Evaluating the Viability of Application-Driven Cooperative CPU/GPU Fault Detection

Dong Li, Seyong Lee, and Jeffrey S. Vetter

Oak Ridge National Laboratory, Oak Ridge, TN
{lid1,lees2,vetter}@ornl.gov

Abstract. Trends in high performance computing are bringing increased heterogeneity among the computational resources within a single machine. The heterogeneous CPU/GPU platforms, however, exacerbate resilience problems faced by current large-scale systems. How to design efficient resilience strategies is critical for the wider adoption of heterogeneous platforms for future exascale systems. The conventional resilience strategy for GPU brings significant performance and power overhead, because they employ a one-size-fits-all approach to enforce uniform data protection. In addition, the isolation between CPU and GPU protection loses potential optimization opportunities provided by the heterogeneous CPU/GPU platforms. In this paper, we explore the viability of using an application-driven CPU/GPU cooperative method to detect faults occurred on GPU global memory. By selectively protecting application-critical data and leveraging time and space redundancy in CPU to detect faults, we bring only 2.2% performance overhead while capturing more than 90% errors that cause incorrect application results.

Keywords: fault detection, heterogeneous computing.

1 Introduction

Trends in high performance computing are bringing increased heterogeneity among the computational resources within a single machine. For example, the GPU accelerator, a major source of heterogeneity today, is already adopted in HPC systems. With appropriate mapping between workloads and heterogeneous resources, the heterogeneous platforms can accelerate performance and/or improve energy efficiency for applications with varying performance characteristics. However, the heterogeneous platforms exacerbate resilience problems faced by current large-scale systems. First, the GPU-like accelerator brings higher processor density and memory bandwidth consumption, which tends to cause higher error rates. Second, the diversity of devices demands various fault protection mechanisms, and these mechanisms must be customized to the needs of architecture and system software. How to design efficient resilience strategies for heterogeneous platforms is an open question. The answer to this question is critical for the wider adoption of heterogeneous platforms for future exascale systems.

The conventional resilience mechanism usually employs a one-size-fits-all approach: a uniform protection is enforced, regardless of application semantics.

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 670–679, 2014.

As a result, a number of resilience mechanisms (e.g., hardware ECC, process replication, and system-level checkpoint/restart) come with large performance and power overhead. However, our recent work [1] reveals that the application data structures can display diverse vulnerability because of variant application behavior and memory access patterns. It is possible to use an application-driven approach to selectively protect data, such that we can reduce fault tolerance overhead.

Furthermore, the conventional resilience strategy enforces isolated protection in most cases. For example, on high-end computing systems, the main memory on the CPU side can be protected by heavyweight Chipkill ECC, while the global memory on the GPU side has no protection or has protection with lightweight SECDED ECC. There is no cooperation for data protection between CPU and GPU, and also the resilience mechanisms are separated from each other. However, heterogeneity provides new opportunities to improve resilience. In particular, each computing unit (e.g., CPU or GPU) can provide space and time redundancy to another. Hence, it is possible to implement a cooperative resilience mechanism by shifting some protection responsibility for one computing unit to another. This cooperative methodology can potentially reduce overhead accompanied with resilience mechanisms.

In this work, we rethink the traditional resilience strategy on a heterogeneous CPU/GPU platform, and explore the viability of using an application-driven CPU/GPU cooperative method to detect faults occurred on GPU global memory. In particular, we investigate a software-based fault detection mechanism on GPU. This resilience mechanism provides protection to data based on application knowledge and data vulnerability analysis, and greatly reduces fault detection overhead. We reveal that this resilience mechanism, however, is not suitable to be completely implemented on GPU, because it has abundant control flow and limited parallelism. This results in low occupancy and poor power efficiency on GPU. In addition, completely implementing the fault detection on GPU may increase memory footprint, which prohibits us from using GPU computing with larger input problems. To solve the above problems, we shift parts of GPU fault detection logic to CPU and embrace a cooperative fault detection. Our method brings only 2.2% performance overhead while capturing more than 90% errors that cause incorrect application results.

2 Related Work

The GPU was originally designed for applications that are intrinsically fault tolerant (e.g., image rendering). Hence it did not have any resilience mechanism. However, the wide adoption of GPU for scientific applications makes GPU reliability become a concern. In fact, recent work has shown that the error rate on some GPUs can be as high as four failures per week [4].

