Improving GPU Robustness by Making Use of Faulty Parts

Artem Durytskyy, Mohamed Zahran, and Ramesh Karri
ECE Department, Polytechnic Institute of NYU, New York, NY
Email: Durytskyy@yahoo.com, mzahran@acm.org, rkarri@poly.edu

***Abstract*—With hundreds of processing units in current state-of-the-art graphics processing units (GPUs), the probability that one or more processing units fail due to permanent faults, during fabrication or post deployment, increases drastically. In our experiments we found that the loss of a single streaming multiprocessor (SM) in an 8-SM GPU resulted in as much as 16% performance loss. The default method for dealing with faulty SMs is to turn them off. Although faulty SMs cannot be trusted to completely execute a single kernel (program assigned to an SM) correctly, we show that we can still make use of these SMs to improve system throughput by generating and supplying high-level hints to other functional SMs. By making the faulty SMs supply hints to functional SMs, we have been able to achieve an average speed-up of about 16 % over the baseline case (wherein the faulty SMs are turned off). The proposed technique requires minimal hardware overhead and is highly scalable.**

I. Introduction

GPU computing has become widespread with improvement in available GPU programming tools and a reduction in the GPU hardware costs. However, the unreliability of GPU computing has remained unaddressed. For example, several NVIDIA mobile GPUs sold in 2008 failed [1] and NVIDIA GTX 470/480 [2] released in 2010 had low manufacturing yields [3]. A straightforward approach to dealing with faulty hardware is to simply turn it off and continue execution on the remaining functional hardware. However, the faulty hardware may still be useful. Furthermore, our experiments showed that the loss of even a single streaming multiprocessor (SM) resulted in a 16% performance loss for an 8-SM GPU.

A. GPU Architecture

NVIDIA's Tesla architecture revolves around scalable processing elements, which can be added or removed without affecting the underlying structure of a GPU [4]. By varying the number of processing elements, NVIDIA can effectively vary the resulting performance of the GPU.

A typical NVIDIA Tesla GPU consists of several streaming-multiprocessors (SMs). Each SM is responsible for executing a software kernel. The SM consists of several streaming-processors (SPs). We operate at the SM level because SPs within an SM work in a lockstep fashion. Consequently, an SP cannot speed-up another SP within the same SM. In addition, Each SM has two special function units, one double precision unit (DPU), shared memory, and several local caches. Two or more SMs form a Thread Processing Cluster (TPC).

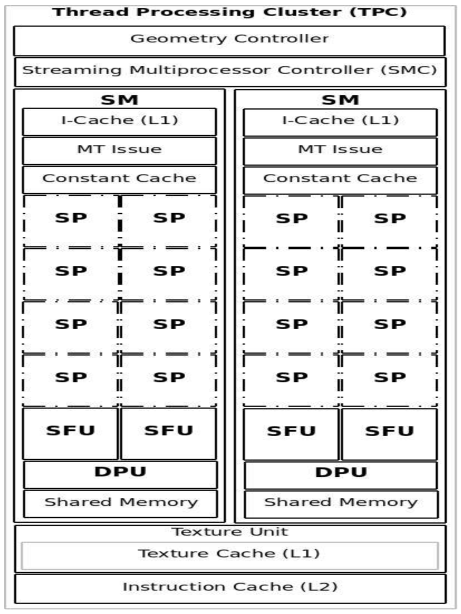

Fig. 1. NVIDIA GeForce 8800 consists of 4 TPCs. The Figure shows a single TPC with two SMs of 8 SPs each.

Figure 1 shows the NVIDIA GeForce 8800 with four TPCs, each consisting of two SMs and each of which in turn is composed of eight 8 SPs.

A thread is an instance of the written GPU program which is responsible for computing one or more result points of the final result returned to the host. An SM is able to execute geometry, vertex, and pixel-fragment programs as well as parallel computing programs. Each SP performs basic operations such as floating-point addition and multiplication. In the target architecture considered in this paper (NVIDIA TESLA), an SM consists of eight SPs, two special function units (SFUs), a multithreaded instruction fetch and issue unit, an instruction cache, a read-only constant cache, and a small amount of read/write shared memory [4].

