Hybrid CPU/GPU Checkpoint for GPU-Based Heterogeneous Systems

Lin Shi[1,2], Hao Chen[2], and Ting Li[1]

[1] School of Computer Science and Enginerring, Hunan University of Science and Technology, Xiang Tan, China
[2] School of Computer and Communication, Hunan University, Chang Sha, China

Abstract. Fault tolerance has become a major concern in exascale computing, especially for the large scale CPU/GPU heterogeneous clusters. The performance/cost benefit of GPU based system is subject to their abilities to provide high reliability, availability, and serviceability. The traditional CPU-based checkpoint technologies have been deployed on the GPU platform but all of them treat the GPU as a second class controllable and shared entity. As existing GPU checkpoint/restart implementations do not support checkpointing the internal GPU status, the codes running on GPU (kernel) can not be checked/restored just like the CPU codes, all the checkpoint operation is done outside the kernel. In this paper, we propose a hybrid checkpoint technology, HKC (Hybrid Kernel Checkpoint). HKC combines the PTX stub inject technology and dynamic library hijack mechanism, to save/store the internal state of a GPU kernel. Our evaluation shows that HKC increases the system reliability of CPU/GPU hybrid system with a very reasonable cost, and show more resilience than other checkpoint scheme.

Keywords: checkpoint, GPU, CUDA, kernel.

1 Introduction

Over the past few years, the modern 3D graphics processing unit (GPU) has evolved from a fixed-function graphics pipeline to a programmable parallel processor with computing power exceeding that of multicore CPUs. Under the GPGPU concept, NVIDIA has developed a C language based programming model, Compute Unified Device Architecture (CUDA) [1], which provides greater programmability for high-performance graphics devices. GPU programming has been successfully exploited in recent years for resolving a broad range of computationally demanding and complex problems.

For the high performance computing (HPC), the exponential growth of GPU supercomputers in the Top500 list has prove the efficiency of large GPU clusters. Since 2000, a large number of large-scale GPU-based clusters merged as the fasted supercomputer in the world. For example, the first (Tianhe-1A), the third (Nebula), and the fourth (Tsubame2.0) fastest supercomputer in the top 500 list of November 2010 are all CPU/GPU heterogeneous systems.

Although the GPGPU paradigm successfully provides significant computation throughput, the reliability of GPUs in error-intolerant applications is largely unproven. Traditionally the 100% accuracy is not necessary for GPU because the errors only affect a few pixels or a few frames and where performance is more important than accuracy. However, an error in a GPGPU application will crash down the whole program or produce an incorrect result. Since the purpose of GPGPU is to perform massive amounts of computation, the failure of a GPU could result in significant loss of application progress. The Fermi and Kepler GPUs of Nvidia already support the ECC (Error Correcting Code) in graphic memory to provide some kinds of fault tolerance, but the GPU is still vulnerable to control logic errors and multi-bit errors in memory. A test in the Tokyo Institute of Technology observed eight errors in a 72-hour run of a GPGPU testing program on 60 NVIDIA GeForce 8800GTS 512 [2]. Stanford University find that the two-thirds of tested GPUs on Folding@home exhibit a detectable, pattern-sensitive rate of soft error [3] .

The NVIDIA developed a new compute mode SIMT (single-instruction, multiple-thread) based on the SIMD (single-instruction, multiple-data). The feature of SIMT lead us to develop the HKC (Hybrid Kernel Checkpoint), a novel state store/recovery method for GPU code. HKC support to check a running kernel at any time, at any place, and recover from errors detected in a code region by partially recomputing the region. Meanwhile HKC is totally transparent to the programmer, no source code modification is needed to perform the checkpoint.

To the best of our knowledge, HKC is the first work on how to check/restore the kernel level state in a transparent way.