Hardware-Based Resilience on GPU: Fermi is the first GPU that supports hardware ECC. Fermi's register files, shared memory, L1 and L2 caches and DRAM memory are SECDED ECC protected [9]. With hardware ECC,

the application can suffer from large performance loss. For example, LAMMPS suffers from 30% performance loss [5], FFT, S3D and SCAN suffer from 21%, 20% and 28% respectively based on our own experience. Jeon and Annavaram propose a hardware-based approach to opportunistically detect GPU computation errors [6]. They exploit idle resource (idle cores from underutilized warps or idle execution unit) to implement dual modular redundancy. Similarly, Tan and Fu [7] leverage the idle time in GPU streaming processors during branch divergence and pipeline stalls to implement computation redundancy to detect errors. Sheaffer et al. [8] introduce a series of architecture modification to implement hardware redundancy (e.g., redundant ALU).

In general, hardware-based resilience demands architectural support and lacks flexibility. It is difficult to inform hardware of the application semantics for error detection, and hence the hardware-based approach cannot provide flexibility required by adaptive low-overhead protection.

Software-Based Resilience on GPU: Yim et al. [10] introduce a guardian process to intercept crash events and restart the GPU kernel using checkpoints. They also instrument the source code, duplicate non-loop code, and insert range checking code for loops. Dimitrov et al. [11] investigate three software redundant execution approaches. In the first approach, they simply duplicate memory copy and kernel executions; in the second approach, they selectively duplicate instructions to interleave redundant execution with the original code; in the third approach, they leverage thread-level parallelism by assigning redundant computation to redundant thread blocks.

The above software-based approaches can introduce significant performance overhead (15%-100%), because they are ignorant of data vulnerability difference and enforce a uniform protection. Our work is different from them by selectively protecting data. Also, we partition the fault detection functionality between CPU and GPU, and leverage time and space redundancy provided by the heterogeneous platform to reduce protection overhead.

3 Approach

We aim to implement a software-based fault detection (SBFD) mechanism for GPU. Compared with hardware-based mechanisms, the SBFD provides great flexibility to control how to enforce protection. SBFD allows programmers to implement tunable protection according to application-specific vulnerability without introducing hardware overhead. Our study focuses on fault detection on GPU main memory. We assume that data resident on other architectural subsystems (e.g., register file and data bus) is well protected by hardware ECC or MCA [3]. In addition, we limit our study to the silent soft error because of its concealing nature to applications and programmers. Any soft error that causes bit flips on memory cells should be captured by our method.

To evaluate the viability of our approach, we select the benchmark k-means from Rodinia benchmark suite [12], since k-means is extensively used for many data intensive scientific applications, including geo-statistics, astronomy, and

```
1     do{
        delta = 0.0F;

        /*The clustering loop offloaded to GPU*/
        for(tid=0; tid<nthreads; tid++)
6       {
          for (ii=0; ii<_UNROLLFAC_; ii++) {
            i = tid + ii*nthreads;
            max_dist = FLT_MAX;

11          /* find the cluster center id with min distance to pt */
            ...

              /* if membership changes, increase delta by 1 */
              if (membership[i] != index) delta += 1.0F;
16
           /* assign the membership to object i */
           membership[i] = index;

           /* update new cluster centers */
21         partial_new_centers_len[tid][index]++;
           for (j=0; j<nfeatures; j++)
             partial_new_centers[tid][index][j] += feature[i][j];
          }
        }
26
        /*Array reduction performed on CPU*/
        for (i=0; i<nclusters; i++) {
          for (j=0; j<nthreads; j++) {
            new_centers_len[i] += partial_new_centers_len[j][i];
31          partial_new_centers_len[j][i] = 0;
            for (k=0; k<nfeatures; k++) {
              new_centers[i][k] += partial_new_centers[j][i][k];
              partial_new_centers[j][i][k] = 0.0F;
            }
36        }
        }

        /* replace old cluster centers with new_centers on CPU*/
        for (i=0; i<nclusters; i++) {
41        for (j=0; j<nfeatures; j++) {
            if (new_centers_len[i] > 0)
              clusters[i][j] = new_centers[i][j] / new_centers_len[i];
            new_centers[i][j] = 0.0F;
          }
46        new_centers_len[i] = 0;
        }
      } while (delta > threshold && loop++ < 500)
```

