Graphics Hardware (2007)
Timo Aila and Mark Segal (Editors)

A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors

Jeremy W. Sheaffer[1] David P. Luebke[2] Kevin Skadron[1]
[1]University of Virginia [2]NVIDIA Research

Abstract

General purpose computation on graphics processors (GPGPU) has rapidly evolved since the introduction of commodity programmable graphics hardware. With the appearance of GPGPU computation-oriented APIs such as AMD's Close to the Metal *(CTM) and NVIDIA's* Compute Unified Device Architecture *(CUDA), we begin to see GPU vendors putting financial stakes into this non-graphics, one-time niche market. Major supercomputing installations are building GPGPU clusters to take advantage of massively parallel floating point capabilities, and Folding@Home has even released a GPU port of its protein folding distributed computation client. But in order for GPGPU to truly become important to the supercomputing community, vendors will have to address the heretofore unimportant reliability concerns of graphics processors. We present a hardware redundancy-based approach to reliability for general purpose computation on GPUs that requires minimal change to existing GPU architectures. Upon detecting an error, the system invokes an automatic recovery mechanism that only recomputes erroneous results. Our results show that our technique imposes less than a* 1.5× *performance penalty and saves energy for GPGPU but is completely transparent to general graphics and does not affect the performance of the games that drive the market.*

1. Introduction

Exponential device scaling has produced incredible advances in the capabilities of today's computing infrastructure. Graphics processors have taken advantage of these scaling trends to achieve dramatic increases in throughput. Semiconductor devices, however, have now become so small that they are vulnerable to transient faults caused by cosmic and terrestrial radiation; and to noise due to crosstalk, di/dt induced voltage droop, and parameter variations. As the importance of these phenomena all grow exponentially with decreased feature size or supply voltage [SABR04], the 'free lunch' of Moore's Law for graphics architects approaches its end. Future designs must be more aware of such low-level physical challenges.

1.1. The Case for Redundant GPUs

Graphics processors provide cheap, commodity access to floating point throughput that until recently was only available in supercomputers. New GPGPU APIs like CUDA and CTM ease the retargeting of traditional supercomputing applications, like cosmological n-body simulations and nuclear testing, to GPUs. Increased programmability will lead to adoption, and in turn to the development of new supercomputing infrastructure built on GPU technology at much higher performance per dollar than can be achieved with current systems.

NVIDIA's GeForce 8800 GTX GPU produces a theoretical maximum of 346 GFLOPS, almost double the throughput of the fastest supercomputer in the world only a little over a decade ago [TOP94]. With this kind of processing power, even lower budget organizations are starting to look toward GPUs as a platform for highly parallel, compute intensive, vertical market applications with the aim of moving batch processing online and away from more costly, cluster based solutions. One example of a new market where GPUs are starting to make inroads is that of radiology, where GPUs are employed for medical image processing. This is a domain in which errors are often very costly, both in financial terms and liability, and potentially also in human life. Erroneous computation can lead to death.

ECC in memory systems for these critical applications is a necessity, but it is not sufficient. An error can occur in con-

GH 2007, San Diego, California, August 04 - 05, 2007
© 2007 ACM 978-1-59593-625-7/07/0008 $ 5.00

trol or logic that silently and undetectably corrupts computation outside of the auspices of any memory protections. Transient errors in logic are not yet prevalent, but rates are increasing exponentially with each generation and logic errors are expected to become a significant concern in practice within the next three to five years [SABR04].

To achieve reliability, logical structures must be protected with redundancy. Possible ways to build a redundantly reliable GPU-based system include replicating the entire computation (temporal redundancy) or using two GPUs (spatial redundancy) and comparing the result. Both of these involve a 2× overhead, either in time or space respectively, plus comparison time, in the expected, no-error case. Architectural solutions place the redundancy on-chip, but must answer more complicated questions about what hardware will fall within or without the *sphere of replication*, and how to ensure that the likelihood of a silent data corruption is minimized.

In this paper we explore the concept of the *sphere of replication*—the set of hardware which must be copied or replicated for parallel, redundant execution in a redundantly reliable processor—as it applies to general purpose and scientific computation on graphics hardware. We discuss the array of possibilities for the sphere of replication and choose one solution which uses the existing parallelism of the fragment shader cores as redundantly parallel processors. This paper shows how a commodity GPU architecture can be modified in a few simple ways to create a dual purpose GPU that loses no performance in graphics while providing redundancy for reliable GPGPU. With dual issue of each fragment from the rasterizer, our solution takes advantage of high temporal locality of identical fragments, pushing up texture cache hit rates and allowing computation to complete usually with less than 1.5× overhead, all while saving energy when compared with a naïve reliable system. Furthermore, we provide a simple mechanism for reissue of detected errors that does not impact the critical path bandwidth of the system.

1.2. Overview

A transient, single bit corruption in a microelectronic circuit is termed a *soft error*. Soft errors have long been an important design constraint in general purpose processor design, especially in engineering reliable memory systems for enterprise servers. The common wisdom is that this problem is a non-issue for graphics processors, and in previous work we showed that this common wisdom usually, but not always, holds for GPUs in general graphics applications [SLS06]; however, soft error rates are projected to continue their current trend of increasing at a rate of about 8% per technology generation [HKM*03]—making soft error rates at 16-nm nearly 100 times that of the 180-nm generation [Bor05]—and they are already causing noticeable problems and real concerns for GPGPU developers. The Folding@Home GPU client has now been in distribution for 6 months, running on approximately 500 GPUs. Over this sample set, Folding@Home has shown a failure rate of approximately 1% [Hou07]. This number may seem high; however, note that Folding@Home users are competing to complete work units, thus many overclock their GPUs, which certainly impacts reliability.