In summary, the main contributions of our work are:

– We introduce HKC, a hybrid-kernel-checkpoint method to recover from some given GPGPU faults efficiently by exploiting the SIMT characteristics of programs running on GPGPUs. HKC overcome some serious shortcoming of traditional method: the inefficiency from the full re-computation, the lack of flexibility in check interval and the intrusiveness arise from some compiler-aid method.
– We give a detail search on the key features of SIMT architecture and show the possible to checkpoint this new architecture in a transparent way.
– We describe an implementation of our framework on the CUDA platform for NVIDIA GPGPUs.
– We carry out detailed performance evaluation on the overhead of our framework. This evaluation shows that the HKC framework is practical and can deliver high performance for checking/restoring GPGPU applications in the CPU/GPU hybrid system.

While this work focuses on checkpoint on CPU/GPU heterogeneous systems, HKC is also applicable to any distributed systems equipped with the SIMT-style processors.

The rest of the paper is organized as follows: In Section 2, we provide some necessary background about this work. Section 3 discusses the design and implementation of HKC respectively. Next we evaluate HKC in Section 4 . In Section 6, we discuss related work. Finally, in Section 7 we present our conclusions.

2 Backgroud

2.1 CUDA

In 2007 NVIDIA propose the CUDA (Compute Unified Device Architecture) framework. CUDA is the first dedicate GPGPU interface on the GPU platform, which provide a C-like semantics to write the parallel code (kernel), and give programmer the ability to direct control the GPU. In recent years a large number of compute-intensive applications have been ported to CUDA platform, the power of CUDA and GPU has been proved in many cases. The device code running on the GPU is usually called a KERNEL according to the CUDA's terminology.

Runtime and Driver API of CUDA. There are two different kinds of CUDA API exist in the CUDA software stack. The low-level API system is called the CUDA driver API, the high-level, runtime API. The runtime API is easier to use, but the driver API gives programmer more control over low level details. They both provide the interface to allocate/withdraw the graphic memory objects, and launch the kernel.

3 Design and Implementation

The checkpoint-based GPU fault-tolerant technology will periodically save the current execution state of the GPU, and roll back the GPU state to a certain checkpoint when a failure happened. It comprise two phases: checking phase and restore phase.

Checking Phase. During the normal execution of GPU application, triggered by the user or system, the current GPU state is copied and stored in the persistent media, which is called a checkpoint file.

Restore Phase. When a fault occurs or application get unexpected results, the GPU rollback to a previous checkpoint, then continue to run from this state.

The key issue of checkpoint technology is how to define, check and restore the application state. The checkpoint image should be the complete unity combining the internal and external state of a program which support the its successful execution in the future. Typical states for Linux and Windows applications include registers, heap, stack, thread, address space, and signal, lock, file, socket, handler, I/O device, external connections etc. Relative to the applications running on the CPU, the internal state of code running on the GPU (Kernel) exhibit some different characters. The kernel state lies in the GPU and device memory against CPU and host memory, the address space of kernel is organized by the CUDA runtime against the operating system. Further, Kernel state can be divided into in-kernel state and out-of-kernel states, depending on the life cycle of objects.

For example, the global memory belongs to the out-of-kernel states because the object in global memory will persist during different kernel's execution unless an explicit API is called to delete them. The on-chip resource (register file,control unit,local memory, share memory) are in-kernel state because they are valid only when a kernel is running. A CUDA application may involve many kernels, each one of them involves different internal state when they are executing on the SMs.

In short, the out-of-kernel state is relatively simple, while the in-kernel state is more complex. In order to reduce the complexity of the GPU checkpoint, the existing CPU/GPU hybrid checkpoint mechanism [17, 23–26] focus only the simplified out-of-kernel state. Figure 1 shows the flow chart of OKC: When the CPU receives the checkpoint signal it does not immediately perform a checkpoint, but to determine the current state of the GPU. If there is no any running kernel and pending kernel in the queue, the checkpoint is performed. Otherwise OKC will wait until running or pending kernels all finished. If an error happened in the the kernel execution, the entire system is rolled back to the previous checkpoint.

