Dept. of Electrical and Computer Engineering
University of Wisconsin Madison

Performance-cost analysis of Software Implemented Hardware Fault Tolerance Techniques

Ameya V. Abhyankar
Advisor: Prof. Kewal Saluja

May 14, 2010

Performance Cost Analysis of Software-Implemented Hardware Fault Tolerance Techniques

Ameya V. Abhyankar

***Abstract*—As Moore's law projections continue to hold true, more and more transistors are getting integrated on a single chip. These transistors are being smaller each passing technology generation. Forecasts suggest that the reliable operation of future devices with continuously shrinking geometries cannot be guaranteed. Currently used hardware-based fault tolerance techniques are expensive in terms of the hardware modifications, the corresponding expenses on verification and testing and loss of volume savings. Hence there is a need for high-level techniques that will ensure reliability even when the underlying hardware is not reliable. The advent of multi-cores coupled with the accompanying exponential increase in transistor counts threatens to make the problem even more severe. This report compares the performance cost of Software-Implemented Hardware Fault Tolerance (SIHFT) techniques on two distinct architectures: CPUs and GP-GPUs. The choice of multi-core architectures is due to the belief that they are the way to go for all processing cores of the future. We analyze the relative overhead of implementing several SIHFT techniques on four different computational kernels, present and analyze the relative overheads of implementing them and highlight the architectural features that lead to different tradeoffs on CPUs versus GPUs. The results of this study present insight into the cost of implementing SIHFT-based fault tolerance and the role that different architectural model play and impact these costs.**

I. Introduction

Over the years, electronic circuits have become smaller and more complex. Moore's Law [1] projections have held so far and technology roadmaps suggest that they will hold for at least a few more generations [2]. Each reduction in process size has brought with it a commensurate increase in transistor count. However, with each reduction in process size there have been increasing concerns about the reliable operation of these semi-conductor devices.

The decrease in transistor size and increase in transistor density affects reliability in two ways: increasingly complex circuits make formal and comprehensive verification costly and difficult, and reliable fabrication of small feature sizes is difficult [3]. These factors imply that the post-silicon behavior of a design may not be as expected. Fundamentally, the major reliability threats are from the following factors:

1. Silicon defects during production.
2. Failures and noise effects at run time due to high transistor density and the ever-increasing number and density of interconnect.
3. Extreme process variations caused by the physical limits of manufacturing techniques.
4. Design errors caused due to the ever increasing challenge of exhaustive verification.

These challenges threaten future scaling of silicon devices. A major reason for making transistors smaller is to reduce cost and enhance the capabilities of electronic devices. However, if design, verification and manufacturing become more expensive due to reliability concerns, it will increase the cost of the product and thereby, defeat the purpose of having more and smaller transistors on a chip. At the same time, there will be additional replacement, repair and serviceability costs involved. Furthermore, a large part of the transistor budget of the chip may be expended on building reliability-improving features on the chip causing performance degradation: highly undesirable by the designers and the users alike. This is an irony considering that among the many advantages/goals of device scaling is to improve performance and provide a larger transistor budget to the designers.

Many high-throughput and compute-intensive applications as well as the increasingly complex client applications run on commodity processors that don't support robust fault tolerance at the hardware level. Hence, there is a need for a flexible and scalable scheme at the architectural or programming levels to improve system reliability.

We choose to take a look at SIHFT techniques because these techniques offer a good trade-off in performance, cost, and flexibility. SIHFT techniques don't require any hardware modification and allow for selective use of fault tolerant mechanisms: something that hardware based techniques can't achieve. Further, software-based techniques can be used to improve the reliability of existing commodity processors. They are also effective even in circumstances where errors are intermittent and don't justify the power and performance overhead of hardware-based redundancy.

We decided to take a look at two diverse architectures: Central Processing Units (CPUs) and General-Purpose Graphics Processing Units (GP-GPUs). CPUs are characterized by their Instruction Level Parallelism oriented architectures and large caches. They typically have a small number of complex cores and are primarily geared to reduce latency. GP-GPUs on the other hand are targeted at compute-

intensive applications and are geared to exploit Thread-Level and Data-Level Parallelisms. They have a large number of relatively simple cores and small caches. They tolerate memory latency relatively well but generally fare poorly in control intensive applications. GP-GPUs have changed the GPU computing by allowing some programmability of the cores: A trend first started by NVIDIA through CUDA [4] [5]. We use CUDA-capable NVIDIA GP-GPUs and Intel CPUs for our experiments.

The motivation behind choosing these two architectures has a lot to do with the advent of multi-core processors in the commodity segments. If the current trends continue, the number of cores is bound to keep on increasing. The Intel CPUs of today are at one extreme of the multi-core era: trying to provide multiple cores while still maintaining the abstraction of serial uni-processor execution. The GP-GPUs on the other hand are at the other extreme: exposing the muti-core architecture (to some extent) and making the programmer deal with the intricacies. The multi-core processors of the future are bound to lie somewhere in between these two extremes. Our results will try to evaluate these architectures objectively. However, justifying which of the architectures is better or more promising is beyond the scope of this work.

The rest of the report is organized as follows: Section 2 describes the related work in the field. Section 3 discusses CPU and GPU architectures. Section 4 describes the various SIHFT methods studied in this report. Section 5 describes the hardware infrastructure and the simulation methodology used for this study. In Section 6, we show the results and discuss the reasons for the trends. Section 7 provides some insight on the implementability of the algorithms. We conclude the report in Section 8.

II. Related work In Hardware

Error Detection by Diverse Data and Duplicated Instructions (ED4I) by Oh et al. [6] is a SIHFT method that looks at instruction re-execution with diverse data. The scheme targets permanent and transient errors and focuses on finding the best multiplication factor for embedded systems.

Rotenberg [7] proposes a microarchitecture-based fault-tolerance scheme that uses Active and Redundant streams, called AR-SMT. In this scheme, a redundant thread runs parallel to active threads, and a result comparison is performed to check for errors before the instruction commits.

Rashid et al. [8] propose re-execution on multiscalar architectures as an efficient method for implementing fault tolerance without the need for much hardware modification.

Mahmood [9] and DIVA [10] suggest the deployment of a watchdog processor or co-processor to perform dynamic runtime verification of results and guard against faults in high-speed and small-dimension circuits.

Erez et al. [11] discuss the Fault Tolerance techniques for the Merrimac Streaming Supercomputer. The novel feature is that they allow programmability of the hardware/software techniques available and leave it to the programmer to either run the program at full speed using SIHFT or at a compromised speed using hardware redundancy methods.