The SMs employ a Single Instruction Multiple Thread (SIMT) architecture. In this SIMT architecture, each SM manages a pool of independent threads without incurring overhead. Thirty two threads are typically pooled together into a warp and warps are assigned to individual SMs. At each clock cycle, warp manager schedules an available warp for execution and issues the next instruction to each one of the thirty two threads in that warp. The active threads within the warp execute the issued instruction in a lock step fashion. At the next clock cycle, another warp is selected from the pool and issued for execution. If an instruction causes different threads in a warp to follow different execution paths, then each thread is executed serially until all thirty two threads are executed. This slows down the processing and consumes resources.

B. Reliability Issues of GPUs

In a recent study in [5], it was found that among a large sample of systems with dedicated GPU hardware, 2/3 exhibited a detectable, pattern-sensitive, rate of memory or logic soft errors. It was also found that these soft errors in memory or logic correlate strongly with the underlying GPU architecture. These factors make it necessary to design GPU hardware that is not only reliable but robust as well. A GPU architecture that can make use of some faulty components in order to speed-up its own execution is thus the focus of our research.

In this paper, we propose a hardware solution that utilizes hints from one or more faulty SMs to speed-up execution of the programs running on fully functional SMs. We then quantify the performance improvement over a system in which the faulty hardware has simply been turned off. Finally, we show the impact of the proposed approach on GPU-optimized benchmarks to demonstrate that the approach scales with increased program complexity and size.

C. Previous Work

There has already been a significant published body of research regarding detecting hard faults in many and multi-core systems [6] [7] . Current many and multi-core tolerance techniques include disabling faulty hardware components [8], partitioning cores into pipeline stages and sharing these stages between cores by the use of complex interconnection networks [9], or using a checker pipeline to ensure correct execution [10]. Most of these methods are accompanied by either high hardware overhead, performance penalties, or high power consumption. In [11] and [12], a functional CPU core in a traditional multi-core processor has been assisted by a faulty core to speed up the functional core. A 40% speed up over the baseline configuration could be achieved by using I-Cache Hints, D-Cache Hints, and Branch Prediction hints for a multi-core CPU system. Whenever the available CPUs/cores are not-faulty, the hardware associated with the faulty units that collect and distribute hints is turned off. We show that a scheme loosely based on this approach can be applied to a GPU architecture which has a multiplicity of SMs, one or more of which has a permanent fault.

We present our technique in Section II and analyze our results in Section III. Finally, we draw some conclusions in Section IV. Although the GPU architecture and associated terminology are based on NVIDIA GPUs, the developed techniques apply to AMD-ATI architecture as well.

II. Proposed Approach

A. The Main Idea: A Hint

The approach is based on the observation that even though a faulty SM cannot perform an error-free execution of the entire GPU kernel, it can still execute certain segments of the GPU kernel correctly. This is because the fault in the SM may only affect some parts of the GPU kernel execution and not others. We use this capability of a faulty SM to collect *s*peed-up hints by running the same kernel on a faulty SM and on a functional SM. The hints collected by the faulty SM are then passed to the functional SM. A functional SM will use these hints to speed up the execution of the GPU kernels. Some hints may be more effective than others. Those hints whose effectiveness crosses a threshold are used otherwise,they are ignored. So, a hint is a piece of information which can be used to speed up the execution. For example, an instruction cache prefetch is a hint that can reduce instruction cache miss-rate of the functional SM.

In this paper we assume a TPC with 2 SMs. This is common in conventional GPUs [4]. This means, we have at most one faulty SM and at least one functional SM. In case the TPC has more than 2 SMs there could be more than one SM of each type.