A soft error is distinguished from a *hard error* by its transient nature—a soft error is random, temporary, and unpredictable. Soft errors are referred to by several names, including *transient fault, transient error*, and *single event upset* (SEU). While these are often used interchangeably, there are subtle differences in meaning. ‘Soft error’ and ‘SEU’ have classically referred only to radiation-induced transient faults, which are not of great concern with respect to logic. ‘Transient fault’ and ‘transient error’ are more general terms that include soft errors.

Not all errors are cause for concern. If errors do not matter for *architecturally correct execution* (ACE)—in other words, if they do not affect the final outcome of a computation—they are harmless. An error might be harmless, for example, if it strikes a storage location that is not currently in use (i.e., not ACE). The greatest worry of a user who desires reliability is of a *silent data corruption*, or an error that corrupts a result but gives no indication that anything is amiss.

The dominant metric for quantifying the chance of an error as a result of a transient fault is the *Architectural Vulnerability Factor* or AVF [MER05]. The AVF of a structure is a fraction from zero to one which represents the likelihood that a transient fault in that structure will lead to a computational error. AVF takes into account the total amount of time that each bit can contribute to a computation, the total number of bits in the structure, and the size of the structure. More formally, Architectural Vulnerability Factor is:

$$\mathrm{AVF} = \frac{\sum_{b \in B} t_b}{|B| \times \Delta t} \tag{1}$$

where B is the set of all bits in the structure, t_b is the total time that bit b is ACE, and Δt is the total time necessary to complete the computation.

In general-purpose computer systems, any ACE bit must be assumed important. Even a single error in a low-order bit in a commercial or scientific computation can invalidate a computation. What makes graphics hardware unusual is that most state on the graphics card—despite being technically ACE—can tolerate some degree of error. In graphics domains, errors only matter if they affect the user’s perceived experience. An error in a single pixel, for example, may not be noticeable even if it changes the color from white to black. Errors in other state may create more visible errors, but if those errors only last a single frame, the harm is minor. These observations led to the development of the *Visual Vulnerability Spectrum* for characterizing architectural vulnerability for graphics hardware under graphics workloads [SLS06].

The Visual Vulnerability Spectrum makes it clear that full protection of the graphics pipeline is neither necessary nor desirable for reliable rendering; however, this observation obviously does not apply to graphics hardware used in non-visual applications, like GPGPU, where CPU error metrics *are* applicable. Games are still the driving economic force behind GPU development, but as the introduction of GPGPU APIs like CTM and CUDA indicate and as is evident by the work at various National Labs and other supercomputing facilities to build supercomputers of commodity graphics hardware, these GPGPU applications are now of sufficient commercial value that it is worth thinking about what can be done to make GPUs more reliable for scientific computation. Early adopters, like Folding@Home, are demonstrating sufficiently high error rates that it may deter followers until reliability can be assured.

2. CPU Reliability

CPU redundancy literature primarily focuses on minimizing the computation necessary for reliability with techniques like *Redundant Multithreading* (*RMT*), where a primary thread of computation fully computes a result and a secondary thread recomputes as little as possible while maintaining reliability before both results are sent to a comparator. Recovery from detected logical errors is largely untouched in the literature, with some notable exceptions [GSVP03, VPC02]. The concept of memoizing and caching instructions and results so that repeated computation can be bypassed is an important technique in redundancy because it actually reduces the number of instructions executed [PGS04], while more typical solutions do not.

The ways in which computational redundancy is used to protect logic vary over a wide spectrum. At one end lies triplicate replication in *Triple Modular Redundancy*, which includes both redundancy and recovery in one package. In a triple modularly redundant solution, all components are replicated in triplicate, and a comparator accepts votes from each module. This technique relies on the fact that there is a very low probability of two or more of the computational units succumbing to an error in the same calculation (by way of a *simultaneous two-bit error*). Triple modular redundancy is not unique in this reliance—in fact, all reliability techniques rely on a low incidence of simultaneous two-bit errors—it is simply more obvious about it.

Software solutions involve software creation of redundant threads and use a software comparator. A major advantage of a software only solution is the ease of comparison and recomputation. Unfortunately, such solutions place the burden of error detection and correction on the programmer. This burden is further compounded by the fact that threads with side effects are not so straightforwardly handled.

Hybrid hardware/software solutions use replicated hardware for redundant computation with a software comparator. This offers the financial advantage of not requiring any specialized hardware and imposing a very small overhead, but still places heavy burdens on the programmer [RCV*05].

Fully hardware solutions focus their efforts on reducing the temporal and spatial overhead of error detection. Chip multiprocessors (CMPs) bring the idea of a *Chip-level Redundantly Threaded Processor* or *CRT*, which provides some minimal hardware support for redundant multithreading on a CMP. A slightly more sophisticated approach is found in *Simultaneously and Redundantly Threaded Processors* or *SRT*s [RM00, MKR02], which take advantage of hardware multithreading, but not of multiple cores.

There are several reasons why memoization [PGS04], or caching multiply computed results for reuse in a reliable store, is not suitable for application to graphics processors. The most important of these is that it depends upon the existence of complex decode logic to cover the latency of a cache access. This logic simply is neither present nor desirable in current GPU architectures. Furthermore, the technique is engineered for temporally redundant computation, but as is presented in Section 3, a reliable GPU solution is far more likely to use spatial redundancy rather than temporal redundancy.