As can be seen with reference to Figure 1 , OKC achieve both concision and effectiveness by keeping the check/restore mechanism out of the kernel. But there are many shortcomings with OKC as follows:

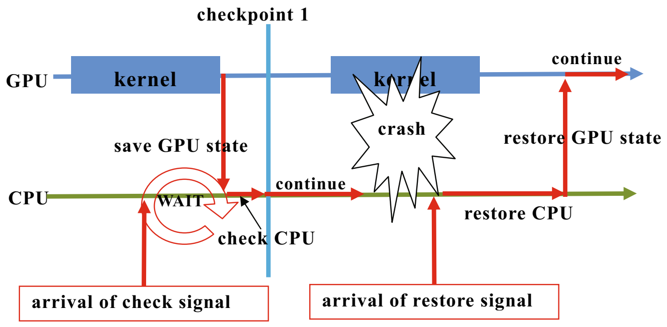


Fig. 1. The control flow of OKC

Non Real-Time Processing in Check Stage. OKC must wait until the kernel completed means there exists a delay between the arrival of check signal and actual check operation. The delay time depend the total execution time of running and pending kernels. For long-running kernel the check operation is postponed in a great extend. This has a negative impact on application require strong real time capacity. Whats more, in the GPU cluster, a large number of GPUs must be synchronized to the keep the consistency of global checkpoint. If some of them are running heavy kernel the overall OKC check time will be extended even lead to domino effect (A distributed system can not find the global consistent checkpoint, all task roll back to the initial state and lost all the effective computation).

Unbalance Interval Time of Check. Checkpoint technology needs to balance the relationship between the fault tolerance capability and inspection overhead. Only when the check event is distributed evenly during the execution of GPGPU applications, can the checkpoint technology achieve optimal tradeoff balance between performance and reliability. But the check interval of OKC is heavily depend on the workload of kernel. For example, although the administrator can set a reasonable check interval to one hour, the OKC has to stretched this interval to threes hours because of a long-running kernel. Theoretically the longer a KERNEL is running, the more frequently it should be checked since it is more vulnerable to errors, which OKC can not do.

Time-Consuming in Recovery Phase. In recovery phase OKC should recompute the entire kernel from the last checkpoint even only one transient fault is detected on the GPU. Each kernel need be re-started from scratch and thousands of threads in it are fully recalculated. The kernel represent the most intensive computation in GPGPU applications, the cost of such full-fledged recomputing may be unnecessarily high.

The shortcoming of OKC lead us to find a more flexile checkpoint mechanism. In CUDA framework a kernel is decomposed into many CTAs, which run independently from each other and occupy SMs in the forms of groups. The execute order of these groups, the binding relationship of CTA to SMs are not guaranteed by the CUDA framework. The programmer should not make any assumptions or rely on the order of CTAs or coupling schemes between CTA and SM.

The above characteristics determines the correctness of the kernel depend on the correctness of various CTA, regardless of the execution order between the CTA or the coupling relationship between CTA and SM. Based on this observation, we develop a partially check/restore mechanism which cares about the CTA-level state in the kernel.

When a kernel is suspended, all CTAs must freezed in one of the following three situation: finished (endCTA), on-the-fly (halfCTA), or unborn (beginCTA).

endCTA contains the threads have ended their job, the calculation results are reflected in the global memory

halfCTA: contains threads are running on the SMs. Notes in CUDA the threads in the same CTA may execute different instructions.

beginCTA: the threads in it do not start yet.

So a sophisticated check/restore scheme should save the internal state for the halfCTA and record which CTA is finished or unborn in the check phase. In the restore phase, the check/restore scheme make is sure to never re-execute the endCTA, and re-execute the halfCTA from the place it break and re-launch the beginCTA from the beginning. In some case the kernel is reenterable so it is ok to re-execute the endCTA but it is difficult to distinguish whether a kernel is reentrant or not. For performance reasons, the restore operation should not repeat the already completed calculation.