B. Which Hints to Use

We studied several type of possible hints. Based on their effectiveness and the hardware requirements we narrowed them down to three types:

- **Hint 1: TPC cache prefetches:** SMs within the same TPC are able to supply hints based on the instruction cache of the faulty SM. According to [13] , NVIDIA GPU architecture contains three levels of instruction caches. The size of per-SM L1 cache is 4 kB and instructions are fetched from L1 cache 64 bytes at a time. The hints delivered include the addresses of the block of instructions to be loaded. Once the functional SM receives this hint, it will use it to warm up its own L1 instruction cache, thus resulting in a cache hit and minimizing the penalty associated with an instruction cache miss in that SM. Because faulty SMs are not allowed to write to memory, they can proceed in their execution faster than the functional SMs. Therefore, faulty SMs can *see* which instructions will be executed by functional SMs and deliver this information in the form of a hint. However, sometimes the faulty SM can go on a wrong path or get its execution delayed, depending on its fault type. So these hints may not always be correct or on-time. The effectiveness of these hints is tracked by keeping a counter associated with the number of cache hits/misses for each of the cache entries brought in using the hints. By resetting this counter to half its maximum value in the beginning of each hint recovery period (more on that later) and then checking its most significant bit (MSB), we can effectively know whether or not the majority of I-Cache hints were effective or not with minimum overhead.
- **Hint 2: inter-SM warp memory coalescing:** Whenever the faulty SM issues a load instruction to fetch a certain region of memory it is possible that the functional SM will try to fetch a similar or nearby region of memory. These load instructions can be coalesced (i.e. merged

so that a big continuous chunk of memory is brought with one access, which saves a lot of time) between the faulty SM and the functional SM. Memory coalescing hardware already exists in TPCs. When the memory request is satisfied by the memory controller, it is sent to both, the faulty and functional SMs. In this case the hint will contain the address ranges of the memory request issued by the faulty SM. Then, by keeping a counter associated with the number of load instructions issued by the functional SM in which the memory regions are within those of the hint and the timestamp of the hint, we can say whether the hints were effective.

- **Hint 3: Instruction cache prefetches:** When the faulty SM initiates a load instruction, the memory controller will take care of the memory request and will bring the required memory portion into L2 cache which is shared between the faulty SM and the functional SM. When the instruction is issued by the functional SM, the required data will have already been preloaded into the cache, and thus be retrieved from the cache instead from device memory. This hint type is similar to the first hint type with the exception that this one deals with the shared L2 cache, while the first one deals with L1 cache private to each SM. The effectiveness of this hint type can be tracked in a similar manner to the first hint type.

The hints generated by the faulty SM will be correct and hence useful if the fault in the SM has not affected the hint. If the fault affects the hint generating processes, then a wrong hint is generated which in turn might affect the performance of the functional SM. Assuming a uniform fault distribution, the probability of generating a wrong hint increases with the execution time. Hence, the accuracy of hints have to be checked periodically. We call this period as the hint recovery period (HRP).

If the accuracy of the hints are above a threshold, then the hints generated by faulty SM will be used. Otherwise, the hints will not be used. If the HRP is small, then the accuracy is checked often which might penalize performance. However, if the HRP is large, then faulty hints might be used which once again cause performance penalty. Hence, we need to identify the optimal HRP. From our experiments, an HRP of 2000 clock cycles is adequate. A functional SM will use a counter to keep track of the cycles elapsed since last accuracy check. When this counter reaches the HRP value, the accuracy of all hints are checked and the counter is reset.

Each hint has a specific format with several fields as described below.

- **(faulty) SM ID (6 bits):** The SM ID information is used to communicate with a faulty SM to turn off ineffective hints.
- **Hint Type (2 bits):** This depends on the number of hint types.
- **Time-stamp (32 bits):** Contains the current PC at the faulty SM when the hint is received by the functional SM. This information is required so that the functional SM hint processing unit can know whether or not the hint should be used now, waited on, or thrown away.
- **Warp ID (10 bits):** This is determined by the maximum allowable number of hardware threads.
- **Hint Type Specific Information (54 bits):** Includes data that corresponds to that type. Typically, this includes the address of the memory to fetch. The length of the field will vary, as different data will need to be encapsulated within the hint.

