A Lightweight Approach to GPU Resilience

Max Baird(✉), Christian Fensch, Sven-Bodo Scholz, and Artjoms Šinkarovs

Heriot-Watt University, Edinburgh, Scotland
{mmb1,c.fensch,s.scholz,a.sinkarovs}@hw.ac.uk

Abstract. Resilience for HPC applications typically is implemented as a CPU-based rollback-recovery technique. In this context, long running accelerator computations on GPUs pose a major challenge as these devices usually do not offer any means of interrupt. This paper proposes a solution to the aforementioned problem: it suggests a novel approach that rewrites GPU kernels so that a soft interrupt of their execution becomes possible. Our approach is based on the Compute Unified Device Architecture (CUDA) by Nvidia and works by taking advantage of CUDA's execution model of partitioning threads into blocks. In essence, we rewrite the kernel so that each block determines whether it should continue execution or return control to the CPU. By doing so we are able to perform a premature interrupt of kernels.

Keywords: HPC · GPU · Resilience

1 Introduction

A large number of high-performance systems these days are equipped with GPGPUs [2,6,7,13,15], as they provide higher energy efficiency and offer a significantly larger degree of parallelism than traditional multi-core CPUs. As a result, the number of compute cores on such systems becomes very large, which in turn, increases the probability of hardware failures. This brings the problem of resilience to hardware failures, which is known to be a challenging topic already [3,5], to the next level. First, the mean time between failures (MTBF) for a single node becomes shorter. Second, resilience for failing GPU nodes requires special treatment.

The de-facto resilience technique today is *application checkpointing*. A checkpointing system pauses the running application and takes a snapshot of its state. The state is either captured automatically by recording register values and the state of the memory of a paused process (*e.g.* by using software such as BLCR [1]) or by explicit stores of relevant data (*e.g.* by using libraries such as FTI [4]). On restore, the captured state is restored and the application restarts its execution from the latest checkpoint. For applications that use GPUs, the described checkpointing mechanism will not work without further measures. GPU kernels do not run as a part of any operating system processes. Even if a process is suspended,

G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 826–838, 2019.
https://doi.org/10.1007/978-3-030-10549-5_64

the GPU kernels keep on running. Actually, at the time of writing, we are not aware of any hardware mechanisms to interrupt a running GPU kernel.

This behavior poses a serious problem to automated checkpoint mechanisms such as BLCR: as the GPU cannot be interrupted, it is not possible to save a stable snapshot of the GPU state. Checkpoints are only possible between kernel invocations. The problem is intensified, as the size of memory on the GPUs increases, resulting in longer runs of the individual kernels [18,19]. In the case of long kernel execution times, explicit stores of relevant data do not help either as snapshots can only be orchestrated between kernel executions. Furthermore, snapshots require all relevant data to be present on the host.

This paper focuses on finding a solution to the GPU kernel snapshotting and restoring problem in a checkpointing-system agnostic way. We propose an approach that is based on the observation that most GPU kernels schedule orders of magnitude more threads than a GPU can physically execute concurrently. While it is not possible to interrupt an individual thread, a thread can voluntarily stop its execution. Thus, in principle, we are able to interrupt a kernel execution, after the currently running threads have terminated.

We describe a technique on how to rewrite CUDA kernels so that they become "interruptible", and we provide a library[1] with a concise API to simplify this task. We demonstrate how the proposed approach can be used by modifying the code of a real world application. We measure the overheads that our approach brings, using real-world and synthetic benchmarks, concluding that typically the overheads are below 0.2%. This shows that the proposed approach can be used in combination with any checkpointing system for applications that use GPUs.

2 Mechanism Description

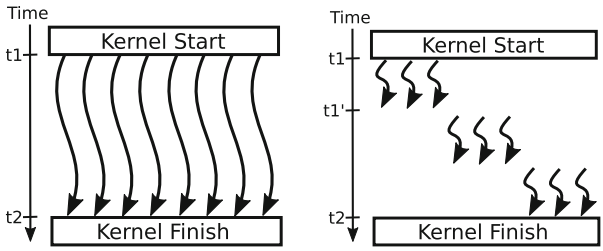
The key idea of our approach lies in the observation that due to resource constraints, it is not possible to schedule all kernel threads simultaneously, instead, threads are scheduled in blocks. After all threads in a block are terminated, they are replaced by remaining threads of the compute kernel. This staggered starting makes it possible to instrument every thread at the beginning of its execution with a check of a shared interrupt variable and terminate the execution if the variable is set to a specific value. As a result, a kernel can be forced to terminate in a very short time.