Our report draws inspiration from some of the above work, but sticks to implementing fault-tolerance strictly at the programming level with no hardware modification.

III. Architecture

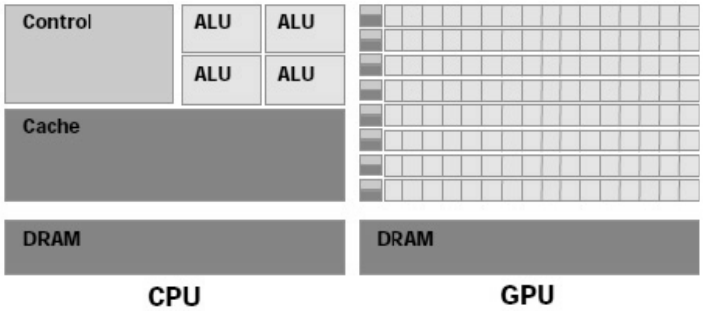
A. CPU Architecture

Because SIHFT techniques are particularly important on systems without robust hardware-based fault tolerance, we chose to evaluate them using commodity, multi-core, x86-based CPUs. Since the general architecture of CPUs is familiar to most system designers, we will forego an in-depth discussion. The specific CPU models used in our experiments are described in Section V.

B. CUDA and GP-GPU Architecture

To understand the reasons that certain SIHFT methods may perform differently on a GPU compared to a CPU, it is critical to understand the underlying architectural differences between the two. Rather than focus on details of a specific microarchitecture, we will base our discussion of GPU architecture on NVIDIA's Compute Unified Device Architecture (CUDA) [5] [4], an architectural model designed for GP-GPU. We choose CUDA since it is the most stable environment for GP-GPUs and is a good example of limited abstractions and the challenges of programming a GP-GPU.

The development of GPU architecture has been driven by graphics applications. These applications tend to feature regular control graphs, streaming operations, large amounts of data parallelism, and high floating-point throughput requirements [12]. As a result, GPUs feature a large number of relatively simple cores and devote most of their transistor and power budgets to arithmetic units rather than on-chip caches or control structures. By devoting most of the transistors to ALUs, GPUs are able to achieve significantly better peak floating point performance [13] [14] and energy per instruction [15] than comparable commodity CPUs. This means that the GP-GPUs are very good at computations while being not so good at control. In fact, the GPU that we are using is among the first to incorporate a degree of control programmability. This inclusion of programmability is however a double edged sword, as while it gives a degree of programmability, the control is not like a CPU: it has much more overhead than a CPU. This difference between CPUs and GPUs is illustrated in Fig.1. Fig. 1: Difference in resource



allocation of CPUs and GPUs.

CUDA uses a clustered processor model. Threads are scheduled 32-thread groups known as 'warps'. The programmer arranges these warps into multidimensional arrays known as 'thread blocks' that are assigned to processor clusters. To reduce control overhead, all concurrently executing threads in a cluster share a program counter. In order to take divergent control paths within the same warp, CUDA GPUs use branch predication [13], which requires every thread in a warp to execute all branching paths. The use of predication means that programs with complex control flow and multiple divergent control paths are ill-suited to the GPU. This causes the threads to execute the diverging paths one at a time. This serialization of these sections causes a serious slowdown in the execution of the threads.

As shown in Fig. 1, the GPU architecture eschews most of the control hardware present in a modern CPU to devote those resources to extra cores. GPU cores are in-order and lack sophisticated instruction issue, branch prediction, or register-renaming logic. As a result, individual threads may easily become stalled due to branch resolution or data dependency issues. The GPU tolerates these problems through the use of low-overhead simultaneous multi-threading. When a thread in a thread block stalls waiting for a memory access or branch resolution, the scheduler swaps in another thread block, if available. As the number of threads in the application increases, the GPU's ability to tolerate these stalls improves. This is possible because the GPU features very low-overhead thread scheduling mechanisms. At the same time, as the data set size increases, the performance of the GPU tapers off as the threads can no longer negate the effect of memory misses.

One of the most significant aspects of CUDA is its memory hierarchy. Communication between clusters is done through a high-latency global device memory. The CUDA model uses a shared memory scratchpad rather than a cache to allow for low-latency communication between threads within the same cluster. Registers are statically partitioned based on the number of active thread blocks. Increasing the number of threads reduces the number of registers available per thread. If threads require more registers than are available in hardware, registers are spilled to global memory. Fig. 2 shows the GPU memory hierarchy.

GPUs use very wide memory buses that allow the global memory to achieve a memory bandwidth of the order of 100 GB/s – several times higher than typical CPUs [5]. However the device memory's latency is similar to that of a CPU main memory and amounts to several hundred cycles. In contrast, the GPU's arithmetic units are very low-latency. This large disparity has given rise to a model in which computations on the GPU can be viewed as nearly 'free' since the time it takes to compute results is often far exceeded by the time it takes to transfer data to and from global memory.

Performance optimization on the GPU requires management of many resources. Correctly balancing the number of registers and threads, eliminating divergent control paths, fitting the working set inside of shared memory, and properly orchestrating memory accesses are critical to achieving peak performance [16]. When modifying pre-optimized applications to introduce SIHFT, perturbations to the optimized code can potentially result in unexpected performance effects. GPU characteristics like register-spilling, data-sets spilling on to main memory and so on are hard to justify accurately without actual computations. Therefore it is important to evaluate any application changes across a range of input sizes.

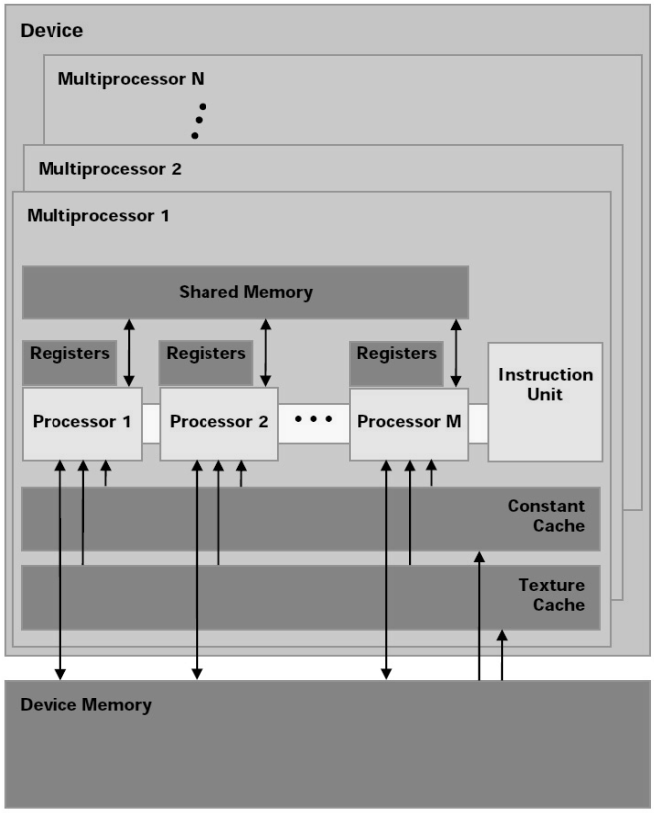


Fig. 2: CUDA memory hierarchy.