The following format and microarchitecture implementation are just one way of implementing our scheme. Other formats and designs are possible.

C. GPU microarchitecture support to generate and distribute hints

The hardware needed to support hint generation and usage consists of four parts: hint gathering unit (per SM), hint processing unit (per SM), hint disabling unit (per SM), and hint queue connecting SMs within the TPC. Since any SM can be faulty, all SMs have to be incorporated with the hint collection hardware and hardware that uses these hints.

- **Hint Gathering:** This unit is responsible for gathering the data necessary to form a hint. Figure 2 shows a high level design of the hint gathering unit. The design of this unit is very simple. Its main task is to complement the collected data (i.e. the hint itself) with other identifiable markers, namely the program counter (PC) of the currently executing instruction, active warp ID, and SM ID that has generated the hint.

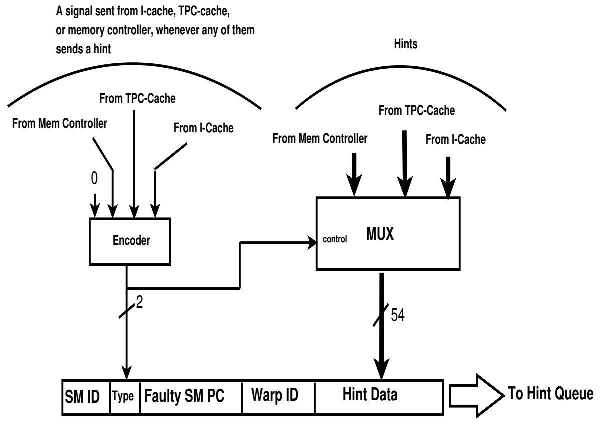

Fig. 2. Hint gathering unit: when instruction-cache, TPC-cache, or memory controller sends a hint, it also sends a signal indicating it is sending a hint.

When the hint is unpacked these markers are extracted. The PC of the functional and faulty SMs are compared to verify that the functional SM has not already executed the instruction the hint was provided for. Warp ID and SM ID are used to forward hint data to the appropriate destination when the hint is finally used.

- **The hint processing** unit is responsible for receiving the hints from the hint queue and unpacking their format in order to extract the hint related information. The unit checks the difference between the functional SM PC and hint PC and if it is less than a specific threshold, the hint is forwarded to the I-Cache to be used. If the difference is greater than the threshold, the hint remains in the hint queue until it is checked once again by the hint distribution unit. If the difference is negative, the hint is thrown away as the functional SM has already executed the instruction the hint was meant for.

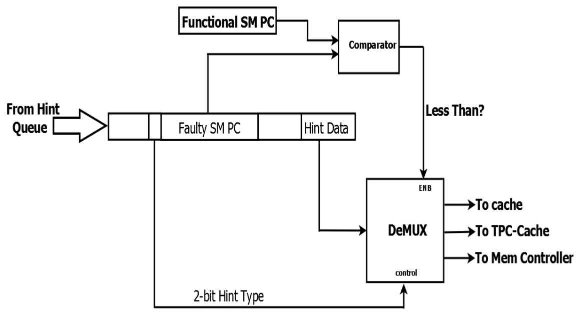

Fig. 3. Hint processing unit: Depending on the PC of faulty SM and functional PC the hint can be used, discarded, or kept in the queue for later. If the hint is used, the 2-bit hint type directs the hint data to the corresponding hardware unit (instruction cache, TPC-cache, or memory controller.