Issuing an interrupt. In order to implement the interrupt mechanism, we use a memory-mapped integer variable that is shared between the host and the device. On the host we define a variable and ask the CUDA driver to share it with the GPU:

```
1 int *timeout = 0;
2 cudaHostAlloc(timeout, sizeof (int), cudaHostAllocMapped);
```

[1] Freely available at https://bitbucket.org/maxbaird/cuda_backup.

For a host to issue an interrupt, it writes a value 1 into a shared variable `*timeout`. The host waits for the kernel to terminate then transfers data from the GPU. After that the value of `*timeout` can be set back to 0 so that further kernel invocations could perform some useful work.



(a) Conceptual View (b) More Realistic View

Fig. 1. GPU multithread execution model. (a) shows the conceptual view with all threads running in parallel, while in reality (b) the number of concurrent executing threads is limited by hardware resources.

Interrupting a kernel. The CUDA execution model [11] suggests that all the threads are launched simultaneously and the kernel runs till all the threads are completed (see Fig. 1a). However, in reality, threads are scheduled in blocks as it is shown in Fig. 1b.

While indeed a thread cannot be interrupted once it has been started, a thread can decide to interrupt itself. Such a decision can be based on checking the state of a global variable at the start of execution. According to the model from Fig. 1a, this approach would not work: all threads check the variable at the same time and then either all continue or interrupt. However, using the more realistic model, the threads that have not been scheduled will observe a change in the variable and will interrupt. As a result, the kernel terminates faster than the case where we wait for all threads to complete, as we only have to:

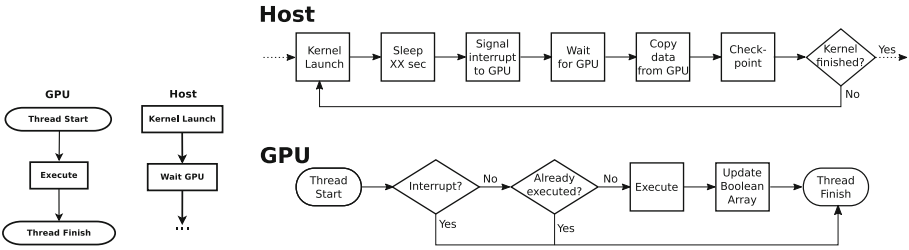
- wait for all the currently scheduled threads to complete; and
- execute all the remaining threads where the first statement within every such a thread will terminate its execution.

Consider a host issuing an interrupt at time $t1$ in Fig. 1b. The kernel can complete at $t1' + \max((c \times n), t_r)$, where c is the time it takes to execute one conditional per block (GPU executes in a lock-step), n is the number of remaining unscheduled blocks, and t_r is the time to finish already scheduled threads.

Snapshotting. After a kernel has been interrupted, the host copies all data that will be necessary to restart the kernel. In the simplest case, these data include inputs of the kernel and partial outputs of the kernel. In addition, we need to perform a bit of bookkeeping via a boolean array which tracks which threads have been executed to completion. At the start of each thread, we check whether

this thread has yet to run. At the end of each thread, we update this boolean array to indicate that it has completed. At every snapshot, we also copy this array back to the host.

Restarting. Restarting a kernel is straight-forward: we copy all the kernel-relevant data back to the GPU and launch the kernel again. We use above mentioned mask to prevent already completed threads from executing again.



(a) Original application (b) Modified application.

Fig. 2. How to apply the proposed technique to the original application.

2.1 Integration with Checkpointing System

Once the GPU kernel has been interrupted and all the relevant data has been copied to host, it is safe to snapshot the global state of the application. However, as most checkpointing systems are not aware of GPUs, it is difficult to predict when the checkpointing will happen, and, as a consequence, when to capture the state of running kernels. Our solution to this is to make kernel snapshots every n time units. After each snapshot, the host checks the boolean array for any unfinished threads. If such threads exists, then the kernel is relaunched. This process continues iteratively until all threads are executed.