IV. SIHFT TECHNIQUES

Commercial Off The-Shelf (COTS) systems are usually designed without too much consideration for fault tolerance [17] apart from the customary ECC checks on memories, some mechanisms on I/O and in case of large applications such as databases/servers, high-overhead techniques like Dual/Triple Modulo redundancy [26]. Due to their low cost, it is becoming increasingly desirable to integrate COTS components into large-scale high-throughput computing systems. Commercial considerations such as time-to market and cost prevent the adoption of most of the currently known fault tolerance techniques from appearing in COTS systems. On the other hand, feasibility and continuous power-performance overheads are major concerns for large scale computers. Therefore, SIHFT techniques which require no extra hardware and a few software modifications are attractive options to developing fault-tolerant systems from these components.

When considering which SIHFT techniques to evaluate, the cost and feasibility of implementation was a primary concern. Zero hardware modification and flexibility of implementation were the other axes of selection of a fault tolerance

mechanism. We chose Re-execution (Time Redundancy), Algorithmic Redundancy, Space Redundancy and Data Diversity as the four basic redundancy techniques for this study. These techniques provide good coverage, while being practical to implement on a COTS microprocessor [17]. These techniques require no other hardware apart from the microprocessor itself but do have some overhead in terms of performance and power. Moreover, SIHFT techniques are also much more cost-efficient than comparable hardware techniques. The cost of adding fault tolerance to a COTS processor would be prohibitive given the expenses incurred in design, verification and testing of the modified processors. It would be more efficient just to have a few more processing nodes to overcome the performance cost of SIHFT than trying to modify existing hardware for fault tolerance. Further, SIHFT techniques also give the option of selective implementation and ease of modification/upgrades. We analyze the performance overhead cost in this report. The following sub-sections discuss the techniques used for evaluation.

A. Time Redundancy by Re-execution

Re-execution is a classical time redundancy technique for fault tolerance [8]. The code is executed back to back on the same hardware and the results are compared. If the results don't match, there must be an error.

All of the kernels used floating point numbers in the IEEE 754 format, so an absolute comparison of results of both the executions would not necessarily show identical results due to the inherent rounding-based variation in floating point computations. Therefore, we did a range check to see if the results of the two executions were within the same range of values [17] [18]. The types of faults that are targeted by this technique are transient faults (provided we wait for the transient to subside) and permanent faults (using shifted data). A major limitation of this approach is the high performance overhead on conventional processors. However, this approach is the most intuitive and simple to implement, and requires no changes in the existing hardware and minimal changes in existing software setups. There is also a good scope of reducing the performance overhead of this method provided there is some hardware support, as demonstrated in AR-SMT [7] and Rashid [8] .

B. Space Redundancy

A limitation of re-execution is that permanent faults cannot be detected or at best have limited coverage provided shifted operands are used. Since the same ALU/core is going to run both versions of the code, the error will get propagated in both the versions of the code [17] if it is permanent or if the error-causing transient hasn't passed till the time of the second execution. Therefore, the error can escape detection and render the redundancy technique futile. The solution to this issue is to use different hardware to execute the two versions of the code and then compare the results of both. GP-GPUs typically have a very large number of cores. Hence, different threads were mapped to different processing elements and their results were compared. Space diversity is also possible in multi-core CPUs. This ensures a much greater coverage and error detection capability when compared to simple re-execution. The performance overhead should not be as high as that of re-execution provided there are idle cores, but the scheduling constraints imposed by running two independent threads in parallel tends to limit the performance of COTS processors. This method can be used to handle intermittent, transient as well as permanent faults. This method differs from AR-SMT [7] which uses the same hardware to run the redundant threads. This method may seem a lot like Dual modulo redundancy, however, the main difference between the two is that while DMR requires that the redundant hardware should execute the code all the time, it has a 100% performance overhead in the least (provided the same hardware) or a 100% hardware overhead if the hardware is replicated. Spatial redundancy however, can be judiciously used, thereby having a much lesser overhead than a full-time DMR system.

C. Redundancy through Data Diversity

Data Diversity involves dividing the input region into two parts: failure and non-failure regions [17] [19]. The testing process involves executing the program with one given input and then executing once again with slightly perturbed inputs [6]. The outputs of the two runs are then compared, preferably before the computation modifies any stable storage/ architectural state. If either of the two sets of inputs lies in the failure region, the error can be detected.

In our implementation, we used the explicit perturbation method for data diversity and exact re-expression, allowing us to use the output of the perturbed set of inputs as well. We also compared the outputs of the two runs for errors. Since we were dealing with floating point numbers, we had to choose a suitable method for data diversity. Simply shifting or adding integral constants will not give a good coverage, because it will not affect the floating point components. Therefore we chose the method of multiplying the data by a floating point and/or integer constant. The fault coverage of this method depends on the data sets chosen for re-computation. It is important to note that while this may seem a lot like re-execution, it differs from it in two major ways: the fault coverage afforded by this is much higher and it involves extra computations to produce the perturbed inputs and to compare the results of the two runs.

D. Algorithm-based Fault Tolerance (ABFT).

Algorithmic Redundancy/Algorithm Based Fault Tolerance (ABFT) [20] is yet another approach to provide fault detection and diagnosis through data redundancy. It is usually implemented at the application software level and hence has

various implementations possible. ABFT techniques have been widely used for matrix-based signal processing applications such as matrix multiplication, inversion, LU decomposition and Fast Fourier Transforms [17].