- **Hint disabling unit:** It is possible for a faulty SM to stray off the correct execution path and in turn it is possible for the effectiveness of hints to become low. It would then be necessary to disable the corresponding hints so as to not incur the overhead associated with generating them. The hint disabling unit reports the ineffective hint types that need to be turned off to the faulty SM. At the end of each HRP, if it turns out that more than half of the available hint types are turned off, the functional SM and the faulty are resynchronized by loading the faulty SM with instructions currently being executed and register values from the functional SM. To do that, the hint disabling unit is equipped with an up/down counter per hint type. The counter is incremented if the hint results in useful progress, such as I-cache hit, and decremented when the hint results in wasted effort.

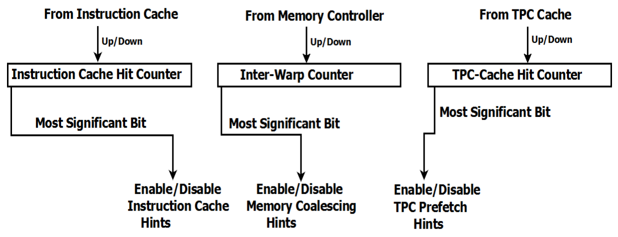

Fig. 4. Hint disabling unit. The Most Significant Bit of each hint performance counter enables/disables each hint generation unit.

- **The hint queue** is the primary communication mechanism for the SMs in the same TPC and contains space for 2k hints, or 24 KB. In our experiments, this size was large enough to accommodate the average number of hints in the queue(about 1400 hints) while adding little hardware overhead to the design. Each SM has access to the TPC hint queue that is filled by the faulty SM. The faulty SM will send the hints to this queue and the functional SMs will use those hints from the queue. Whenever a functional SM uses a hint from the queue, it is removed from the queue. If the hint queue is full, the hint gathering process in the faulty SM will be stopped. Figure 5 shows a SM which has been modified to support our approach. The added hardware is shown with dashed borders and the connections are indicated with arrows.

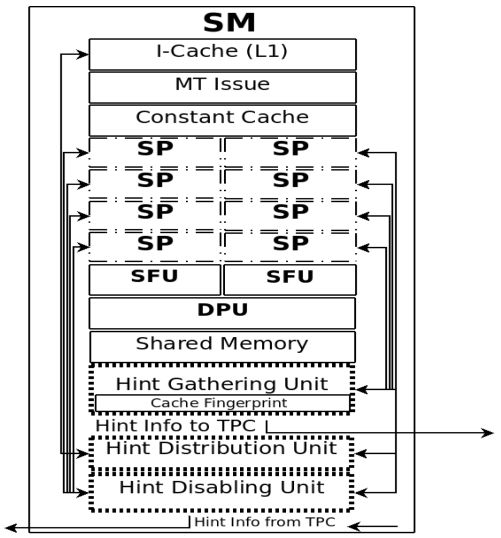

Fig. 5. Modified SM With Hint Generation and Processing Hardware

D. Hardware Cost

The hardware cost can be divided into three main components: the cost of the hardware itself, performance overhead, and energy efficiency.

In every SM we need hint gathering, processing, and disabling units. From the above descriptions of these units we can see that we need: MUX, encoder, comparator, DeMUX, and three counters. We also need two hint registers as shown in Figures 2 and 3. The size and power dissipation of these extra hardware parts are minimal compared to the huge size of SM. Moreover, the exact size depends on type and size of hints. Each TPC needs to have a hint queue and a counter to keep track of the number of cycles elapsed since last HRP period. The performance overhead will be discussed in details in Section III-B.

The extra hardware consumes some energy. But this is extra energy is reduced by the hint disabling unit when hints are not useful. The useful hints reduce execution time, as will be shown in Section III-B, which, in turn, reduces energy dissipation in many cases.

III. Experimental Setup and Results

In this section we present our experimental setup followed by our results and analysis.

A. Experimental Setup

In our experiments, we used a cycle-accurate GPU simulator called GPGPU-Sim [14], which was found to have 0.97 correlation with real hardware, and configured to model NVIDIA

Tesla GPU. The parameters of this NVIDIA Tesla GPU are as follows: 14 TPCs with 2 SMs per TPC for a total of 28 SMs. Each SM was configured to contain 8 SPs with a maximum of 1024 concurrently executing threads per SM.