To apply the proposed technique, we modify the kernel and the code that invokes the kernel as shown in Fig. 2. The wait time on the host should ideally fall within the MTBF.

2.2 Synchronisation Within Kernels

The approach presented so far works for kernels that do not use explicit synchronisation because explicit synchronisation breaks the proposed approach. The reason for this is that threads within blocks are not necessarily scheduled all at the same time. Threads within blocks are split in warps and if a warp is stalled for any reason the scheduler is free to replace it with another warp. Consider the case of a kernel with explicit synchronization, and after the first warp reaches the synchronization point, an interrupt occurs. In this case all the other warps will skip their executions, but threads from the first warp will never leave synchronisation point, resulting in a hung kernel.

Our solution to this problem is to terminate the entire block when we receive an interrupt from the host. To do this, the value of `timeout` is only read once by thread zero of warp zero and stored in a variable local to the block. A block-level synchronisation point is set immediately after the read of `timeout` so that no warps can proceed until `timeout` has been read. After all warps have synchronised, the block-local variable is checked to determine whether or not the block should be executed.

A final note about synchronization, CUDA 9.0 introduced grid and multi-device synchronization along with co-operative groups to the programming model. Co-operative groups extends the CUDA model to organise groups of co-operating threads so that programmers can express the granularity of communicating threads. The approach presented in this paper is unaffected by kernels using co-operative groups because co-operative groups are within blocks and still obey block level synchronization. Our approach cannot work if a kernel performs grid or multi-device synchronization because our mechanism would cause the kernel to hang if some blocks terminate early before reaching the grid synchronization point and other blocks are already in wait of synchronization. In summary, the advantages of this approach are as follows:

1. Only makes sense for long running kernels
2. The kernels must be resource intensive enough to exhaust the parallelism in the GPU
3. Does not work for kernels that perform grid or multi-device synchronization
4. May not be suitable for kernels that consume most of the GPU memory

2.3 Implementation

For the adoption of the proposed approach we introduce an API[2] that facilitates adjustment of applications. The core of the API consists of three macros: `BACKUP_KERNEL_DEF`, `BACKUP_CONTINUE`, `BACKUP_KERNEL_LAUNCH` and a wrapper around `cudaMalloc`. Assuming that an application has one kernel, we replace `cudaMalloc` with its `BACKUP_` version. We adjust kernel definitions as follows:

```
1 __global__ void
2 kernel_name (/* args */){
3     /* kernel body */
4 }
```

```
1 __global__ void
2 BACKUP_KERNEL_DEF (kernel_name,
3                    /* args */){
4     BACKUP_CONTINUE ();
5     /* kernel body */
6 }
```

[2] The API with its documentation and examples can be found at https://bitbucket.org/maxbaird/cuda_backup.

We replace kernel invocation as follows:

```
kernel_name<<<blocks,
              threads>>>
      (/* args */);
```

```
BACKUP_KERNEL_LAUNCH
  (kernel_name, blocks,
   threads, /* args */);
```

When we rewrite a kernel definition, the macros extend the arguments with a boolean mask array to log which threads ran previously, and they insert the code that checks for the interrupts and the code that terminates the entire block of threads if the interrupt has been received. The `BACKUP_cudaMalloc` memory wrapper collects the allocated data structures that the kernels need so that we can transfer them back from the GPU to the host for the purpose of snapshotting. The macro that wraps the launch of the kernel defines a loop that launches the kernel, waits for a certain time, sets the interrupt and transfers the data (captured by `BACKUP_cudaMalloc` wrapper) from the GPU.

For a complete example, please refer to *demo* directory https://goo.gl/BKvcxX where we demonstrate how the proposed API applies to an application with a kernel that adds two vectors. We provide an original code *vecadd-original.cu* and its modified version *vecadd-modified.cu* that uses our API.

Currently, the API is restricted to applications with a single kernel and only provides a wrapper for `cudaMalloc`. Allocations done via `cudaMallocManaged` automatically work because they are managed by the unified memory system which means that data is readily available at snapshot time. Wrappers for the remaining CUDA allocation functions like `cudaMalloc3D` and `cudaMemcpy2D` are missing. These limitations are only of a technical nature and will be fixed in the foreseeable future.