In this method, ABFT for matrix multiplication is implemented using a checksum code. We used the Row checksum matrix of one of the operands (A_r) and multiplied it with the Column checksum matrix of the other operand (B_c). If A and B are both *m*n* matrices, then A_r and B_c are of size *m+1*n* and *m*n+1* respectively. The last ie. $m+1^{st}$ row and the $n+1^{st}$ column of A and B have the row and column checksums of the operands. The product matrix is a *m+1*n+1 matrix* [17] [20]. We use the first *m*n* of the product to compute the checksum of the result separately and then compare it with the checksum included in the Row and Column matrices.

For Ocean, a specific ABFT algorithm is devised. This algorithm uses just the row checksums; however, they are divided into even and odd checksums (corresponding to the even and odd elements of the array). The Ocean algorithm was studied in depth and the behavior of the checksums was looked at. Then, the ABFT procedure was drawn out so that the checksums mimic the behavior of the actual data elements (as though the checksums were computed at each step of the algorithm). The verification of checksums was the same as that for matrix multiplication.

The ABFT algorithm for FFT was inspired by the work of Wang et al. [25]. The basic methodology included adding an extra row to the FFT co-efficient matrix and computing the FFT thereafter. The last element pair (real and imaginary) in the computed FFT was matched with the product of the input signal and the extra row in the co-efficient matrix.

The ABFT procedure for SAXPY was very straightforward. Row checksums were added for both the input matrices. Then after SAXPY execution on the extended matrices, the output checksums were compared with the sums of all the rows in the output.

In general, if the checksums don't match, we have detected an error. This method can be used to detect three erroneous elements of any checksum matrix and correct up to one error. The overhead in this scheme is lower as compared to re-execution and it can be performed on existing hardware using no modifications. This scheme was earlier thought to be very restrictive and good only for matrix based computations. However, it has been shown to be a viable alternative for checkpoint/ roll-back and recovery methods for fault tolerance.

E. Expected Outcomes

As part of our study, we wished to consider whether the actual overhead costs of SIHFT techniques were consistent with the intuitive expectations that a typical programmer might have. Hence we decided to evaluate all the techniques for the above kernels on actual hardware for the purpose of performance comparison.

Re-execution is simply the re-computation of every result, so one might intuitively expect it to have a 100% overhead on both CPUs and GPUs. Spatial Redundancy seems to benefit from a multitude of cores, so it should have lower overhead on GPUs than on CPUs. Data Diversity involves computing using one different and one shared operand and a final comparison. Hence, one can expect it to have slightly higher overheads than re-execution owing to extra computations for creating and comparing diverse data sets. ABFT has quite a bit of shared data computations and hence one can expect the GPU's performance to be limited by its ability to move data between threads. Overall, although GPUs may be faster than CPUs in absolute time, the overheads should depend on the different data-sharing and computation models used in CPUs and GPUs.

V. Hardware Used

Though there are several robust CPU simulators, the currently available simulators for GPUs are limited. Therefore, we decided to implement all our methods in software and run them on real hardware systems. This would not only yield accurate, real-world results, but it would also help us in implementing strictly practical and feasible techniques.

One CPU and one GP-GPU hardware platforms were used for the study. The kernels used for the study were chosen to represent the scientific computation environment as well as parallel algorithms very well. The hardware used is briefly described below:

A. NVIDIA GeForce 8800GTX

The 8800 GTX consists of 681 million transistors covering a 480 mm² die surface area built on a 90 nm process [21] [22]. It has 128 stream processors clocked at 1.35 GHz, a core clock of 575 MHz, and 768 MB of 384-bit GDDR3 memory at 1.8 GHz, giving it a memory bandwidth of 86.4 GB/s [22].

B. Intel Core2Quad Q9400 CPU (CPU4T)

The Core2Quad is designed to handle massive compute and visualization workloads. The version that we used had 4 cores, 6 MB of L2 cache and was clocked at 2.66 GHz. It has 1.33 Ghz FSB and is built using the 45nm process technology [23].

VI. Results and Discussion

This section presents the results of the aforementioned algorithms. We have used performance overhead and not absolute timing results as the basis of comparison. The GPUs are an order of magnitude faster than the CPUs for all computations, because they have been specifically designed for compute intensive environments. Therefore, normalized execution times over the baseline are the best metric for comparing the performance of the two architectures. We talk about the cost of each of the techniques one kernel at a time. The matrix multiplication kernel is discussed the first and the other kernels thereafter are described with reference to the matrix multiplication kernel.

A. Matrix Multiplication

Matrix multiplication is a good example of a frequently used engineering/ scientific kernel. It is also an excellent example of a highly parallel algorithm and of how a seemingly serial algorithm can be broken into parallel executable sections.

i. Re-execution

Figure 3 shows the normalized execution time for Re-Execution-based fault tolerance compared to the original non-

fault-tolerant version of the code. The GPU has significantly lower overhead than the CPU for this scheme for all the matrix sizes.

For all the matrix sizes, the CPU implementation required twice as much execution time as its baseline, reflecting a linear scaling with the number of floating point operating. The variation in the overheads at 256 can be attributed to both variations in measurement accuracy caused by the very low absolute execution time on a data set of this size.

The GPU showed a much smaller penalty for the addition of re-execution. For the 256 X 256 matrix, it exhibited only a 17% increase in execution time, despite performing twice as many multiplications. This reflects the GPU axiom that

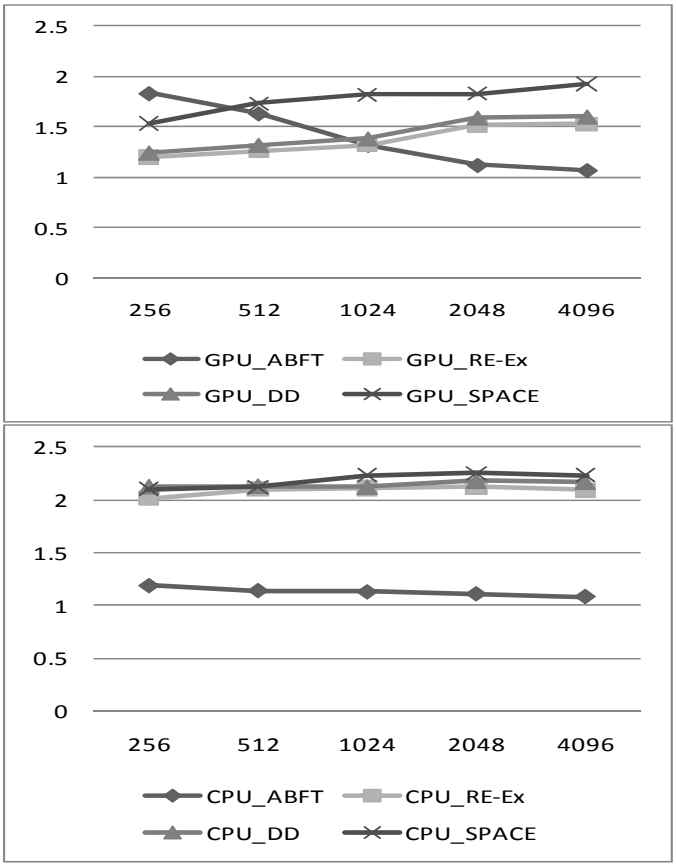


Fig. 3: Comparison of overhead for matrix multiplication

computations are nearly free compared to memory transfers. Re-executing using data already stored in shared memory produced only a small performance penalty.