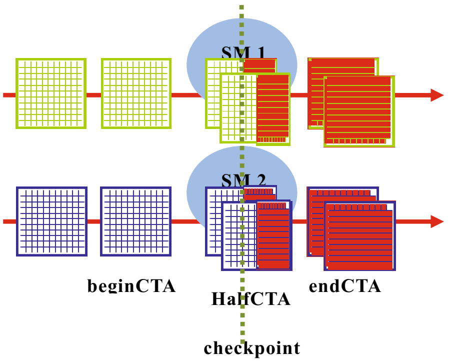


Fig. 2. Three kinds of CTA in a suspended kernel

The above analysis tell us the the CTA running the same kernel code must forked in the restore phase according to their execution state in the check phase. The restore code with different branches should be injected to the kernel which we called the *HKC stub*.

There are many ways to realize such a check/restore scheme, in our design of HKC, we put three main factors in minds:

Transparency. HKC should not force the programmer do anything outside the CUDA program model, which means, HKC need not any source code modification or introduce extra programming rules. In addition, we would like our techniques to easily extend to other SIMT/SIMD architecture, allowing people to use GPU checkpoint tools just like the traditional system-level CPU checkpoint tools.

Performance. The main advantage of GPGPU computation is the high performance, while the checkpoint is recognize as a high overhead fault tolerance technology. HKC should minimize these overhead, combine the strengths of high performance from CUDA and reliability from checkpoint.

Compatibility. We would like our HKC to easily combined to the already exist CPU-based checkpoint technology, to form a sophisticate CPU/GPU hybrid checkpoint system.

4 Evaluation

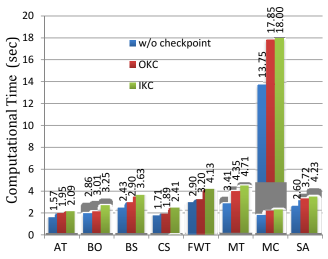
While the previous sections have presented detailed technical descriptions of the HKC, this section evaluates the efficiency of HKC using programs selected from official SDK examples: a set of general-purpose algorithms from various research area. The benchmarks range from simple data management to more complex WalshTransform computation and MonteCarlo simulation. Table 1 shows the statistical characteristics of these benchmarks, such as the quantity of API calls,

Table 1. Statistics of Benchmark Applications

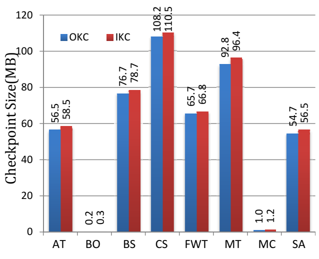
	Number of APIs	GPU RAM	Data Volume
AlignedTypes (AT)	1990	94.00MB	611.00MB
BinomialOptions (BO)	31	0.01MB	0.01MB
BlackScholes (BS)	5143	61.03MB	76.29MB
ConvolutionSeparable (CS)	48	108.00MB	72.00MB
FastWalshTransform (FWT)	144	128.00MB	128.00MB
MersenneTwister (MT)	24	91.56MB	91.56MB
MonteCarlo (MC)	53	187.13MB	0.00MB
ScanLargeArray (SLA)	6890	7.64MB	11.44MB

the device memory size they consume and the data volume transferred from or to GPU device. As a comparison we re-implement the OKC mechanism developed in the [13].

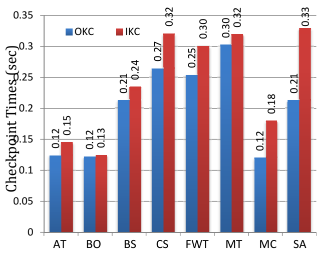
The following testbed has been used for all benchmarks: A HP Proliant ML150G6 server equipped with one Intel E5504 processors with four cores and provided with 8 GBytes of memory. Furthermore, the graphics hardware was NVIDIA's GTX470 with 1.2GBytes graphic memory. As for software, the test machine ran the Fedora 13 Linux distribution with the 2.6.33.3 kernel, with the official NVIDIA driver version 169.19.26 for Linux and CUDA toolkits version 3.2.