3 Experimental Setup

In order to evaluate the proposed mechanism we use a real-world application, PBOOST [19] and an artificial example. Ideally, it would have been more suitable to use an established benchmarking suite such as the Rodinia benchmarks instead of an artificial example. However, these benchmarks do not run long enough on our hardware to escape measurement noise and provide conclusive results. A simple kernel is best to isolate and measure the sources of overhead. PBOOST is a tool for parallel permutation tests in genome-wide association studies which concern single nucleotide polymorphism pairs and their association with diseases via the combination of their main effects and interactions. On our system PBOOST runs for about 38 min. The modified version of PBOOST can be found at https://goo.gl/84pNQs. All the modifications to the code are implemented using our API.

We also use the artificial kernel in Listing 1.1 to perform an in-depth analysis of the overheads introduced by our mechanism.

```
1 __global__ void
2 kernel(unsigned long long n, unsigned long long *res){
3   unsigned long long x = 0;
4   for(unsigned long long i = 0; i < n; i++){
5     x++;
6   }
7   *res = x;
8 }
```

Listing 1.1. Artificial kernel for evaluating overheads

We deliberately choose such a trivial kernel, so that we get 100% occupancy on the GPU. All experiments are performed on a AMD Opteron 6376 system with four sockets running Scientific Linux Release 7.4 (Nitrogen), kernel version 3.10.0. The system is fitted with 512 GB of RAM, running at 800 MHz and an NVIDIA TITAN-XP GPU, which is connected via PCIe x16. The TITAN-XP can execute 61,440 threads simultaneously using its 30 streaming multiprocessors (SM). For our experiments we use CUDA 9.0 with a driver version 384.81. Each application is executed 10 times to eliminate measurement noise. We report the average execution time and 90% confidence intervals. In addition, we note kernel configurations in the tripple-chevron CUDA notation: <<<`blocks, threads`>>>; where `blocks` represents the number of blocks of threads and `threads` represents the number of threads per block.

4 Evaluation

Figure 3 shows the execution times of PBOOST with an increasing number of snapshots. The application runs for approximately 38 minutes and the variation of execution time is within 5 s (0.2%) when performing 0 to 6 interrupts. We see that in this particular example, the execution time of the application with our mechanism enabled is the same as that of the vanilla version within the measurement error.

Despite such a low overhead looking very promising, this result is not conclusive. In order to understand the nature of the overheads that our mechanism really brings, we study them in isolation. We investigate how expensive is it to do:

1. Conditional checks in each thread
2. Soft interrupts of a kernel
3. Memory transfers

For these experiments we will use the kernel from Listing. 1.1.

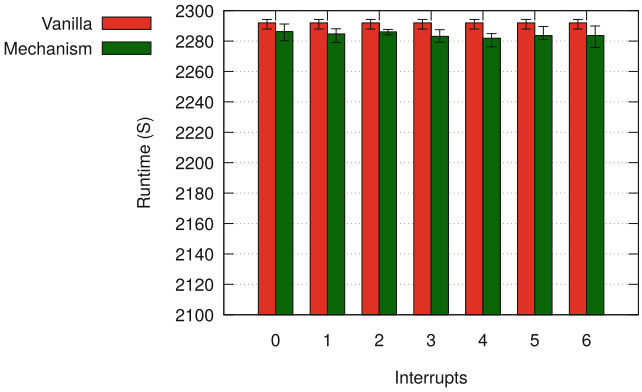
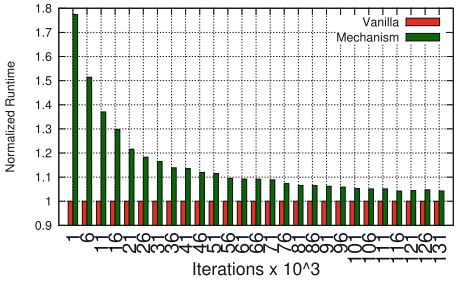