Listing 1.1. The major computation loop of k-means

computer vision. The pseudo-code for the major computation loop of k-means is shown in List 1.1. In general, k-means divides a set of data objects into clusters (represented by the array *membership*) according to object features (the array *feature*). Each cluster is represented by the mean value or centroid (the array *new_centers*). The number of data objects in each cluster is represented by the array *new_centers_len*. The algorithm iteratively associates each data object with its nearest centroid based on some chosen distance metric. Within each iteration, the new centroid is re-calculated by averaging the features of all data objects within each cluster. The benchmark offloads the data intensive clustering workload into GPU, while updating centroid happens on CPU. The arrays *partial_new_centers* and *partial_new_centers_len* save temporal data where the GPU clustering produces and consumes. These two arrays are iteratively offloaded to CPU, and CPU performs reduction operations on them to update centroids.

In this section, we will first present our vulnerability study for five data structures (i.e., *clusters, feature, membership, partial_new_centers* and *partial_new_centers_len*) in the k-means GPU kernel in Section 3.1. These data structures take close to 100% of global memory footprint of the GPU kernel.

There are other thread-private data (e.g., *tid, nclusters*, and *nthreads*). These data are used within control flows and array indices, and hence they are critical to the application resilience. However, these data structures are small in terms of size. They can be protected by heavyweight hardware ECC (e.g., double-chipkill) without causing noticeable performance and energy overhead; hence they are not the focus of our study. Based on the vulnerability study in Section 3.1, we discuss our selective data protection method and present how it is implemented with a cooperative CPU/GPU approach in Section 3.2.

3.1 Fault Injection

We develop a user-level fault injection framework to emulate single-bit and multi-bit transient faults by flipping bits in specific program variables (e.g., a data array). The framework consists of a set of C library functions to inject faults and an enhanced GPU CUDA compiler. To inject faults, the user needs to decide the fault injection target (i.e., which program variable), the condition to inject faults (e.g., the fault should occur after a specific program statement or occur at a specific iteration), and the specific faulty position within the target variable. The compiler then inserts the fault injection function calls into the GPU kernel based on user decisions. At runtime, a GPU thread will trigger bit flips based on those function calls.

To study vulnerability of data structures, we fully emulate the randomness of fault occurrences. In particular, we emulate both temporal and spatial randomness. The temporal randomness means that the fault occurs randomly during the GPU kernel execution; the spatial randomness means that the fault occurs randomly in any position with the target variable. To minimize computation and memory overhead for emulating the random fault injection on GPU, we use a CPU/GPU cooperative approach. In particular, CPU generates necessary random numbers and passes them to GPU either through GPU kernel parameters or through GPU global memory, and then GPU performs actual fault injections using those random numbers.

Figure 1 shows the fault injection results. We perform fault injection tests for each data structure for 200 times. In each test, we use a randomly selected GPU thread to randomly inject one fault (either single-bit, 2 bits, or 4 bits) into the target data. The figure indicates that *membership* is the most resilient to errors; this is due to two factors: *self error containment* and *self error correction*. As shown in List 1.1, the corruption in *membership* may change the value of *delta* (line 15 in List 1.1), which in turn may increase the number of invocations of the enclosing *do-loop* (line 48 in List 1.1). However, the error is not propagated to any other code section (i.e., the self error containment), and a correct value will be re-assigned to the *membership* in the next iteration of the do-loop (line 18 in Listing 1.1) (i.e., the self error correction).

In fact, *partial_new_centers* and *partial_new_centers_len* data structures also have the similar self-error-correction property; even though corruptions in these data may be propagated to CPU regions, CPU will reset these data at each iteration of the do-loop (line 31 and 34 in List 1.1). However, Figure 1 reveals