The performance cost of re-execution scales upward on the GPU, reaching almost 50% for the 1024 X 1024 matrix and upwards. This increase occurs because the large matrix size requires a large number of thread blocks. As the number of blocks increases, there is an inflection point at 1024 X 1024 after which the impact of register spilling kicks in and beyond that, the overhead becomes stable right up to 4096 X 4096.

ii. ABFT

Fig. 3 shows the normalized execution overheads are as expected for CPU and GPU for ABFT. The CPU outperforms the GPU in this segment. The GPU's relatively poor performance in ABFT when compared to the other SIHFT techniques can be ascribed to two facets of the matrix checksum algorithm used: First, ABFT is a relatively control intense application with interleaved computations. CPUs are more suited for these than GPUs. ABFT computations require performing block wise matrix multiplication as well as row and column-wise checksum computation and result verification. Hence a simple parallel version of ABFT requires launching multiple kernels on GPUs. The overhead associated with each kernel launch is a significant factor of the run time. The changing memory access sequences also contribute to the time overhead. Therefore, we can see that as the matrix size increases, these overheads get amortized over a large number of elements and hence GPU performance keeps on improving. The GPUs' overhead eventually matches the CPUs when the matrix size goes to 4096 X 4096. The difference in the overheads for the two CPUs is within tolerable limits.

iii. Space Redundancy

Figure 3 shows the normalized performance when implementing space redundancy – re-execution with the second execution taking place on a different processor core. This was achieved on the CPUs by ensuring that two copies of the multiplication algorithm run on separate codes by using duplicate threads. The CPU incurs large performance penalties for this technique, due to the high overhead for scheduling, communication, and synchronization between CPU cores and the inability to cache data between executions. These performance hits can be attributed to the fact that using cores independently limits the dynamic scheduling capabilities of the processor.

The GPU is able to take advantage of its high-efficiency thread scheduling to achieve substantially less overhead than the CPU. The large number of processing elements and low-overhead scheduling on the GPU means that the scheduling limitations which hamper the CPU performance are felt much less on the GPUs - at least for small data sets. Up to the 2048 X 2048 matrix, its performance overheads are smaller than those of the CPUs. However, the cost scaling due to register spilling is amplified when using space redundancy because it generates twice as many threads as other forms of re-execution, halving the number of registers available to thread blocks. Since our CUDA GPU doesn't have automatic caches, the cost of register spilling is extremely high as the spill goes out to main memory. This comes into play at larger data sizes and leads to the trend of increasing overhead as the size of the dataset increases. The expected performance from spatial redundancy was different from the actual performance owing to several architectural factors for both CPUs and GPUs.

iv. Redundancy through Data Diversity

Fig. 3 below shows the normalized performance when implementing Re-execution with Data Diversity. To add diversity to our floating point data, we multiplied each result by 2 then divided the each element of the result matrix by 2 after it had been computed. The addition of this extra operation has some effect on the performance of the CPU when compared to conventional re-execution.

This can be attributed to the extra multiplication operations (2 times per element) that need to be performed for this scheme. The small matrix size (256 X 256) is an outlier here for the CPU with 4 cores. We believe this is due to the reasons mentioned in Re-Execution. The two CPUs overall have similar overheads.

The overheads on GPUs are much less than those on CPUs for all operations. However, there is an increase in the overhead over Re-Execution. The amount of increase in overhead is less than that of the CPUs, because floating point operations are executed much more efficiently on GPUs as compared to CPUs. Overall, the normalized execution times for CPUs followed intuition, while GPUs outperformed due to cheaper computations.

The matrix multiplication comparisons reveal that ABFT is the best technique for CPUs while the technique of choice for GPUs varies as per the size of the input matrices.

B. Ocean Simulation

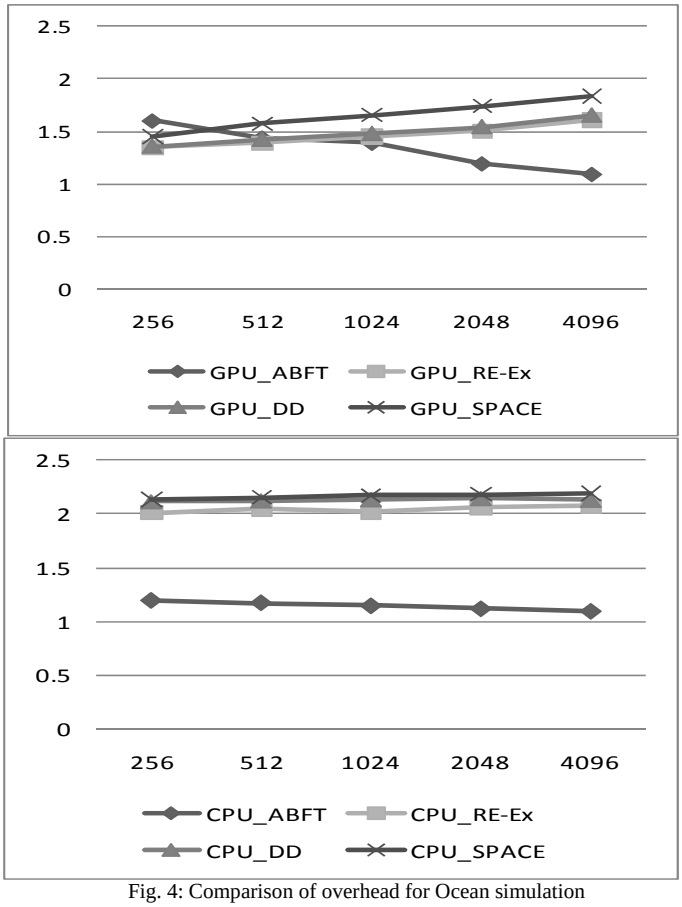


Fig. 4: Comparison of overhead for Ocean simulation

Ocean simulation is a kernel from the SPLASH2 application kernel. It is representative of a good multiprocessor application that requires inter-thread synchronization and is not exactly embarrassingly parallel. Fig. 4 above shows the results for Ocean.

i. Re-execution

The re-execution approach for Ocean shows trends that are similar to those for matrix multiplication. On the CPU, the overhead for re-execution is almost 100%. This is again due to the very nature of the re-execution method.