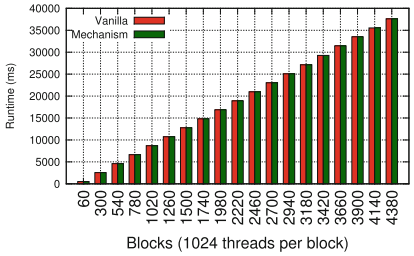
Fig. 3. Absolute runtimes of modified application

4.1 Conditional Checks Overheads

Every thread needs to perform two conditional checks (interrupt and did it already execute) to determine whether to continue or not, so the goal of this experiment is to determine the cost of having these extra checks. To avoid also measuring any overhead that may come from the GPU scheduler, we use a kernel configuration that matches the number of simultaneous threads that can be executed by the GPU. As the TITAN-XP can execute 61,440 threads simultaneously, we use a configuration of `<<<60, 1024>>>`. The results in Fig. 4a show that the conditional checks is significant if number of operations, and by extension runtime, is very small. However this becomes irrelevant for the cases, we are interested in. Figure 4b shows that the overhead remains minimal for much larger values of n.



(a) Mechanism normalized to vanilla

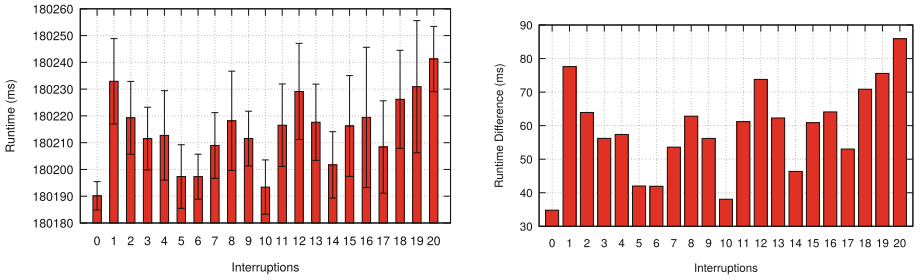


(b) Vanilla vs Mechanism. $n = 67 \times 10^6$

Fig. 4. Overhead of conditional check

4.2 Interrupt Overheads

In order to examine this overhead, we performed set $n = 1.6 \times 10^9$ with a kernel configuration of <<<1320, 1024>>>. This configuration is needed to be able to interrupt the kernel at least 20 times. To be able to interrupt a kernel N times, at least $(N + 1) \times s \times t$ threads are required, where s is the number of SMs and t is the maximum number of threads an SM can execute. No memory transfers were made. Figure 5a shows the absolute runtime for each interrupt and Fig. 5b shows the time each interrupt adds to the vanilla execution. We see that for a runtime of approximately 3 min, each interrupt adds between 34 ms to 85 ms; the variability of which can be attributed to measurement noise.



(a) Interrupt Runtime. $n = 1.6 \times 10^9$. (b) Added time per interrupt

Fig. 5. Overheads of soft interrupts. Kernel configuration <<<1320, 1024>>>

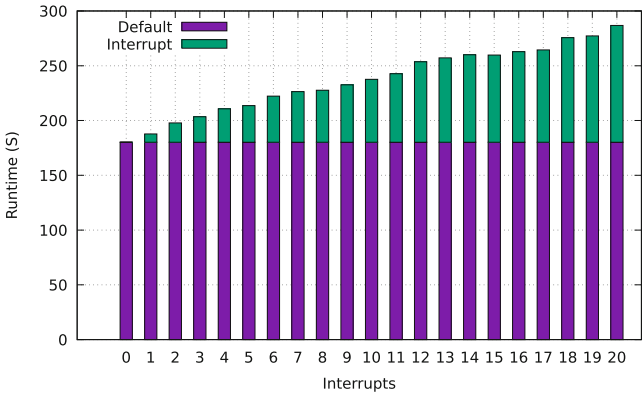


Fig. 6. $n = 1.6 \times 10^9$. Kernel configuration <<<1320, 1024>>>. Single memory copy of 11.9 GB made at each interrupt

4.3 Data Transfer Overheads

To measure this overhead, we set $n = 1.6 \times 10^9$ with 98.3% (11.9 GB) of GPU memory allocated. At each interrupt, all GPU allocated memory is transferred back to the device. The results in Fig. 6 show that having to perform large memory transfers at each interrupt noticeably increases the overhead.