that these two data structures display very diverse vulnerability; *partial_new_centers_len* is very vulnerable, while *partial_new_centers* is not. This divergent error behavior is mainly due to the accuracy difference between integer operations and floating point operations on GPU. The integer addition, which is the main operation performed on *partial_new_centers_len*, does not lose accuracy as long as there is no overflow. Therefore, most of the faults in *partial_new_centers_len* can affect the application output data, *clusters*. On the other hand, the floating point addition, which is the main operation for *partial_new_centers*, may lose accuracy due to truncation and roundoff errors inherited in floating point operations. This fundamental inaccuracy loss sometimes works as a filter to remove errors. For example, if a fault accidentally happens to the exponent portion of an array element in *partial_new_centers*, the addition operation may truncate out the faulty element during internal normalization when the corrupted value is much smaller than the original value. In this case, if the original value was zero or already much smaller than the other operand , this truncation will not affect the output. Therefore, not all faults injected to *partial_new_centers* are propagated to the output data *clusters*. Moreover, propagation of errors in *partial_new_centers* to the output data is dependent on other data, *new_centers_len* (line 42 in List 1.1); if *new_enters_len* has a zero value, the corresponding element of the output data will not be updated. In this case, the errors in *partial_new_centers* will not be propagated to that element of the output data.

We also found that the output data *clusters* is less susceptible to fault injection than other data structures. This is because k-means has an inherent fault-tolerant property. The main do-loop repeats computations until the output data meet certain conditions. If a fault is injected to *clusters*, it is likely that the corruption changes the output data in a way not to meet the termination condition. Therefore, errors in the output data may increase the number of iterations of the do-loop, but the increased iterations may self-correct the output data.

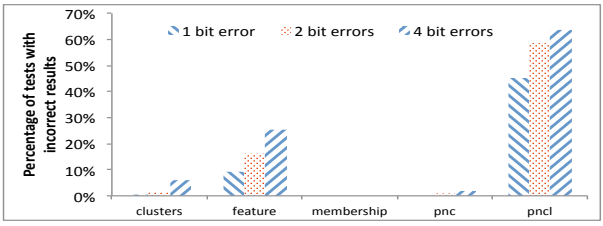

Fig. 1. Fault injection results. pnc=partial_new_center, pncl=partial_new_center_len.

3.2 Cooperative CPU/GPU Fault Detection

Based on the fault injection and application knowledge, we conclude the data structures *feature* and *partial_new_center_len* are the most vulnerable. Assuming that bit errors are uniformly distributed across time and memory space and independent of each other, only capturing the errors in these two data structures will capture 99% (1 bit error), 97%(2 bits error) and 92% (4 bits error) errors in global memory that cause incorrect application results. Moreover, only protecting these two data structures instead of all application data can reduce fault detection overhead.

We explore efficient protection for the two data structures at the software level. The array *feature*[NOBJS][NFEATURES] ($NOBJS$ is the number of data objects, and $NFEATURES$ is the number of features) is a read-only two dimensional array in the GPU kernel. A specific element *feature*[x][y] specifies the *yth* feature value for the *xth* data object. In general cases, the number of data objects is much larger than the total number of GPU threads, and each thread is in charge of $\_UNROLLFAC\_$ data objects ($\_UNROLLFAC\_ = 100$); also the number of features is much less than the total number of GPU threads. To protect *feature*, we add a checksum value for each data object. The checksum i is the summation of all feature values for the data object i. The baseline checksums are calculated before the do-loop, and saved for checksum verification. At the end of the GPU kernel, the checksums are re-calculated and verified with the baseline. If there is any difference, the fault is detected at the *feature* array.

With the above checksum-based fault detection, each thread is involved in a limited number of floating-point addition operations (i.e., $O(NFEATURES)$), and the load is well balanced between threads. However, to completely implement the above algorithm, we must transfer the baseline checksums to the GPU global memory before each kernel invocation. The size of the baseline checksums is in proportion to $NOBJS$. Furthermore, uploading the baseline checksums to GPU exposes them to the same memory architecture as other GPU data, and hence they suffer the similar error rate and there is no sufficient protection for the baseline checksums.

To solve the above problem, we leverage memory space redundancy on the CPU side to save the baseline checksums. On the CPU side, the main memory is usually better protected with hardware ECC. Hence, we ask CPU to keep the baseline checksums. Furthermore, we decouple the fault detection into checksum calculation and checksum verification, and ask CPU to perform the checksum verification. In addition, the checksum verification can be overlapped with the GPU computation, hence removing the verification overhead off the computation critical path.