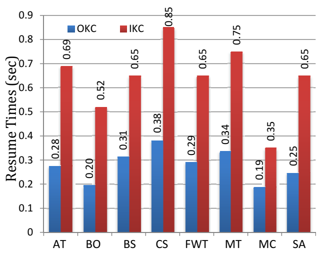
(a) Runtime Overhead



(b) Check Overhead (Space)



(c) Check Overhead (Time)



(d) Resume Overhead

4.1 Runtime Overhead

Even no check is performed, HKC mechanism will also introduce some overhead which come from the monitoring operations on CUDA library and the stub injection. Figure 3(a) depicts the execution time of each benchmark under three different configuration: without any checkpoint mechanism, with HKC and with OKC. As it shows, the maximum overhead of OKC is up to 42% (SA), minimum of 5% (BO), an average of 21%. The runtime overhead of HKC is up to 62% (SA), and average of 38%. The HKC runtime overhead is relatively high because the HKC initialization involves the establishment of a memory pool and the operation to inject the stub. OKC also need to keep track of the CUDA API, but they do not need to adjust the KERNEL code so gained some performance advantages. From another perspective, as the supplementation of OKC, HKC only add maximum of 29%, an average of 14% of the overhead at the base of OKC.

Although in eight samples the HKC runtime overhead can not be ignored, partly arise from the facts they involves relative short running kernel. Taking the constant initialization time into account, HKC is still practical to long-running applications. If HKC provide some options to switch on/off itself, so as to distinguish between critical and non-critical KERNEL, the runtime overhead of HKC will be further reduced.

4.2 Checking Overhead

The checking overhead include the time and space overhead introduced by the checkpoint operation. Figure 3(b) shows the average capacity of the checkpoint file. Since the check in HKC happened when the kernel is running the internal state is more complex then OKC, the space overhead of HKC is higher (9% on average) than OKC, the degree depends on the numbers of halfCTA, numbers of CTA and the usage of registers, shared memory. In the BO and MC test HKC increase the check contents to a extent of 40% and 17%, the reason is their state set are relatively smaller making the HKC overhead more obvious.

Time overhead from the checkpoint means the interval between the arrival of the checkpoint signal and the finish of the checkpoint snapshot, as shown in Figure 3(c). Relative to the space overhead, time overhead of the HKC achieve an average of 16% and a maximum of 35%. This is due to the fact HKC need to exploit the CUDA debugging API to obtain the internal state, each of this API involves one or more system calls.

On the basis of both Figure 3(b) and Figure 3(c), it is easy to find a strong positive relationship between space overhead and time overhead. This is because most of time overhead of checking is the write operation on checkpoint file.

4.3 Restore Overhead

The recovery is the process to rebuild the CUDA state on a GPU. Figure 3(d) measure the time for the recovery of each sample program. A main portion of the recovery operation is the reconstruction of the GPU state, and thus the size of the state set has direct impact on the recovery time. In addition to the factors of states, the endCTA still occupy the dispatch unit and instruction unit, the increase of endCTA also degrade the performance of HKC.

5 Discussion

The debug interface of CUDA is exploited in our HKC scheme which may cause some side effect. For example, according to the official document, leave the debug interface open will force the KERNEL running in synchronization mode. Some debug API require the detailed debug information which can be found only in the debug version of CUDA application. The requirement is unacceptable for commodity software, not to mention debug version generally shows awesome performance compared with the release version. We argued the debug interface is not native designed for the check/restore operation, a more sophisticate interface should be provided by the graphic vendor.

In this embodiment of HKC we choose to ignore the state in the halfCTA, it is not only because its complexity but also the lack of method to write the PC (Program Counter) in a warp. The CUDA debugger API provide the device state inspection interface *readPC* and *readVirtualPC*, but no corresponding device state alteration APIs.

6 Related Work

Checkpointing and rollback recovery (Checkpoint and Rollback Recovery) technology is commonly used in post-recovery techniques to fault-tolerant. Before the advent of the GPU checkpoint technology has been widely used in the CPU-based computing systems, such as CRAK [10] BLCR [11], ckpt [12]. Recent years many research concentrate on the GPU-oriented checkpoint mechanism.