The first two experiments show that practical overheads of checking a conditional or doing a software interrupt are close to zero. Memory overheads on the other head, can be quite expensive, but we did not observe them in PBOOST. The reason for this is that despite the kernel runs for such a long time, it only uses 213 MB of the GPU memory, which can be copied very quickly.

5 Related Work

CheCUDA [16] and NVCR [10] are presented as checkpoint/restart tools both of which take a similar approach to GPU fault tolerance. The former works by hooking into basic CUDA driver API calls to record status changes on the device, writing those changes into a file at checkpoint time and using this file to re-initialise the device at restart. NVCR works in a similar way but deletes all CUDA resources before checkpointing and restores them right after checkpointing. Both approaches need the kernel to run to completion and depend on the CUDA runtime to automatically detach itself from the running process and destroy its context. Unfortunately the CUDA runtime stopped doing this from version 3.2 and onward when support for 64-bit device side memory space was added. This is a problem because an existing context at checkpoint time will have its information captured. Restarting an application with this information will fail because the context is no longer attached to the device.

A possible way to circumvent the limitation of CUDA's runtime remaining attached would be to mimic the approach taken by CheCL [17]. CheCL is implemented in the context of OpenCL and transparently provides checkpointing capabilities by substituting the OpenCL shared library with its own version. This allows CheCL to decouple the process from the OpenCL runtime by forwarding all API calls to a proxy process that executes the real API function.

Virtual machines (VMs) are a viable option to achieve both fault tolerance and process migration. Along these lines vCuda [9] and GVIM [8] are proposed as a GPGPU computing solution for applications running on VMs. The advantage of a VM is that it inherently decouples the application from the GPU hardware interface thus simplifying the checkpoint step. This means that API calls need to be intercepted and redirected to the guest OS resulting in large communication overheads and performance degradation.

CudaCR [14] and VOCL [12] are presented as schemes for soft error recovery for GPUs and coprocessors respectively. CudaCR captures the GPU state within the kernel to be able to roll back to a previous state if a soft error occurs. VOCL provides a transparent virtualization layer between applications and the OpenCL runtime. This allows the capture of API calls so that they can be replayed if a soft error occurs. It is worthy to note that CudaCR does address soft errors for long

running kernels, however in contrast to our work, neither of these approaches specifically target hardware failures. Approaches mentioned in this section need to wait for GPU kernels to complete before taking a checkpoint; to the best of our knowledge none consider the case of a long-running kernel.

6 Conclusions and Future Work

This paper proposes a solution to the problem of checkpointing applications that use GPU kernels. We present a mechanism that periodically captures a state of the running GPU kernels. With such a mechanism in place, we can use any existing checkpointing system to make snapshots of an application, while a GPU kernel is still running. It is guaranteed by the construction of our mechanism that any such a snapshot captures enough of a state to safely restore the application.

The key insight of this approach lies in the observation that not all the threads of the kernel start at the same time. Such a delay makes it possible to instrument the thread to check for the interrupt and terminate voluntarily if the interrupt has been received. As we have demonstrated on a real-world and synthetic examples, the runtime overhead of the proposed mechanism is very small. We have implemented a library with a compact API which makes our approach straight-forwardly applicable to existing applications. The implementation is freely available at BitBucket.

The effectiveness of the proposed approach enables several future directions of research. First of all, the straightforward nature of our API suggests an automated instrumentation of GPU kernels should be easily possible. Secondly, we would like to integrate our approach with an existing checkpointing system. All we need to do is to make sure that the system makes a snapshot at the time when we captured the state of a kernel ("Checkpoint" stage in Fig. 2b). The checkpointing system could also set or change the time we wait after the kernel launch, so that the snapshotting frequency could be altered.

As our experiments show, the amount of data that is transferred to enable checkpointing dominates the overall overhead. The amount of data that we currently copy at every interrupt/kernel restart is a conservative over approximation. The blocks of results that have been computed do not need to be copied to the GPU. However, figuring out whether it is safe to copy data partially is far from trivial. Despite being inspired by the needs of resilience, our interrupt mechanism has further uses. Fail early scenarios can use our approach so that GPUs return as soon as possible if the system has already started failing and a rollback has to occur. It can also be used for fault injection testing on GPUs which is difficult if kernels run to completion.