The data structure *partial_new_center_len*[NTHREADS][NCLUSTERS] ($NTHREADS$ is the total number of GPU threads, and $NCLUSTERS$ is the number of clusters) is an array with both read and write accesses. Because it is not read-only, the above checksum-based approach cannot work. To detect faults, we leverage an implicit invariance existing in *partial_new_centers_len*. In particular, each element $partial\_new\_centers\_len[x][y]$ is reset as zero before the GPU

kernel invocation, and then adds by at most 1 at each iteration of the inner loop (the line 21 in List 1.1). Hence, after the GPU kernel is finished, we can bound the value of each element in *partial_new_centers_len* by $[0, \_UNROLLFAC\_]$. We check the validness of each element with CPU after the kernel is finished. This fault detection method does not add any extra computing on GPU. The element value validation occurred on CPU can also be overlapped with the GPU kernel execution, hence minimizing the fault detection overhead.

Discussion: We do not claim the full fault coverage with the above fault detection algorithms. For *feature*, like other checksum-based fault tolerance algorithms [13] [14], the checksum can result in false negative tests when the error effects from multiple array elements are nullified when calculating checksum. However, this kind of scenario is extremely uncommon, given the rareness of soft errors. In addition, the error in *partial_new_center_len* can still make the element value fall within the bound, hence resulting in false negative tests. However, this kind of scenario is much less common than the other scenarios detectable by our algorithm, assuming that the error has an even chance to occur on every data bit. Hence, the invariance-based algorithm can still capture most of errors.

4 Evaluation

We evaluate our approaches in this section. Figures 2 and 3 compare data transfer size and performance for three fault detection scenarios, including *no resilience* (i.e., the original code), *GPU-only* (i.e., fault detection implemented on GPU only), and *GPU+CPU* (i.e., cooperative fault detection)

Figure 3 shows that *GPU-only* brings 11.1% performance loss, comparing with *no resilience* scenario. The performance loss is due to two reasons: first, the extra data transfer from CPU to GPU (shown in Figure 2) occurs to fetch the reference *feature* checksum data to GPU; second, the extra fault detection logic is added into the application critical path on GPU. *GPU+CPU* significantly reduces the data transfer from CPU to GPU by 23%, because data verification logic is offloaded to CPU, and the reference checksum data does not have to be uploaded to GPU. However, the data transfer from GPU to CPU increases, because GPU has to offload updated checksums to CPU for verification at each iteration. As a result, the total data transfer size between GPU and CPU for the *GPU+CPU* is the same as that for GPU-only. However, the performance of *GPU+CPU* is improved by 8.1%, comparing with that of *GPU-only*. Considering the same data transfer size for the two scenarios, the performance improvement must come from the overlapping of CPU and GPU work. Note that we have to synchronize CPU and GPU when overlapping CPU and GPU work. However, even with the synchronization overhead, we still get performance improvement. In general, the optimized cooperative CPU/GPU fault detection brings only 2.2% performance loss, comparing with *no resilience* scenario.

Furthermore, considering the fact that the data verification logic with *GPU+CPU* is implemented with only one single CPU thread, while with *GPU-only* the logic is implemented with massive parallelism with low GPU occupancy, we derive that the *GPU-only* implementation is not as power efficient as

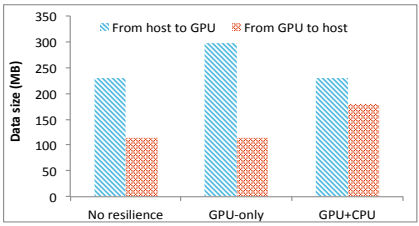

Fig. 2. Compare data transfer size between various resilience scenarios

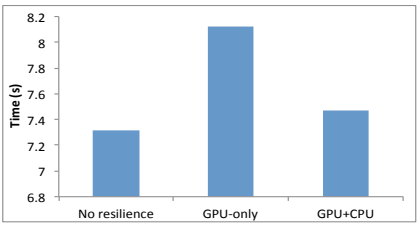

Fig. 3. Compare performance between various resilience scenarios

GPU+CPU, because CPU can power down the idle cores (CPU only uses one thread) while GPU cannot. Also, the thermal-design power (TDP) of the main stream GPU is comparable or even much larger than CPU. Hence, using CPU to implement data verification is more power-efficient.