GPUs also show a similar trend with smaller matrices having very low overhead and the overhead keeps on increasing as the matrix size grows.

ii. Data Diversity

The data diversity trends are also the same as those for matrix multiplication: they follow the re-execution numbers closely with an extra overhead justified by the additional calculations owing to the extra computations required for the operations on creating the perturbed data sets.

iii. ABFT

The CPU shows a trend similar as that for ABFT for matrix multiplication. The overhead is close to 10% irrespective of the matrix size; in fact it goes on reducing slightly as the matrix size increases. This can be attributed to the fact that even the CPU version is able to have a better parallelization characteristic as the number of threads increases.

The performance overhead on the GPUs however shows a difference. The overhead trend is the same as that matrix multiplication, however, the values are lesser. This is due to the fact that the operations in this kernel are performed on a single matrix as opposed to two matrices in matrix multiplication. Therefore, the overhead of computing and comparing the checksum as well as the general book-keeping of the checksum is much lesser here. Though the same argument applies to CPUs as well, their effect is less pronounced due to their inherently efficient execution of the ABFT code.

iv. Space Redundancy

The space redundancy trends are similar to those of the earlier kernel for CPUs and GPUs, except for the fact that they have a little decrease in overhead. This is again due to the fact that the overall size of the data set for this kernel is half that of the earlier one.

C. SAXPY

SAXPY is a widely used benchmark for processor performance characterization. It is an example of a embarrassingly parallel kernel as well as a kernel that is very light-weight in terms of the order of computations. However, at the same time, its data-set is as big as that of matrix multiplication. Fig. 5 shows the results for SAXPY.

i. Re-execution

Re-execution for both CPU and GPU on SAXPY continues the trends of the earlier kernels.

ii. Data Diversity

Data diversity too mimics the behavior of the earlier kernels, as expected.

iii. ABFT

The performance for this algorithm however, throws a few surprises. ABFT has been by far the most efficient SIHFT techniques in this study so far, with less than 15% overhead on

CPUs for both the kernels above. However, for the SAXPY kernel, it shows a much higher overhead on CPUs, starting at nearly 1.75X and dropping as the number of threads increases (matrix size increases). This can be attributed to the very nature of the ABFT implementation. The order of computations for the checksum evaluation is close to that of the order of computations for the actual algorithm. Therefore, at small matrix sizes, the overhead is high, however, as the data-set expands, the overhead becomes smaller as the difference in the orders becomes more significant. However, ABFT is not as convincing here as in earlier cases.

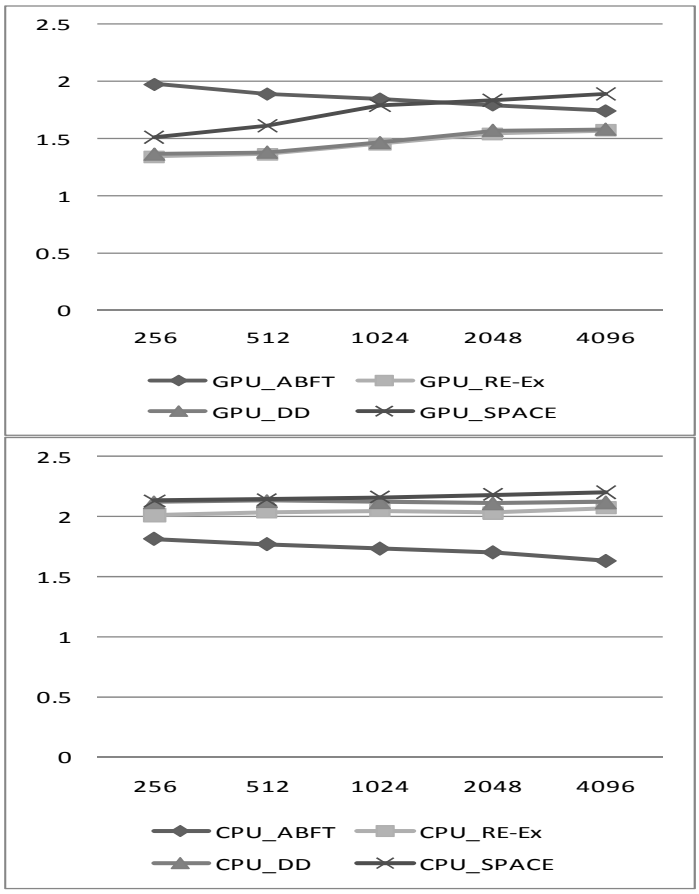


Fig. 5: Overhead comparison for SAXPY

A similar effect can also be seen from the GPU figures, with the initial overhead with ABFT being almost 2X (100%) and then coming down to 1.7X as the data set size increases. The drop in the overhead as the matrix size increases is not as great as the ones in the earlier kernels. This is mainly due to the fact that the creation of extra threads cannot negate the very fact that the orders of computations in the actual kernel and in the checksum algorithm are comparable.

iv. Space Redundancy

The figures for Space Redundancy are very similar to those of the earlier kernels and there are no surprises in store here. This is a reflection of the similar natures of matrix multiplication and SAXPY.