A total of 10 benchmark kernels are used in this study. These benchmarks vary greatly, both in their applications and implementations. Some of the benchmarks such as AES utilize both the shared and constant memories while others such as WP and MUM do not. The benchmarks also differ in the dimensions of grids and blocks. All of these parameters have an effect upon benchmark performance. Table I shows each of the benchmarks next to their names as well as the number of threads and the number of instructions executed.

TABLE I
THE BENCHMARKS USED IN THE EXPERIMENTS. TABLE BASED ON INFORMATION IN [15]

Benchmark	Abbreviation	# of Threads
AES Cryptography	AES	65792
Breadth First Search	BFS	65536
LIBOR Monte Carlo	LIB	4096
3D Laplace Solver	LPS	12800
MUMmer GPU	MUM	50000
Neural Network	NN	28392
N-Queens Solver	NQU	21408
Ray Tracing	RAY	65536
StoreGPU	STO	49152
Weather Prediction	WP	4608

B. Experimental Results and Analysis

First, we discuss hint accuracy. Hint Accuracy is a metric that measures how effective each hint type is. This metric is dependent on the nature of the benchmark itself and the hint gathering and distribution units. Hint accuracy of 0% shows that none of the generated hints of a certain type contributed to accelerating the execution of the benchmark, while hint accuracy of 100% shows that every single generated hint was delivered and used correctly by the functional SM.

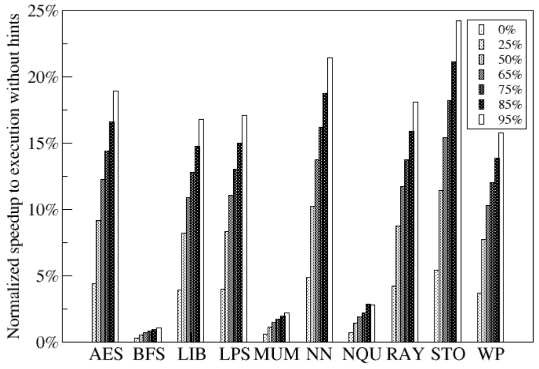

Fig. 6. Maximum speed-up obtained using I-Cache hints. These values show the maximum achievable speed-up for a given hint accuracy.

Figure 6 shows the maximum performance gain of each of the benchmarks for different I-cache hint accuracies. The figure is a limit study assuming perfect hint gathering and distribution units. The effectiveness of this type of hints depends on I-cache hit rate. The higher the I-cache hit rate without hints, the lower its benefit from the hints, which is obvious for BFS, MUM, and NQU.

Figure 7 shows the speed-up obtained by our proposed approach in the case of using only Inter-SM Warp Memory Coalescing and TPC Cache Prefetch Hints. We combined them because they are both related to the memory hierarchy used to deal with data. A benchmark like WP has the least benefit from such hints because the majority of WP instructions are computations not memory accesses. NN too does not benefit much from this type of hints because it has the lowest warp occupancy.

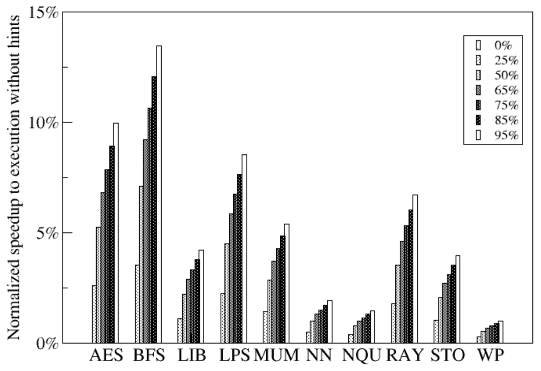

Fig. 7. Maximum speed-up obtained by enabling Inter-SM Warp Memory Coalescing and TPC Cache Prefetch Hints

Figure 8 combines the performance gain associated with each of the 3 type of hints, broken down by hint accuracy. In that graph, all hints are enabled and no hints are turned off. The benefit depends on the instruction mix of each program.