Figure 4 displays the fault injection results with our fault detection mechanism. There are four groups of bars within the figure. The first two groups of bars (i.e., *feature* and *r_feature*) are for the results when injecting faults into *feature*, while the last two groups of bars (i.e., *pncl* and *r_pncl*) is for *partial_new_centers_len*. The first and the third groups of bars represent the percentage of fault injection tests that causes incorrect application results. These bars are already shown in Figure 1, but we list here again to show if our fault detection mechanisms have captured them. The second and the fourth groups of bars represents the percentage of faulty tests captured by our mechanism. We perform 200 fault inject tests for each test. The y axis represents the percentage of tests that have incorrect application results (i.e., visible errors) for the first and the third groups of bars, or the percentage of tests that have errors detected by our mechanism for the third and the fourth groups of bars.

The figure shows that for *feature* and *pncl* our mechanism captures all of errors that cause incorrect application results. However, our mechanism reports more errors than necessary (i.e., having false positive cases). Those cases do not cause incorrect application results because of statistical nature in k-means. However,

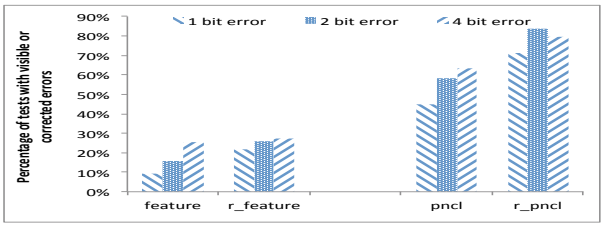

Fig. 4. Fault injection results with cooperative fault detection. pnc=partial_new_center, pncl=partial_new_center_len; r_* refers to the cases with fault detection enforced.

our mechanism only focuses on data structures themselves without application algorithm knowledge, hence resulting in false positive fault detection.

5 Conclusions

In this paper, we target on reducing fault detection overhead on a heterogeneous CPU/GPU platform. Based on the fault injection tests and application knowledge, we selectively protect data according to data-structure-specific vulnerabilities, instead of using a one-size-fit-all approach to protect data. This application-driven approach can capture most of errors that cause incorrect application results, while resulting in reduction in resilience overhead. Furthermore, by dismantling the fault detection mechanism and offloading part of it to CPU, we can further reduce the detection overhead.

References

1. Li, D., Vetter, S.J., Yu, W.: Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In: SC 2012 (2012)
2. Honarkhah, M., Caers, J.: Stochastic Simulation of Patterns Using Distance-Based Pattern Modeling. Mathematical Geosciences (2010)
3. Iyer, R., Nakka, N., Kalbarczyk, Z., Myra, S.: Recent Advances and New Avenues in Hardware-Level Reliability Support. IEEE Micro (2005)
4. Haque, I.S., Pande, V.S.: Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In: CCGRID 2010 (2010)
5. Brown, W.M.: GPU Acceleration in LAMMPS. In: LAMMPS User's Workshop and Symposium (2011)
6. Jeon, H., Annavaram, M.: Warped-DMR: Light-Weight Error Detection for GPGPU. IEEE Micro (2012)
7. Tan, J., Fu, X.: RISE: Improving the Streaming Processors Reliability Against Soft Errors in GPGPUs. In: PACT 2012 (2012)
8. Sheaffer, J., Luebke, D., Skadron, K.: A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors. In: Proceedings of Graphic Hardware (2007)
9. ECC and Keeneland GPUs, `http://keeneland.gatech.edu/`
10. Yim, K.S., Pham, C., Saleheen, M., Kalbarczyk, Z., Iyer, R.K.: Hauberk: Lightweight Silient Data Corruption Error Detector for GPGPU. In: IPDPS 2011 (2011)
11. Dimitrov, M., Mantor, M., Zhou, H.: Understanding Software Approaches for GPGPU Reliability. In: GPGPU 2009 (2009)
12. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: IISWC 2009 (2009)
13. Wu, P., Ding, C., Chen, L., Gao, F., Davies, T., Karlsslon, C., Chen, Z.: Fault Tolerant Matrix-Matrix Multiplication: Correcting Soft Errors On-Line. In: Workshop on Latest Advances in Scalable Algorithm for Large-Scale Systems (2011)
14. Huang, K.-H., Abraham, J.A.: Algorithm-Based Fault Tolerance for Matrix Operations. IEEE Transactions on Computers C-33(6), 518–528 (1984)