CheCUDA [13] is the first GPU checkpoint mechanisms based on the BLCR. Since the BLCR itself does not support GPU device, CheCUDA had to remove the GPU context before starting BLCR, and restore the GPU state after BLCR finished.

As a first dedicate checkpoint for GPU, CheCUDA has some apparent drawback: Firstly the combination way with BLCR is not elegant enough because the context of GPU must be destroyed and rebuilt out of the BLCR. Secondly only two simplest SDK examples involves 20 CUDA APIs out of total 60 official APIs can not prove the effective of CheCUDA. The last but not less important, the performance of CheCUDA is not good enough, it cost more than ten times the normal execution in checking phase. NVCR [15] strengthen the CheCUDA in transparency and adaptability. The authors of CheCUDA put their idea on another GPGPU framwork: OPENCL [20] to form a new checkpoint scheme CheCL [14]. VCCP project [18] discussed the GPU checkpoint mechanism in theory and analyse its optimized CUDA asynchronous mechanism. The biggest problem is VCCP do not implement its mechanism at all (no implementation), the theoretical design can not be verified in practice. Tokyo Institute of Technology has some important progress in the distributed GPU cluster checkpoint technology. Based on the CPU/GPU hybrid Tsubame2.0 supercomputer, in [19]they integrated the BLCR, OpenMPI and GPU checkpoint mechanism, extended the single-node GPU checkpoint technology to multi-GPU field. [17] is the further improvement of distribute GPU checkpoint, the authors take a variety of techniques to optimize the initial design. First, it completely abandoned BLCR, put the checkpointing base on the Reed-Solomon encoding. Second, it dug the compute potential of idle nodes to accelerate the check operation. The third it make full use of the asynchronous mode of CUDA and diskless storage. It is worth noting that although the project shows impressive high efficiency, its GPU checkpoint program is still based on the interception and encapsulation of *cudaMemCpy* function, the check events occurred outside of the KERNEL, thus it still belongs a outside-kernel-checkpoint category.

7 Conclusion

In this paper we proposed HKC, an inside-kernel-checkpoint mechanism for CUDA applications. HKC allow the administrator to fully leverage the traditional check/restore technology to increase the reliability of GPUs. We explained how to access the kernel state by using the CUDA debugger interface, and how to suspend/resume a running kernel. Compare to the OKC, HKC is not limit to the check interval and exhibit high performance because of its partial recover nature. HKC need no any source code modification and is totally transparent to the programmer. Our evaluation showed that HKC for HPC applications is feasible and competitive with OKC.

The new methodology proposed in HKC is simple yet effective as it is particularly suited for an implementation of fault recovery techniques for GPGPU programs, by leveraging the tremendous computing power of GPGPUs and exploiting the SIMT characteristics of GPGPU programs.

Acknowledgment. We thank the anonymous reviewers for their helpful feedback. This research was supported in part by Hunan Provincial Natural Science Foundation of China under grant 13JJB006, the National Science Foundation of China under grants 61133005, 61070057 and 61272190.

References

1. CUDA: Compute Unified Device Architecture (accessed September 2012), `http://www.nvidia.com/object/cuda_home_new.html`
2. Maruyama, N., Nukada, A., Matsuoka, S.: A high-performance fault-tolerant software framework for memory on commodity GPUs. In: Proc. Int'l Symp. Parallel and Distributed Processing (IPDPS 2010), pp. 1–11 (April 2010)