3. A Redundant GPU Architecture

The design of any reliable architecture requires the analysis of several key design tradeoffs. Among these are:

- The employment of temporal or spatial redundancy in the solution
- The size of the sphere of replication (in a spatially redundant solution)
- The location of the comparison mechanism
- The datapath to signal an error or reissue a computation

In this section we explore these tradeoffs in more detail

3.1. Design Decisions

Implementing a reliable functional unit is not as straightforward as simply replicating computational structure. Figure 1 depicts a an example implementation of a reliable ALU. The sphere of replication for this design fully encompasses two ALUs and partially contains the operand latch and comparator. The operand latch will protect the inputs with ECC, allowing correction of single bit errors and providing a high level of reliability for the inputs should the computation require a reissue. Within the sphere of replication, all datapaths and logic hardware are fully redundant.

The comparator itself is necessarily outside of the sphere of replication. To place it within would require that it be replicated, then the results of the replicated comparators

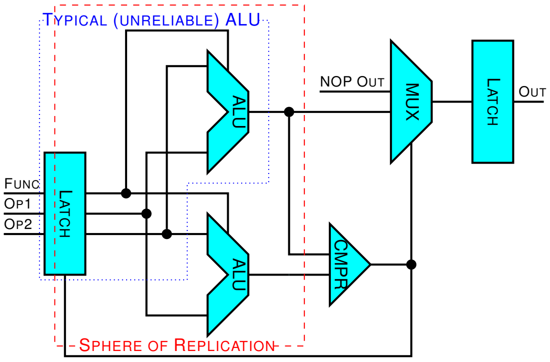

Figure 1: *A redundant ALU uses two parallel ALUs to calculate a result and a comparator to verify that both ALUs have come up with the same answer. The comparator result controls both the input latch and a multiplexor which chooses either the ALU result or a no-op result to be latched in the output.*

would require a third comparator which would need to be either outside of the sphere of replication or itself replicated... The comparator output controls the input latch, either allowing new inputs to enter the ALU or causing the reissue of a failed computation. The comparator output also controls a multiplexor to select the appropriate output of the ALU: either the result of the computation or a NOP result.

The architect of a reliable system must make efforts to protect those logic units and data paths outside of the sphere of replication. For instance, since the comparator unit and its output data paths all fall outside of the set of replicated structures, they are significantly more vulnerable than the redundantly protected state and logic. Architectural and circuit level techniques, such as hardened transistors and careful wire routing, can be employed to increase their reliability.

Figure 1 is in some ways a gross simplification. For instance, it creates no backpressure to correctly handle new inputs. The complexity in even this “simple” example serves well to illustrate both the non-triviality of correctly implementing a redundant functional unit and the space overhead requirements of introducing new hardware into the pipeline (indeed signal propagation delay increases, leading to a possible reduction in clock frequency, due to the new hardware being placed in the critical path, are another concern).

3.1.1. Shader Redundancy

We implement a design for a redundantly reliable graphics processor with the explicit intent that this reliability is intended for GPGPU domains. We have previously shown that the type and level of reliability described in this paper is not necessary for general graphics [SLS06]. Specifically, we provide a redundancy mechanism for the fragment engine, other stages of the pipeline having small AVFs and therefore being of little import to GPGPU applications. In addition to providing a redundancy mechanism, we work under the additional constraint that a solution should require a minimal set of changes to existing hardware—we seek a solution that is zero-cost in terms of the performance overhead when processing graphics workloads and nearly zero-cost with respect to die space sacrificed to implement the solution—this means using existing logic and data paths whenever possible.

While the solution presented in this paper is specifically aimed at redundancy in the fragment engine, it is straightforward to see how the ideas apply more generally to all programmable stages in a unified shader architecture. The extensions necessary to provide redundancy to vertex and geometry shaders follow closely—though GPGPU APIs do not make use of these units—as do the concepts that apply to general multicore architectures.

Redundancy can be implemented over a range of structures, each involving different tradeoffs. Most of these have important consequences that conflict with our goal of modifying the existing architecture minimally. We choose to implement our redundancy at the level the fragment shader core; that is, data are replicated coming in to the fragment array so that each element is computed twice. This solution takes advantage of the already highly parallel shader cores to implement execution of redundant computation without requiring any new computational logic. A mechanism is necessary to replicate the incoming fragments, whether this is dual issue of individual fragments or replication of the actual fragments in the fragment queue. We choose to replicate coming out of the rasterizer, again because this minimizes the scope of the change. If shader cores are paired in a spatially coherent way, it is plausible that core pairs could be made to run in lockstep and share level-1 cache, minimizing off-chip memory bandwidth requirements. Our implementation makes no efforts to ensure either spatial or temporal locality; however, we show very promising cache and memory behavior in our results. Depending on implementation details of the fragment array, it may be necessary to enforce a policy governing allocation of data to shader cores.

Redundancy could be implemented at a finer granularity; for instance, at the level of an ALU. Implementing redundancy at the ALU level means replicating logical structure within a shader unit. Not only would this constitute significant change within each shader core, but unless scheduling opportunities are created to effectively handle this computational bandwidth for general graphics shaders, it effectively halves the shader throughput of the processor for general graphics. Furthermore, as demonstrated in Figure 1, such fine-grained redundancy is expensive in terms of hardware overhead.

Quads (2×2 arrays of fragments) and *Warps* (NVIDIA’s term for a minimum set of threads for SIMD execution in CUDA) both represent minimum SIMD execution blocks in their respective domains. Implementing redundancy over quads or warps is essentially equivalent to shader core redundancy, the primary difference being the size of the replicated computation blocks, i.e. pairs of quads or warps work in parallel on replicated data instead of pairs of cores, complicating the dual issue semantics and restricting its flexibility. Again, a lockstepping and cache sharing mechanism might be plausible here, but seem less likely than at the shader core, as well as highly dependent on current hardware organization.