D. FFT

FFT has always been a kernel that has drawn a lot of attention: be it in terms of efficient and fast hardware implementations or even the parallelization potential of the basic computation. It is also a good example of a typical engineering or scientific

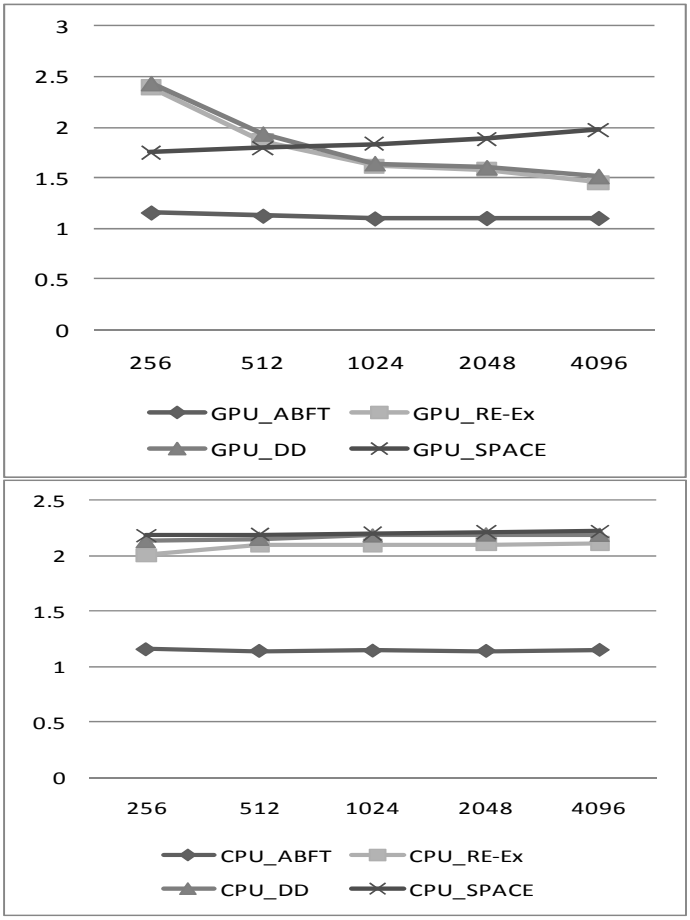


Fig. 6: Overhead comparison for FFT.

computation and is representative of the nature of work that one can see in compute intense environments.

i. Re-Execution

The re-execution trends for the CPU are unchanged from the previous algorithms. However, the trends for the GPU are surprising. The GPU shows a 2.5X normalized execution time.

This is mainly to do with the nature of re-execution. In re-execution the entire algorithm has to be executed twice. The FFT algorithm can be basically split up into 2 parts: creating the co-efficient matrix and performing the actual FFT computation. The GPU is not the thing to choose if there are large scale accesses to memory involved and that is precisely the reason for its dismal performance. While re-executing the code, the GPU has to fill the co-efficient matrix 2 times. This means that a large block of memory has to allocated for this matrix and that the threads have to store data into this large matrix. The size of the matrix itself demands that there have to be global memory accesses. And, the GPU execution is penalized severely for every global memory accesses, especially if the number of threads present is insufficient to

hide the memory accesses latency. This is precisely what happens at small matrix sizes.

As the matrix size increase, however, the number of threads available for scheduling increases and therefore enables the GPU to tolerate memory latency better. Hence, the GPU performance increases as the number of point for FFT increases, thereby increasing the number of threads in-flight for every computation.

ii. Data Diversity

As always, the data diversity numbers follow the numbers for re-execution to a great extent.

iii. ABFT

The ABFT algorithm for FFT is unique. The order of computations required for producing the checksum is the same as the order of computations required for the actual FFT algorithm. Therefore, if a single signal is to be transformed, the overhead will be closer to 100%.

However, FFT is not usually an algorithm that is used on a single input data-set. In fact, one of the most common applications of FFT is transforming streaming data. With that in mind, it is not impractical to consider that the number of n-point (n = number of elements in the input) signals available is to the order of 10. Hence we have assumed that the FFT is computed on ten signals in succession. This means that the overhead of the ABFT checksum computation is distributed over 10 runs and hence falls down to around 1.1X or 10%. Here again, the CPU performs better than the GPU but not by much. In fact, the presence of ten input signals creates an abundance of threads for the GPU to use in its various tricks to hide memory latency. Hence, ABFT on GPU in this case works better than for any other kernel.

iv. Space Redundancy

The space redundancy figures are in tune with the other applications that have been looked at earlier.

VII. Insights on Implementability

Having looked at the various performance overheads of the SIHFT techniques, it makes sense to look at some things that are not glaringly evident from the above numbers and discussion. In this section, we will talk about those in some detail.

A. Flexibility of Implementation

We had claimed that SIHFT techniques allow for implementation flexibility unlike hardware based techniques. To make that picture clearer, we present a small study of the overheads in a flexible implementation. We have assumed that Re-execution, Data Diversity and Space Redundancy have overheads close to 100% (2X normalized execution time). ABFT is assumed to have a 40% (1.4X) execution overhead. These values are not too far from the actual (pessimistically at least) and therefore should provide as good indicators of some typical implementations.

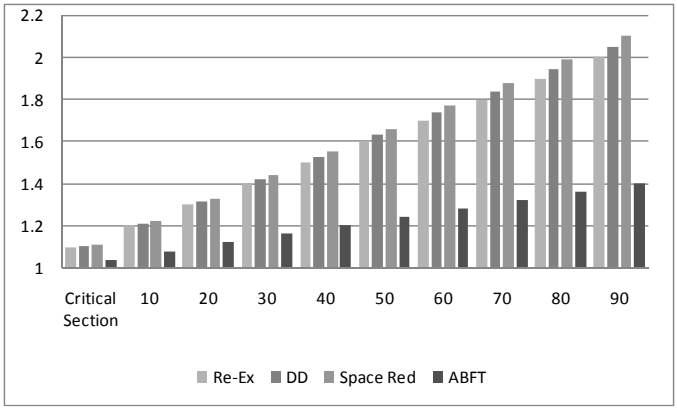


Fig. 7: Demonstration of flexible execution for various critical sections.

This analysis may seem intuitive to some, but without it, the flexibility of implementation cannot be represented effectively. The fig X. shows the numbers for our implementations. The critical section i.e. the section of the code that has to be protected using SIHFT techniques is varied from 10% to 100%. This would usually be a function of the nature of the application being used.

The figure shows that depending on the length of the critical section, the overhead changes. The normalized execution time can be as low as 1.04X of baseline (ABFT for 10% critical section) to 2.15X (Space Redundancy for 100% critical section). The above figure gives a good idea of the performance overhead that can be expected in a flexible implementation.

B. ABFT: Pros and Cons

As can be seen on most of the results above, ABFT looks like the best technique for fault tolerance. It has low overhead for most types of computations. However, what is not so evident is the challenge of implementing ABFT.

ABFT implementation needs a thorough understanding of the algorithm under study. The behavior of the data elements and how the checksum computation is affected by them needs to be closely monitored. In all the above cases, ABFT has been the most challenging algorithm to implement. Thus, the low runtime overhead is also coupled with a high overhead for implementation. It is a classical trade-off, in the sense that if the resources are available to study and implement it, it is surely a hands-down winner at runtime.