3. Haque, I.S., Pande, V.S.: Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In: 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp. 691–696 (2010)
4. NVIDIA CUDA debugger API Reference Manual, http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation
5. Allinea DDT, http://www.allinea.com/products/ddt
6. TotalView, http://www.roguewave.com/products/totalview.aspx
7. Shi, L., Chen, H., Sun, J.: vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. In: Proc. Int'l Symp. Parallel and Distributed Processing (IPDPS 2009), pp. 1–11 (May 2009)
8. GPGPU: General Purpose Programming on GPUs, http://www.gpgpu.org/w/index.php/FAQ#WhatprogrammingAPIsexistforGPGPU.3F
9. Tian, Z.A., Liu, R.S., Liu, H.R., Zheng, C.X., Hou, Z.Y., Peng, P.: Molecular dynamics simulation for cooling rate dependence of solidification microstructures of silver. Journal of Non-Crystalline Solids 354, 3705–3712 (2009)
10. Zhong, H., Nieh, J.: CRAK: Linux Checkpoint/Restart As a KERNEL Module. Technical Report, Columbia University,2002
11. Duell, J.: The Design and Implementation of Berkeley Labs Linux Checkpoint/Restart. Paper LBNL-54941. Berkeley,2005
12. Litzkow, M., Tannenbaum, T.: J. Basney, et al. Checkpoint and Migration of UNIX Process in the Condor Distributed Processing System. Technical Report, 1346, University of Wisconsin Madison
13. Takizawa, H., Sato, K., Komatsu, K., et al.: CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In: Proc. of International Conference on Parallel and Distributed Computing Applications and Technologies, Higashi Hiroshima, pp. 408–413 (2009)
14. Takizawa, H., Koyama, K., Sato, K., et al.: CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In: Proc. of International Parallel and Distributed Processing Symposium, Anchorage, pp. 864–876 (2011)
15. Nukada, A., Takizawa, H., Matsuoka, S.: NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA. In: Proc. of IPDPS Workshop, Alaska, pp. 104–113 (2011)
16. Li, T., Narayana, V.K., El-Araby, E., et al.: GPU Resource Sharing and Virtualization on High Performance Computing Systems. In: Proc. of International Conference on Parallel Processing, Taipei, pp. 733–742 (2011)
17. Bautista, L., Nukada, A., Maruyama, N., et al.: Low-overhead diskless checkpoint for hybrid computing systems. In: Proc. of High Performance Computing, Dona Paula, pp. 1–10 (2010)
18. Laosooksathit, S., Naksinehaboon, N., Leangsuksan, C., et al.: Lightweight Checkpoint Mechanism and Modeling in GPGPU Environment. In: Proc. of HPCVirt Workshop, Paris (2010)
19. Toan, N., Jitsumoto, H., Maruyama, N., et al.: MPI-CUDA Applications Checkpointing. In: Proc. of Summer United Workshops on Parallel, Distributed and Cooperative Processing. Technical Report, Kanazawa (2010)
20. OpenCL: Parallel Computing on the GPU and CPU. In Beyond Programmable Shading Course of SIGGRAPH 2008 (August 14, 2008)
21. Chen, H., Shi, L., Sun, J.: VMRPC: A High Efficiency and Light Weight RPC System for Virtual Machines. In: The 18th IEEE International Workshop on Quality of Service (IWQoS), Beijing, China (2010)

22. Mohr, A., Gleicher, M.: HijackGL: Reconstructing from Streams for Stylized Rendering. In: Proc. of International Symposium on Non-photorealistic Animation and Rendering, New York, p. 13 (2002)
23. Xu, X., Lin, Y., Tang, T., et al.: HiAL-Ckpt: A hierarchical application-level checkpointing for CPU-GPU hybrid systems. In: Proc. of International Conference on Computer Science and Education, Hefei, pp. 1895–1899 (2010)
24. Dimitrov, M., Mantor, M., Zhou, H.: Understanding software approaches for gpgpu reliability. In: Proc. of Workshop on General-Purpose Computation on Graphics Processing Units, Washington, pp. 94–104 (2009)
25. Sheaffer, J., Luebke, D., Skadron, K.: A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors. In: Proc. of ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, San Diego, pp. 55–64 (2007)
26. Maruyama, N., Nukada, A., Matsuoka, S.: A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs. In: Proc. of IEEE International Symposium on Parallel & Distributed Processing, Atlanta, pp. 1–12 (2010)