A software solution might replicate the entire computation on the GPU by issuing it twice from the CPU. This requires either active involvement of the programmer to explicitly write redundant code with a CPU side comparison, or a compiler that generates code to perform a calculation twice and compare the results in software. Clever solutions can do this entirely on the GPU, for example, rendering to a texture on the first pass, then comparing to the stored texture values on the second. Both computational and memory bandwidth overhead are greater than 2× here, but this has the advantage that it requires no hardware modifications. The reliability of the communications network does come into question, though, as a validated result still has a long way to go before it hits memory, and additional error protection hardware may be necessary to protect validated computed results in their registers and as they are communicated from the shader core to memory.

Yet another solution involves dual GPUs. There are several viable ways that one might use multiple GPUs to implement redundancy, with or without hardware support. An on-board solution places two or more GPUs on one board. A comparator verifies the result of the two computations by checking the contents of the output buffers in VRAM before allowing the computation to be uploaded to the CPU. Multi-board solutions would need to take advantage of a CPU-side comparator, *SLI* or *CrossFire* based solutions, or solutions that compare DVI output. This assumes driver support for the multi-board GPU redundancy.

3.1.2. Comparators

The choice of the location of the comparator is highly dependent on the choice of redundancy as discussed above. In general, design choices should be based on the typical case, not the exception. For this reason, it is inadvisable to include, as an example, a comparator per shader unit. This would only eat into (possibly unavailable) timing slack on the critical path of the shader computation.

3.1.3. Datapath

With error detection in place, a mechanism to reissue erroneous computation must accompany it. This requires a datapath to transport the inputs back to the beginning of the compute stage. There are two suitable datapaths, applicable with little to no modification (save signal decoding), already extant on current or future AMD hardware. The first is the datapath associated with the F-buffer [MP01]. The second is the path necessary to implement a unified shader model architecture, which is already present in NVIDIA's G80 GPU, AMD's R600, and the AMD GPU in the XBox 360.

The F-buffer was originally devised as a mechanism to easily enable multi-pass shader calculations when shader complexity was crucially limited by the hardware. AMD implemented the F-buffer in the R5XX hardware family [HPS05], though this hardware was never exposed. It is not publicly known at this writing whether NVIDIA has ever implemented an F-buffer, nor whether AMD retains the structure in R600. After each pass of a multi-pass shader on an architecture with an F-buffer, data is written to the F-buffer. At the beginning of the subsequent pass, data is pulled from the F-buffer in rasterization order to be used in the continuation of the computation. The complexities of the F-buffer, including the ordering requirements, are not necessary for the proposed work, only the datapath. Using the F-buffer datapath, a path from VRAM to the top of the fragment pipeline, fragments that are found to be erroneous by the comparator can be returned to the fragment processor for reissue.

A unified shader model architecture must have a datapath from the back to the front of the unified shader pipeline. This is necessary to carry transformed vertices to geometry processing input and to move processed geometry to be rasterized. This suggests, in fact, that there are two such datapaths, since the rasterizer is still specialized hardware. It is less likely that these datapaths provide the necessary functionality, but the specifics depend on the location of the comparator.

Other clever algorithms, *Delay Streams* [AMN03] being a prime example, have proposed and could capitalize on a similar datapath.

3.2. A Redundant Solution

Figure 2 depicts a block diagram of one possible solution that fits within the constraints defined above. This architecture makes use of the rasterizer as the first stage in the reliable pipeline. The rasterizer issues two copies of each fragment into the fragment queue for processing by the fragment cores. The fragment processor processes all fragments normally and passes them on to the raster operations unit (ROP). ROP will write data to the framebuffer normally if the existing data at that location is in the cleared state—i.e. there is no data in the framebuffer at that location—but will compare all of the calculated results (color, depth, etc.) if data is already present in the framebuffer. When any difference is found between two calculated results that map to the same framebuffer location, an error is signaled.

Reissue depends on a new structure we call the *domain buffer* and minor augmentation of the rasterizer. The domain

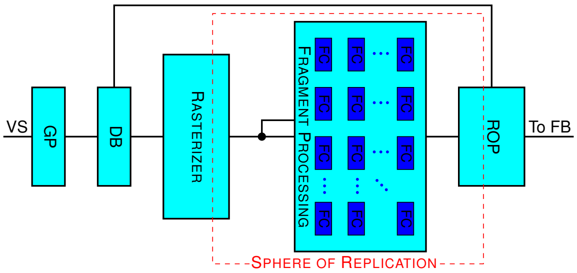

Figure 2: *Our proposed reliable GPGPU system. We add a* domain buffer *to store data needed to set up the rasterizer for the domain in the event of a reissue, augment the rasterizer to produce two of each fragment and to handle reissue requests from the domain buffer, and repurpose raster operations as our comparator. In the figure, "VS" is the vertex stream, "GP" is the geometry processor, "DB" is the domain buffer, "FC" is a fragment core, and "FB" is the framebuffer.*

buffer sits between geometry processing and the rasterizer. It stores the information necessary to set up the rasterizer to produce fragments over the computational domain; for typical GPGPU applications, this is only two triangles, so the domain buffer can be very small When ROP sends a reissue signal, containing the fragment $\langle x, y\rangle$ for the offending datapoint, the domain buffer reissues the geometry and the error location(s) to the rasterizer. The rasterizer applies a stencil to reissue only those fragments which need to be recomputed. In the case of NVIDIA CUDA, a direct compute environment which does not use the rasterizer for thread creation, the *Thread Execution Manager* [NVI07] must be modified to reissue an appropriate subset of the compute domain. We leave issues related to CUDA and direct compute to future work.