Another subtlety in the implementation of ABFT is its applicability to non-matrix applications. Much work needs to be done to prove that it is indeed widely applicable irrespective of the underlying algorithm.

At the same time, if hardware support for checksum computation on the working sets of a program is incorporated, it will not only bring down the initial implementation and programming overhead, but also improve the performance of ABFT further.

VIII. Conclusion

GP-GPUs provide an attractive platform for scientific applications. Commodity GPUs do not provide built-in support for hardware fault tolerance, however many SIHFT

techniques can be applied to GPUs. We have shown that several of these techniques incur much smaller performance penalties on a GPU compared to a CPU as long as the physical limits of the GPU are not exceeded. In particular, re-execution is well-suited to the GPU, since computational power is plentiful and adding extra computations is inexpensive since global memory latency often dominates execution time. Space redundancy also works well, provided the application size is small. With the exception of control-heavy techniques like ABFT, the overhead of software-based fault tolerance is much lower than on CPUs Therefore, we believe that most of the software-based fault tolerance techniques devised for CPUs can be efficiently employed on GPUs with a much lower performance penalty.

An important observation from the research is also the difference in the predictability of performance of CPUs and GPUs. This is extremely important from a programmer's perspective. As the size of the data set increases, the CPU performance remains very predictable, while the GPU performance improves or deteriorates depending on the effect on its memory management. This may sound like a challenge to a few programmers, but on a whole, any sort of performance unpredictably, especially to the degree shown by the GPU is generally not welcome.

We have also seen that the SIHFT techniques deliver on the promise of allowing flexibility of implementation and are totally feasible with zero hardware modification. These attributes will certainly mean that they are front-runners when it comes to handling processor fault tolerance efficiently.

Going forward, it will be interesting to see the approaches that processor makers take to implement high-speed processors on a relatively unreliable fabric. GPU-type architectures promise much better performance on SIHFT techniques. CPU architectures meanwhile have the traditional heavy weight of backward compatibility in their favor. They also tend to give much more predictable performance while at the same time being better at control oriented SIHFT techniques (as expected). Whether it is the GPU-like parallel architectures that compel programmers to know their target hardware intricately or the expanded multi-core versions of CPUs that promise compatibility and predictability at the cost of outright performance that will gain further traction in the future is an open question. In either case, with the realistic threat of unreliable transistors in the near future, we believe that the importance of using high-level and cost-effective SIHFT for fault tolerance will grow.

IX. Acknowledgements

I would like to thank the continuing guidance of my advisor, Prof. Kewal K. Saluja who gave a lot of suggestions and direction to this work. I acknowledge the contribution and assistance of Anthony Gregerson for giving some initial momentum and providing the required GPU. This work was in part supported by the National Science Foundation under the grant CPA-0811467.

X. References

[1] Gordon E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, April 1965.

[2] "International Technology Roadmap for Semiconductors," 2009. http://www.itrs.net/

[3] T. Austin et. al, "Reliable Systems on Unreliable Fabrics," in *IEEE Design and Test*, vol. 25, 2008, pp. 322-332.

[4] NVIDIA. (2009) CUDA Zone. [Online], http://www.nvidia.com/object/cuda_home.html

[5] NVIDIA. (2008) CUDA Programming Guide 2.1. [Online]. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf

[6] N. Oh, S Mitra, and E.J. McCluskey, "ED4I: Error Detection by Diverse Data and Duplication Instructions," in *IEEE Transactions on Computers*, vol. 51, 2002, pp. 180-199.

[7] E. Rotenberg, "AR-SMT: A Michroarchitecture Approach to Fault Tolerance in Microprocessors," in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 1999, pp. 84-91.

[8] F. Rashid, K.K. Saluja, and P. Ramanathan, "Fault Tolerance through Re-execution in Multiscalar Architectures," in *Proceedings of the 2000 International Conference on Dependable Systems and Networks* , 2000, pp. 482-491.

[9] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processor- A Survey," in *IEEE Transcations on Computers*, vol. C-37, 1988, pp. 160-174.

[10] Todd M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," , 1999, pp. 196-207.

[11] E. Matten, N. Jayasena, T.J. Knight, and W.J. Dally, "Fault Tolerance Techniques for the Merrimac Streaming Supercomputer," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005, p. 29.

[12] J.D. Foley, A. Van Dam, and J.F. Hughes, *Computer Graphics : Principles and Practice*.: Addison-Wesley Professional, 1995.

[13] H., Arora, H. Sharangpani, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, pp. 24-43, 2000.

[14] Ian Buck and et. al, "Brook for GPUs," in *ACM Transactions on Graphics*, 2004.

[15] L. Seiler, et. al "Larrabee: A Many-core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, vol. 27, no. 3, August 2008.

[16] C. I. Rodrigues, et.al, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73-82.

[17] Israel Koren and C. Mani Krishna, *Fault-Tolerant Systems*: Elsevier, 2007.

[18] R. Guerraoui and A. Schiper, "Software-Based Replication for Fault Tolerance," in *Computer*, vol. 30, 1997, pp. 68-74.

[19] S. Rangarajan, S. Setia, and S. K. Tripathi, "A Fault Tolerant Algorithm for Replicated Data Management," , vol. 6, 1995, pp. 1271-1282.

[20] K.H. Huang and J.A. Abraham, "Algorithm- Based Fault Tolerance for Matrix Operations," in *IEEE Transactions on Computers*, vol. 33, 1984, pp. 518-528.

[21] Scott Wasson, "NVIDIA's GeForce 8800 graphics processor," *Tech Report*, November 2007.

[22] NVIDIA. NVIDIA 8800GT Specifications. [Online]. http://www.nvidia.com/object/product_geforce_8800_gt_us.html

[23] Intel. Intel® Core™2 Duo Mobile Processor T5750. [Online]. http://processorfinder.intel.com/details.aspx?sSpec=SLA4D

[24] J.J. Dongarra and Z. Chen, "Algorithm Based Checkpoint-Free Fault Tolerance for Parallel Matrix Operations on Volatile Resources," in *Parallel and Distributed Systems Symposium*, 2006.

[25] S. Wang and N.K. Jha, "Algorithm-based fault tolerance for FFT networks", in *IEEE Transactions on Computers*, vol. 43, 1994, pp. 849-854.

[26] D.P. Siewiorek,, " Fault Tolerance in Commercial Computers", in *IEEE Transactions on Computers"*, vol. 23,1990, pp. 26-37.