usually caused by malfunctioning hardware, this type of errors are outside of the range range covered by this work.

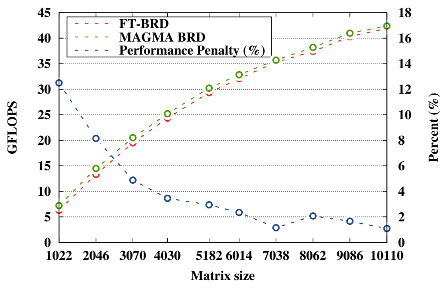
6. EXPERIMENT RESULTS

In this section, we present performance results of our fault tolerant bidiagonal reduction algorithm.

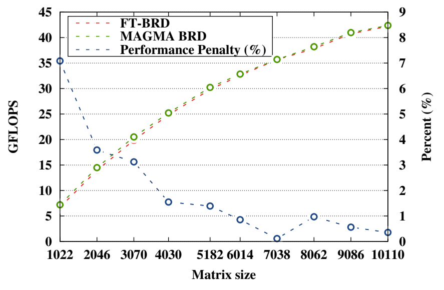
The test platform we use is a machine at the University of Tennessee. This machine has an Intel Xeon E5-2670 processor with a clock frequency of 2.6 GHz. It features an NVIDIA Tesla K20c GPU (known also as Kepler) with the clock frequency of the GPU at 705.5 MHz and the on-board memory of the GPU of 4799.6 MB. Te machine has 62 GB of main memory. The operating system is Red Hat 4.4.6-4 and the compiler is GCC 4.4.6 and NVCC 5.0 V0.2.1221.

Figure 3 shows the comparison of the performance of the fault tolerant bidiagonal reduction and the performance of the MAGMA bidiagonal reduction. Figure 3(a) shows the performance comparison when the fault tolerant bidiagonal reduction suffers from one soft error. The error is injected in the third iteration in the panel area. This is nearly the worst case scenario. The earlier the error occurs, the higher the cost of locating and correcting the error. The reason is that if the error occurs in the early iterations of the factorization, we need to reverse the update of the trailing matrix once we detect an error, and the trailing matrix is large in early iterations. To locate the error, we need two DGEMV operations on the trailing matrix. In early iterations the large trailing matrix also incurs higher costs in these two DGEMVs.

Figure 3(b) shows the performance comparison when the FT bidiagonal reduction does not experience any errors. We can see that the performance overhead also drops when the matrix size increases.



(a) With error



(b) Without error

Figure 3: Performance of the FT-BRD

7. CONCLUSION

In this paper, we showed a design, implementation, and a performance evaluation of a hybrid bidiagonal reduction algorithm based on the MAGMA framework equipped with fault tolerant features. Our fault tolerant bidiagonal reduction algorithm employs reverse computation and algorithm-based fault tolerance to detect, locate, and correct soft errors in the bidiagonal reduction on CPU-GPU hybrid architectures. Experimental results show that the performance overhead of our fault tolerant algorithm is very low when the matrix size is small, and the performance overhead as fraction of the overall computation time continues to drop as the matrix size increases. At matrix sizes of about 10000, the overhead decreases to

1.085% when one soft error occurs, and to 0.354% when no errors occur.

Acknowledgements

This research was sponsored in part by grants from DOE and grants from NSF.

8. REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, Third edition, 1999.

[2] J. Choi, J. J. Dongarra, and D. W. Walker. The design of a parallel dense linear algebra software library: Reduction to hessenberg, tridiagonal, and bidiagonal form. *Numerical Algorithms*, 10(2):379–399, 1995.

[3] M. Dimitrov, M. Mantor, and H. Zhou. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 94–104, New York, NY, USA, 2009. ACM.

[4] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215–227, 1989.

[5] P. Du, P. Luszczek, S. Tomov, and J. Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. *Journal of Computational Science*, 2013.

[6] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The John Hopkins University Press, 4th edition, December 27 2012. ISBN-10: 1421407949, ISBN-13: 978-1421407944.

[7] I. Haque and V. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 691–696, 2010.

[8] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[9] Y. Kim, J. S. Plank, and J. Dongarra. Fault Tolerant Matrix Operations Using Checksum and Reverse Computation. In *6th Symposium on the Frontiers of Massively Parallel Computation*, pages 70–77, Annapolis, MD, October 1996.

[10] N. Maruyama, A. Nukada, and S. Matsuoka. Software-based ECC for GPUs. In *SAAHPC'09*, July 2009.

[11] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity gpus. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[12] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[13] P. Rech, C. Aguiar, R. Ferreira, M. Silvestri, A. Griffoni, C. Frost, and L. Carro. Neutron-induced soft errors in graphic processing units. In *Radiation Effects Data Workshop (REDW), 2012 IEEE*, pages 1–6, 2012.

[14] J. W. Sheaffer, D. P. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 55–64, 2007.

[15] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. On testing gpu memory for hard and soft errors. In *Proc. Symposium on Application Accelerators in High-Performance Computing*, 2009.

[16] J. Tan, N. Goswami, T. Li, and X. Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 226–235, Washington, DC, USA, 2011. IEEE Computer Society.

[17] J. Tan, Z. Li, and X. Fu. Cost-effective soft-error protection for sram-based structures in gpgpus. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 29. ACM, 2013.

[18] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[19] S. Tomov, R. Nath, P. Du, and J. Dongarra. *MAGMA Users' Guide*. ICL, UTK, November 2009.

[20] J. Wilkinson, C. Bounds, T. Brown, B. Gerbi, and J. Peltier. Cancer-radiotherapy equipment as a cause of soft errors in electronic equipment. *Device and Materials Reliability, IEEE Transactions on*, 5(3):449–451, 2005.