Recall Equation 1: AVF $= \frac{\sum_{b\in B} t_b}{|B|\times\Delta t}$. In a typical GPGPU computation, the early stages of the pipeline (primarily vertex and geometry processing) are very short (four vertices and two triangles, respectively). As a result, each respective t_b is very small, leading to a small AVF for the structures, thus they can be ignored without unduly impacting the reliability of the solution. In the case of the rasterizer, however, much of the state is potentially ACE for a significant portion of the application. For this reason, rasterizer state should probably be protected with ECC. We cannot make accurate statements about the importance of protecting the rasterizer without performing an ACE analysis on a simulator with much more faithful rasterizer implementation than we currently have. Nonetheless, we believe that ECC on raster state may be desired. If the rasterizer has a high AVF, it might be necessary to replicate it, move it wholly inside the sphere of replication, and move the double issue to the geometry engine. All off-chip memory and busses must be fully protected with ECC.

3.2.1. Error-Correcting Codes

A redundant multithreading approach such as this protects logic within the sphere of replication. Outside the sphere of replication, memory and communication paths must be protected with ECC; this includes non-replicated caches, video memory, and the PCI-E bus. Furthermore, no solution can guarantee reliability. The goal of any implementation of a reliable system is to reduce the likelihood of a silent data corruption to a negligible value. Precisely what this value is is highly dependent on the application.

4. Experiments and Results

We describe our simulation infrastructure and present results of performance and power studies to evaluate our solution.

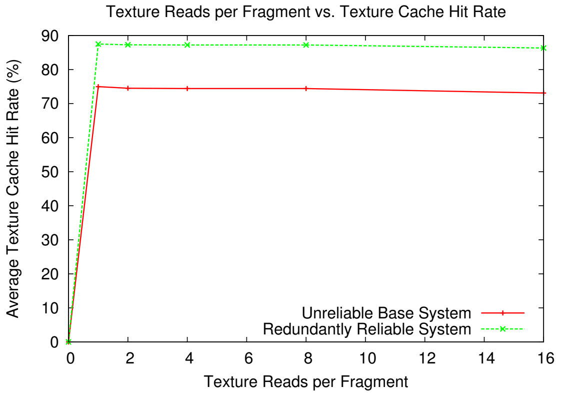

Figure 3: *On the x axis are memory operations per element, with texture cache hit rate on y. Texture cache hit rate is almost universally better in our reliable architecture than in the baseline architecture, the only exception occurring in the* 16×16 *simulations, where the cache did not have sufficient time to warm up. Cache misses are only counted if they lead to a new memory transaction—e.g. misses on an address that will be serviced by a pending memory fetch are not counted—thus the difference in hit rates translates directly to memory bandwidth. These data are averaged over all 16 cores. This graph shows the results over a* 128×128 *domain, though all domains produced qualitatively similar results.*

4.1. Simulation

We implemented the solution depicted in Figure 2 in the GPU simulation infrastructure developed by Bill Mark's group at the University of Texas, Austin [JLBM05]. This system is built in *SystemC* [Sys05], using an event-based model to drive a high-fidelity fragment engine simulation. The fragment engine implements a useful subset of the OpenGL `NV_fragment_program` extension [BK05], including support for branching. Raster operations and the texturing and memory systems are also implemented in some detail, while the rest of the pipeline is purely functional. All fragment program instructions require one cycle to complete, save texturing operations which go through a cache to memory.

The system is engineered with 16 fragment pipelines, each SIMD Float4 with operand swizzling and masking. Each pipeline has 16 thread contexts, which are serviced in round-robin order, with context switches invoked on thread block. Context switches are free as long as there is a runnable thread on the core, else the core must block until a runnable thread is available (a texture reference is serviced or a new thread is instantiated on the core). Each fragment core is connected to memory through a crossbar and cache. Crossbar transactions are processed in two cycles—one cycle in and one cycle out—as long as there is bandwidth available. If there is no available bandwidth, the crossbar queues pending transactions. The queue size is set at 512 elements. The texture caches are 16 kB, 8-way set associative with 64 byte blocks and 256 element queues. Textures are stored in a 128 MB address space, swizzled via a Morton space-filling curve for increased locality [Mor66]. Another crossbar provides a path to raster ops. ROP accesses the framebuffer through an identical interface to that of the fragment processors

We implemented our reliable GPU on top of this infrastructure by making the following changes:

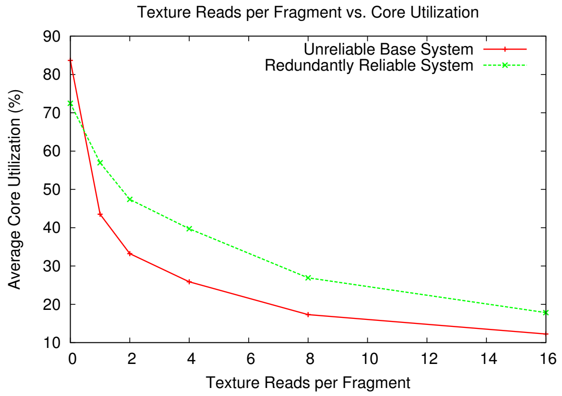

Figure 4: *On the x axis are memory operations per element, with shader core utilization on y. Due to the increased memory locality, core utilization is also better in the reliable architecture. These statistics are averaged over all 16 cores.The graph shows the results over a* 128×128 *domain, though all domains produced qualitatively similar results.*

Double issue from the rasterizer: The rasterizer implementation is functional only. Downstream from the rasterizer is a queue to the fragment engine. The rasterizer is invoked every cycle and runs until this queue is full such that it is never a computational bottleneck. In order to implement double issue we only had to replicate each fragment created by the rasterizer and place them into the input queue for the fragment processors. An actual implementation would require the functionality to turn this feature on for reliable GPGPU operation and off for unreliable GPGPU or general graphics.