Acknowledgements. This work was supported in part by grants EP/N028201/1 and EP/L00058X/1 from the Engineering and Physical Sciences Research Council (EPSRC) as well as the James Watt Scholarship of Heriot-Watt University.

References

1. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters. J. Phys. Conf. Ser. **46**, 494–499 (2006). https://doi.org/10.1088/1742-6596/46/1/067
2. Gõddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P.: Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Comput. **33**(10–11), 685–699 (2007). https://doi.org/10.1016/j.parco.2007.09.002
3. Egwutuoha, I.P., Levy, D., Selic, B., Chen, S.: A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. J. Supercomput. **65**(3), 1302–1326 (2013). https://doi.org/10.1007/s11227-013-0884-0
4. Bautista-Gomez, L., Tsuboi, S., et al.: FTI: high performance fault tolerance interface for hybrid systems. In: 2011 International Conference for High Performance Computing. IEEE (2011). https://doi.org/10.1145/2063384.2063427
5. Cappello, F., Geist, A., et al.: Toward exascale resilience. Int. J. High Perform. Comput. Appl. **23**(4), 374–388 (2009). https://doi.org/10.1177/1094342009347767
6. DeBardeleben, N., et al.: GPU behavior on a large HPC cluster. In: an Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 680–689. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54420-0_66
7. Fan, Z., Qiu, F., et al.: GPU cluster for high performance computing. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC 2004, p. 47 (2004). https://doi.org/10.1109/SC.2004.26
8. Gupta, V., et al.: GViM: GPU-accelerated virtual machines. In: Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing, HPCVirt 2009, pp. 17–24. ACM (2009). https://doi.org/10.1145/1519138.1519141
9. Shi, L., Chen, H., et al.: vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans. Comput. **61**(6), 804–816 (2009). https://doi.org/10.1109/IPDPS.2009.5161020
10. Nukada, A., et al.: NVCR: a transparent checkpoint-restart library for NVIDIA CUDA. In: 2011 IEEE IPDPS Workshops and Phd Forum, pp. 104–113. IEEE (2011). https://doi.org/10.1109/IPDPS.2011.131
11. NVIDIA: CUDA C programming guide (2017)
12. Peña, A.J., Bland, W., Balaji, P.: VOCL-FT: introducing techniques for efficient soft error coprocessor recovery. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, pp. 1–12. IEEE (2015). https://doi.org/10.1145/2807591.2807640
13. Phillips, J.C., et al.: Adapting a message-driven parallel application to GPU-accelerated clusters. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008. IEEE (2008). https://doi.org/10.1109/SC.2008.5214716
14. Pourghassemi, B., et al.: CudaCR: an in-kernel application-level checkpoint/restart scheme for CUDA-enabled GPUs. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 725–732. IEEE (2017). https://doi.org/10.1109/CLUSTER.2017.100
15. Showerman, M., et al.: QP: a heterogeneous multi-accelerator cluster. In: 10th LCI International Conference on High-Performance Clustered Computing (2009)
16. Takizawa, H., et al.: CheCUDA: a checkpoint/restart tool for CUDA applications. In: 2009 International Conference on PDCAT, pp. 408–413. IEEE (2009). https://doi.org/10.1109/PDCAT.2009.78

17. Takizawa, H., et al.: CheCL: transparent checkpointing and process migration of OpenCL applications. In: 2011 IEEE International IPDPS. IEEE (2011). https://doi.org/10.1109/IPDPS.2011.85
18. Mohamed, H., Osipyan, H., Marchand-Maillet, S.: Multi-core (CPU and GPU) for permutation-based indexing. In: Traina, A.J.M., Traina, C., Cordeiro, R.L.F. (eds.) SISAP 2014. LNCS, vol. 8821, pp. 277–288. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11988-5_26
19. Yang, G., et al.: PBOOST: a GPU-based tool for parallel permutation tests in genome-wide association studies. Bioinformatics **31**(9), 1460–1462 (2015). https://doi.org/10.1093/bioinformatics/btu840