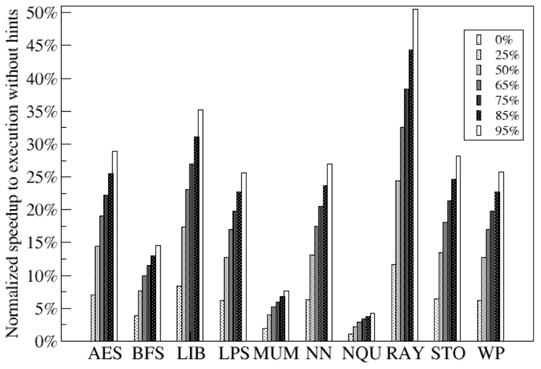

Fig. 8. Speed-up obtained by enabling all available hints

The experimentally determined average hint accuracy values for I-Cache Prefetching Hints is 84%, while the average hint accuracy value for Inter-SM Warp Memory Coalescing Hint and TPC L2 Prefetch Memory Hints is about 67%. The main reason for this is that fetching instructions is usually more predictable than data. Figure 9 shows the total speed-up obtained by enabling turning on and off hints.

From Figure 9 we see that benchmarks such as AES, LIB, and LPS, that gain appreciable speed-up, not only composed largely of ALU and MEM instructions but are also able to balance the workload amongst all available resources. Additionally, benchmarks that do not perform well, such as NQU and MUM, have a large amount of CONTROL instructions, which negatively impact the performance of parallel kernels because of warp branch divergence. In the case

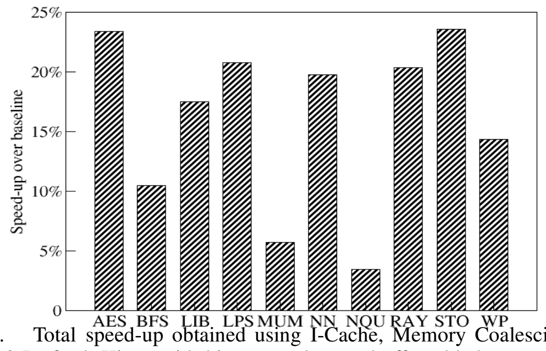

Fig. 9. Total speed-up obtained using I-Cache, Memory Coalescing, and TPC L2 Prefetch Hints, with hints turned on and off enabled

of LPS, which has about 19% CONTROL instructions, the performance penalties associated with these instructions are balanced with speed-ups associated with hints pertaining to ALU and MUM instructions, which compose the rest of the benchmark's instruction breakdown. It is also worth noting that benchmarks that make use of MAD (multiply-add) instructions also exhibit favorable performance in our scheme as well. Figure 10 summarizes instructions breakup for each of the simulated benchmarks.

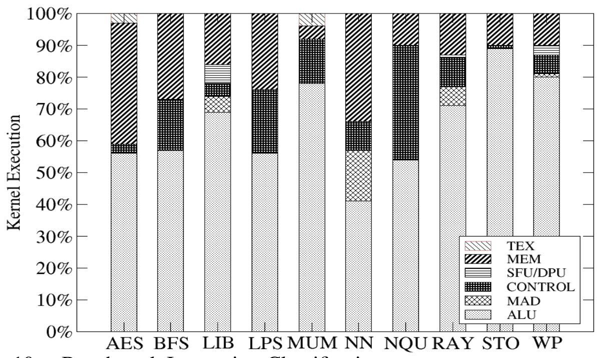

Fig. 10. Benchmark Instruction Classification.