Domain buffer: We implement our domain buffer as a lookup-table containing already rasterized fragments. As the rasterizer is not modeled faithfully, this solution creates less room for error while not changing performance characteristics. The domain buffer gets precedence over new geometry going into the rasterizer.

Reissue path: The simulator does not contain any of the datapaths discussed above for repurposing, so we implemented this with a SystemC channel with a one cycle transfer latency between ROP and the rasterizer (logically the domain buffer).

Comparators: The comparators are implemented in ROP. We use full/empty bits on the framebuffer. If the location is empty, we use the standard semantics and set the bit. When the location is full, we issue reads on all channels (in this simulator, that is 3 channels of color, plus depth) and compare all components. These comparisons can be bitwise—it is not necessary to do floating point compares—as any result that is not exactly, bitwise the same as the stored result flags an error. Independent of the result of the comparisons, the data is discarded; however, if it fails, the pixel's framebuffer coordinates are sent over the reissue datapath to be recalculated and the full/empty bit is reset to empty. We do not impose delay to actually do the comparisons but do go through cache to get to stored color data for the comparison, thus accounting for that latency and bandwidth.

This solution as we have presented it so far assumes there is no legitimate reason for any framebuffer location to be written twice. This is largely a valid assumption, though there are notable exceptions, such as the GPU database work by Govindaraju et al [GLW*04]. One way to get around this problem is to modify the full/empty bit semantics such that each write toggles the bit and with it the semantics of the ROP operation. A third attempt to write a location will find it empty and perform normal ROP operations. At the end of the computation, all full/empty bits must be checked. Any bit in the full state indicates an error and that domain element must be reissued. This solution depends on geometry ordering requirements that are enforced by graphics APIs for correct color blending and on precedence of the domain buffer over geometry processing. These requirements are probably propagated into GPGPU APIs as an artifact of the underlying architecture.

This solution also fails to account for scatter operations which write to texture. We can potentially handle these by requiring that all texture writes in the reliable mode go through ROP, though that presents additional issues with respect to ROP bandwidth. This is clearly a problem for direct compute APIs. The details and evaluation are left to future work.

Memory↓	Domain→	16^2	32^2	64^2	128^2	256^2	512^2
0 reads	Real Cache	1.70	1.38	1.38	1.41	1.30	1.21
	Perfect Cache	1.70	1.38	1.38	1.41	1.30	1.21
1 read	Real Cache	1.72	1.33	1.24	1.22	1.25	1.21
	Perfect Cache	1.72	1.41	1.44	1.43	1.49	1.31
2 reads	Real Cache	1.56	1.32	1.31	1.22	1.25	1.30
	Perfect Cache	1.64	1.47	1.50	1.50	1.62	1.65
4 reads	Real Cache	1.75	1.41	1.30	1.22	1.28	1.31
	Perfect Cache	1.73	1.58	1.57	1.62	1.81	2.13
8 reads	Real Cache	1.49	1.32	1.13	1.21	1.20	N/A
	Perfect Cache	1.78	1.65	1.69	1.71	1.88	2.17
16 reads	Real Cache	1.50	1.22	1.20	1.36	1.57	N/A
	Perfect Cache	1.86	1.74	1.76	1.85	2.02	2.22

Table 1: *Summary of performance results. The data points in this table are the normalized ratios of simulation cycle counts in our reliable architecture to the baseline architecture, or in other words, the overhead imposed by our reliability implementation. The top number in each cell is the normalized simulation time with a realistic cache simulation (as described in 4.1). The bottom is the same value when the caches are perfect and zero-latency (all memory references take exactly one cycle). Simulation time made it impossible to collect data for the cells marked "N/A".*

4.2. Experimental Results

We present results of a series of performance and power studies comparing our reliable architecture with the baseline unreliable system.

4.2.1. Performance

Memory Reads	0	1	2	4	8	16
Total Instructions	7	8	10	15	20	34

Table 2: *Texture instructions and total instructions for each stressmark.*

We implemented our redundant system as described above and verified the detection and correction implementations. We then simulated both our system and the unmodified baseline system over a series of workloads constituting the cross of 16 × 16, 32 × 32, 64 × 64, 128 × 128, 256 × 256, and 512 × 512 computational domains and 0, 1, 2, 4, 8, and 16 memory reads per domain element simulating both real and perfect caches. Our programs are designed to stress the memory system, as memory behavior is an important and interesting concern of all GPUs, but especially of one that is designed to be reliable through redundancy. These *stressmarks* issue two memory instructions then immediately use them with a previous result via a `MAD` instruction, thus making as many memory reads as is reasonably possible such that all of the fetched results are actually used. They are worst case benchmarks with respect to memory. Each stressmark completes with a sequence of 7 arithmetic instructions; the same seven that are used in the 0-memory-reference stressmark. Our simulations are designed to evaluate the performance of our solution in the common case where there are no errors, so we do not inject faults. More detail on the programs are listed in Table 2. Results of the 8 and 16 memory reference simulations on the 512 × 512 domain could not be obtained due to simulation time. Table 1 summarizes the results of this suite of experiments.

From the table we can see that in every case, save 16 × 16 with 4 memory references, the relative performance of the reliable architecture to the baseline with a real cache is at least as good as, and often much better than, that of the corresponding perfect cache simulation. Further evidence for this appears in the plot in Figure 3. This suggests that our solution benefits from improved memory locality with dual issue. We do not understand why four instances of perfect cache simulations show overheads greater than two; however, in these simulations, texture operations are equivalent to ALU operations (in terms of timing), so we note that these programs effectively performed an unrealistic number of consecutive ALU operations without a memory reference [HP03] (15, 20, or 34, as per Table 2). Data show that the cores were poorly utilized in these instances. Figure 4 shows average core utilization for both architectures (Utilization is measured by counting the number of cycles that *some shader has the core*, even if this is a "redundant shader", and dividing by the total number of cycles, thus naïvely we would expect the redundant and non-redundant utilization to be about equal). The reliable architecture makes better use of computational resources in almost all cases, except when there is no texture activity.

4.2.2. Power

Memory↓	Domain→	16^2	32^2	64^2	128^2	256^2	512^2
0 reads	Power	1.15	1.40	1.40	1.38	1.50	1.61
	Energy	1.97	1.94	1.94	1.95	1.95	1.95
1 read	Power	1.14	1.44	1.54	1.56	1.54	1.59
	Energy	1.97	1.93	1.92	1.93	1.94	1.94
2 reads	Power	1.24	1.45	1.46	1.56	1.53	1.49
	Energy	1.94	1.92	1.92	1.91	1.93	1.94
4 reads	Power	1.12	1.37	1.47	1.56	1.50	1.47
	Energy	1.96	1.93	1.91	1.91	1.92	1.93
8 reads	Power	1.27	1.43	1.63	1.54	1.56	N/A
	Energy	1.90	1.90	1.85	1.87	1.88	N/A
16 reads	Power	1.27	1.52	1.54	1.37	1.21	N/A
	Energy	1.91	1.87	1.86	1.86	1.90	N/A

Table 3: *Relative power and energy of our redundantly reliable GPU to the baseline, unreliable GPU. The reliable solution draws more power than the baseline system, which is to be expected, considering its increased core utilization. Note that in all cases our reliable solution uses less energy, however, typically by about 10%, than would a software or dual-gpu solution, which would require at least 2× energy.*

We instrumented the simulator's fragment engine with a power model based on *PowerTimer* [BBS*03]. PowerTimer is an IBM power model, based on the Power 4 architecture. We scaled the PowerTimer model to be more in line with a modern GPU, built on a 90 nm process, running at 600

MHz at 1.45V and assumed 25% leakage current. We used HP Labs *CACTI* [TTJ06] cache modeling tool with the cache parameters described in Section 4.1 to create a cache power model. CACTI allows computer architects to quickly and easily analyze cache design tradeoffs with respect to area, cycle and access times, and static and dynamic power. The results of our power analysis appear in Table 3. Our solution requires more power than the baseline, do to its improved core utilization, but we are able to save about 10% more energy (power × time) than a naïve solution could.

5. Discussion

The issues discussed above and our proposed solution combine to suggest an important concept: *In a reliable multicore architecture, the sphere of replication should be as large as possible.*

Traditional out-of-order processors are designed to maximize single thread performance, with much of the processor real-estate dedicated not to computation, but to support structure, like branch predictors, that help the processor to achieve this goal. Early work on reliability on these architectures recognizes that introducing hardware for redundant computation is expensive. By comparison, comparator hardware is cheap. For these reasons, it is desirable to keep the sphere of replication small.

Conversely, in multicore architectures, where the computational cores are small, simple and already available, the comparators become expensive (recall the costs in Figure 1). Similarly, as the tiling of multicore architectures increases, the applications written for them will likely come to resemble fragment shaders more than they will Microsoft Word, where the parallelism is primarily event-based as opposed to the data parallelism in scientific, image processing and other “traditional” GPGPU application domains. With fragment-shader-like programs, recomputing one result is akin to recomputing from a checkpoint in currently proposed solutions but with much lower overhead. Given this, and the fact that the larger the sphere of replication, the smaller the area in which transient errors are undetectable, it follows that the sphere of replication should encompass as much computational hardware as possible.

6. Conclusions and Future Work

We have presented a set of modifications to existing graphics architectures to allow reliable performance in general purpose computation domains. We have made efforts to leverage and repurpose existing hardware features as much as possible, with the addition of a *domain buffer* as the only new hardware, and minor repurposing of the rasterizer and ROP units to achieve our goals. Because this solution does not require any modification of the fragment array, does not impact throughput for general graphics workloads, and requires negligible new logic to implement the functionality, and furthermore because the ideas presented in this paper apply directly to unified shader architectures—our simulation and testing was based on a more traditional graphics pipeline—we believe that it is near optimal and should be implemented in future GPUs. However, these techniques are not so directly applicable to direct compute architectures like G80 with CUDA, in which data go through neither the rasterizer nor ROP. This observation exposes an important place for future work.

Our implementation demonstrates that a reliable GPU built as described in this paper benefits greatly from increased memory locality inherent in the double issue method, allowing it to perform much better than the naïve expected overhead of 2×. In fact, our simulations show a measured overhead of less than 1.5× on most of our problem domains.

We also make an important observation about the sphere of replication with respect to multicore architectures. When designing a reliable processor based on a traditional, single core architecture, engineers aim to make the sphere of replication as small as possible, in order to minimize the extra costs implicit in replicating hardware. However in a multicore environment, it is desirable to make the sphere of replication as large as possible, because the redundant hardware is already present, so in essence its cost is free, while the cost of comparing results is expensive.

We proposed a solution to the problem of reliable write to texture and by association to the more general memory operations in CUDA in Section 4.1. Completing the details of this solution and evaluating it poses a difficult and important problem that must be solved. Detailed ACE analysis of raster state can help to make a more informed decision with respect to the rasterizer’s inclusion in the sphere of replication.