From figure 10 we see that the two lowest performing benchmarks, NQU and MUM, at 3.48% and 5.74% respectively (from Figure 9), are also the two benchmarks which experienced the least performance penalty. This behavior is due to their mostly linear execution. While other benchmarks are able to distribute the work evenly amongst all available SMs, the nature of the tasks for NQU and MUM requires that a single SM operates alone for a large portion of time. Because this single SM is non-faulty all performance gains associated with our approach are effectively hidden. By excluding these two benchmarks and taking the average from the other 8 benchmarks we see a total speed-up of 18.8%. The introduction of additional hint types may also increase the speed-up associated with our approach. This is a great achievement as opposed to simply turning off these computational units.

Our approach is highly scalable because it is developed with respect to TPCs. Because the hardware is duplicated within each TPC, our approach can handle several failing SMs within different TPCs, and thus help diminish the performance penalty associated with this. The hardware overhead is not high, as TPCs are quite small and are currently composed of few SMs. Thus, interconnection paths required to connect the SMs within the same TPC are not extensive.

IV. Conclusion

In this paper we made use of faulty-SMs to speed-up non-faulty SM within the same TPC of a GPU. We showed that our techniques achieved average performance improvement of 15.94% over base systems where a faulty SM is turned off. The proposed technique can be implemented in many different ways. We presented one possible implementation as a proof-of-concept that hints can be useful in the GPU arena as much as they are useful in multicore processors.

Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-09-1-0146. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Ars Technica, "NVIDIA denies rumors of faulty chips, mass GPU failures" http://arstechnica.com/hardware/news/2008/07/nvidia-denies-rumors-of-mass-gpu-failures.ars
[2] NVIDIA, "GeForce GTX 480." http://www.nvidia.com/object/product_geforce_gtx_480_us.html
[3] Slashgear, "NVIDIA GeForce GTX 480/470 to lose cores over poor GPU yield?" http://www.slashgear.com/nvidia-geforce-gtx-480470-to-lose-cores-over-poor-gpu-yield-2278420/
[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture" *IEEE Micro*, vol. 28, pp. 39-55
[5] I. Haque, V. Pande. "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU" In Proc. of the 10th Annaul IEEE/ACM Internation Conference on Cluster, Cloud, and Grid Computing (CCGrid), pages 691 - 696, 2010.
[6] J. Aidemark, P. Folkesson, J. Karlsson. "On the Probability of Detecting Data Errors Generated by Permanent Faults Using Time Redundancy" In Proc. of IOLTS, pages 68-74, 2003.
[7] S. Nomura, M. D. Sinclair, C. Ho, V. Govindaraju, M. de Krujif, K. Sankaralingam. "Sampling + DMR: Practical and Low-overhead Permanent Fault Detection" ISCA, 2011.
[8] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. "Configurable isolation: building high availability systems with commodity multi-core processors" In Proc. of the 34th Annual ISCA, pages 470481, 2007.
[9] S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. Mahlke. "The stagenet fabric for constructing resilient multicore systems" In Proc. of the 41st Annual International Symposium on Microarchitecture, pages 141151, 2008.
[10] T. Austin. "Diva: a reliable substrate for deep submicron microarchitecture design" In Proc. of the 32nd Annual International Symposium on Microarchitecture, pages 196207, 1999.
[11] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos. "Enhancing System Throughput by Animating Dead Cores." In Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 235-246, 2010.
[12] A. Ansari, S. Feng, S. Gupta, S. Mahlke."Putting Faulty Cores to Work" *IEEE Micro*, vol. 30, pp. 36-45
[13] A. Ansari, S. Feng, S. Gupta, S. Mahlke. "Demystifying GPU microarchitecture through microbenchmarking." In Proc. of 37th International Symposium on Computer Architecture (ISCA), 2010.
[14] W. W. L. Fung, I. Sham, G.L. Yuan, T. Aamodt. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow" In Proc. of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 407-420, 2007.
[15] A. Bakhoda, G.L. Yuan, W.W.L. Fung, T. M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator" In Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 163-174, 2009.