7. Acknowledgments

A portion of this work is built on top the GPU simulation infrastructure developed by Greg Johnson, Chris Burns, Alexander Joly, and William R. Mark at the University of Texas, Austin. Their simulator is still under development, and no papers specifically about the simulator have been published yet, but a limited description appears in their 2005 Transactions of Graphics paper [JLBM05].

This work was supported in part by NSF grants CCF-0429765, CCR-0306404, the Army Research Office under grant no. W911NF-04-1-0288, a research grant from Intel MRL, and an ATI graduate fellowship. We would like to extend out sincere thanks to the anonymous reviewers for their detailed and helpful comments.

One reviewer made the insightful suggestion that the rasterizer stencil hardware could be employed for masking during reissue. Our previous solution was far more complicated in this respect.

References

[AMN03] AILA T., MIETTINEN V., NORDLUND P.: Delay Streams for Graphics Hardware. *ACM Transactions on Graphics 22*, 3 (2003), 792–800.

[BBS*03] BROOKS D., BOSE P., SRINIVASAN V., GSCHWIND M., EMMA P. G., ROSENFIELD M. G.: New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors. *IBM Journal of R & D 47*, 5/6 (2003).

[BK05] BROWN P., KILGARD M. J.: NV_Fragment_-Program, May 2005. http://www.opengl.org/registry/-specs/NV/fragment_program.txt.

[Bor05] BORKAR S.: Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro 25*, 6 (Nov./Dec. 2005), 10–16.

[GLW*04] GOVINDARAJU N. K., LLOYD B., WANG W., LIN M., MANOCHA D.: Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2004), ACM Press, pp. 215–226.

[GSVP03] GOMAA M. A., SCARBROUGH C., VIJAYKUMAR T. N., POMERANZ I.: Transient-Fault Recovery for Chip Multiprocessors. *IEEE Micro 23*, 6 (2003), 76–83.

[HKM*03] HAZUCHA P., KARNIK T., MAIZ J., WALSTRA S., BLOECHEL B., TSCHANZ J., DERMER G., HARELAND S., ARMSTRONG P., BORKAR S.: Neutron Soft Error Rate Measurements in a 90-nm CMOS Process and Scaling Trends in SRAM from 0.25-μm to 90-nm Generation. In *IEEE International Electron Devices Meeting 2003 Technical Digest* (Dec. 2003), IEEE, pp. 523–526.

[Hou07] HOUSTON M.: Personal communication, Mar. 2007. Stanford University and Folding@Home.

[HP03] HENNESSY J. L., PATTERSON D. A.: *Computer architecture: a quantitative approach*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[HPS05] HOUSTON M., PREETHAM A. J., SEGAL M. A.: *A Hardware F-Buffer Implementation*. Tech. rep., Stanford University., 2005.

[JLBM05] JOHNSON G. S., LEE J., BURNS C. A., MARK W. R.: The Irregular Z-buffer: Hardware Acceleration for Irregular Data Structures. *ACM Trans. Graph. 24*, 4 (2005), 1462–1482.

[MER05] MUKHERJEE S. S., EMER J. S., REINHARDT S. K.: The Soft Error Problem: An Architectural Perspective. In *HPCA* (2005), IEEE, IEEE Computer Society, pp. 243–247.

[MKR02] MUKHERJEE S. S., KONTZ M., REINHARDT S. K.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *ISCA* (2002), IEEE, IEEE Computer Society, pp. 99–110.

[Mor66] MORTON G. M.: A computer oriented geodetic data base and a new technique in file sequencing, 1966. IBM Canada.

[MP01] MARK W. R., PROUDFOOT K.: The F-Buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering. In *Proceedings of the SIGGRAPH/Eurographics Graphics Hardware Workshop 2001* (2001).

[NVI07] NVIDIA: NVIDIA CUDA compute unified device architecture programming guide, 2007. http://developer.download.nvidia.com/compute/cuda/08/-NVIDIA_CUDA_Programming_Guide_0.8.pdf.

[PGS04] PARASHAR A., GURUMURTHI S., SIVASUBRAMANIAM A.: A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *ISCA* (2004), IEEE, IEEE Computer Society, pp. 376–386.

[RCV*05] REIS G. A., CHANG J., VACHHARAJANI N., RANGAN R., AUGUST D. I., MUKHERJEE S. S.: Design and Evaluation of Hybrid Fault-Detection Systems. In *ISCA* (2005), pp. 148–159.

[RM00] REINHARDT S. K., MUKHERJEE S. S.: Transient fault detection via simultaneous multithreading. In *ISCA* (2000), pp. 25–36.

[SABR04] SRINIVASAN J., ADVE S. V., BOSE P., RIVERS J. A.: The Impact of Technology Scaling on Lifetime Reliability. In *DSN* (2004), IEEE, IEEE Computer Society.

[SLS06] SHEAFFER J. W., LUEBKE D. P., SKADRON K.: The Visual Vulnerability Spectrum: Characterizing Architectural Vulnerability for Graphics Hardware. In *Proceedings of Graphics Hardware 2006* (Sept. 2006).

[Sys05] SYSTEMC LANGUAGE REFERENCE MANUAL WORKING GROUP: Draft standard SystemC language reference manual (version 2.1), April 2005. http://www.systemc.org/.

[TOP94] TOP500.ORG: June 1994|TOP500 Supercomputing Sites, June 1994. http://www.top500.org/lists/-1994/06.

[TTJ06] TARJAN D., THOZIYOOR S., JOUPPI N. P.: *CACTI 4.0*. Tech. Rep. HPL-2006-86, HP Laboratories Palo Alto, June 2006.

[VPC02] VIJAYKUMAR T. N., POMERANZ I., CHENG K.: Transient-fault recovery using simultaneous multithreading. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 87–98.