Checkpointing Kernel Executions of MPI+CUDA Applications

Max Baird✉[1], Sven-Bodo Scholz[1], Artjoms Šinkarovs[1], and Leonardo Bautista-Gomez[2]

[1] Heriot-Watt University, Edinburgh, UK {mmb1,s.scholz,a.sinkarovs}@hw.ac.uk
[2] Barcelona Supercomputing Center, Barcelona, Spain leonardo.bautista@bsc.es

Abstract. This paper proposes a new approach to checkpointing MPI-CUDA applications based on the idea of the state of the art High Performance Fault Tolerance Interface FTI. It extends FTI to include support for checkpointing data that resides on GPUs and it provides a mechanism to enable checkpointing during kernel executions as well. Jointly, these extensions ensure that checkpointing of MPI-CUDA applications can be seamlessly done with minimal program modifications, even if most of the application time is being spent on GPUs of HPC clusters rather than the CPU-based host nodes themselves.
The paper provides a description of how such a checkpointing of collaborative MPI-CUDA kernels as well as application restarts from those checkpoints can be achieved. Based on a full-fledged, openly available implementation in the context of FTI, the paper also provides some initial evaluations using the well-known Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) application as a case study.

Keywords: HPC, MPI, GPU, Snapshots, Checkpoints, Resilience.

1 Introduction

While increasing number of scientific applications use GPUs, demonstrating impressive speedups due to massive parallelism of the architecture, the problem of resilience for such applications significantly increases. It is a common knowledge, that checkpointing GPU accelerated applications require synchronization around kernel execution. GPUs have an uninterruptible execution model, so there is no universal way to fetch data from the GPU while a kernel is running. Therefore, snapshots of an application with long-running kernels can be taken only between the kernel launches. In case these intervals become larger than mean time between failure (MTBF) of the underlying system, checkpointing for all practical purposes becomes meaningless. This problem is intensified in the context of MPI, as kernel launches may happen asynchronously.

In this paper we propose a solution to this problem: we implement a system that makes it possible to interrupt GPU-accelerated MPI applications without waiting for kernels to complete. In our previous work on soft kernel interrupts [1] we show that with a small rewrite, it becomes possible to request a kernel to

return before continuing its intended work. Due to physical limitations of a GPU, not all the threads of a kernel start at the same time. Instead, they are scheduled in blocks, and each block can check whether to progress or return. With this observation, we derived a mechanism to automatically rewrite kernels making them "interruptable"[3].

In order to make this mechanism practically useful, we leverage its power in the context of a checkpoint/restart tool. We extend a state of the art framework named Fault Tolerance Interface (FTI) [2] that has been rigorously tested on supercomputer clusters with real-world scientific MPI applications. FTI exposes a simple API for the application programmer to indicate what data should be checkpointed at what time. The saving and restoring functionality is then provided by the framework.

Our first extension to FTI adds the ability to checkpoint data which resides on the GPU. This allows one to fetch data from GPUs as a part of the FTI functionality, but it did not solve the necessity to wait till kernel completion problem.

This is addressed with our second extension which integrates our soft kernel interrupt technique into FTI. This allows one to mark which GPU kernels should become interruptable. When FTI decides to make a snapshot, and part of the data resides on the GPU, these marked kernels will be interrupted. This happened to be the most challenging part of this work. First, we had to respect the FTI assumption on the topology of the MPI processes that ensures parallel snapshotting. Secondly, we had to respect the separation between the snapshotted data and metadata. The former is often analysed by application scientists whereas the latter is only used at restart time. Finally, we had to figure out how much metadata is sufficient to restore the state of the GPU kernel in case it was interrupted.

The viability of our approach is demonstrated at the example of applying extended FTI library to LULESH [9] (Livermore Unstructured Lagrange Explicit Shock Hydrodynamics), a widely studied application which models the motion of materials relative to each other when subject to forces. We demonstrate that our system can handle a large scientific CUDA/MPI application; and, most importantly, that the minimal snapshotting interval of the application does not depend on the entire kernel runtime. That is we can take a snapshot without waiting for kernels to complete. With this achievement the snapshot interval is effectively reduced since checkpoints are now possible during kernel execution.

The individual contributions of this paper are:

- extend FTI to checkpoint GPU data;
- extend FTI to mark kernels so that they could be automatically interrupted during snapshots;
- demonstrate and verify the approach at the example of a large real world MPI/CUDA scientific application.

[3] Available at `https://bitbucket.org/maxbaird/cuda_backup`

2 Interrupting A Kernel

In this section we briefly introduce details on CUDA model, the problem of interrupting a running GPU kernel and our solution to it. CUDA is a parallel computing platform and programming model for general purpose GPUs developed by NVIDIA. It provides extensions to C/C++ and Fortran which allow programmers to define special functions called *kernels*. These kernels can be configured to execute in parallel by N different CUDA *threads*. Each such a thread runs on a separate GPU core. At runtime, CUDA threads are partitioned into groups called *blocks*. As GPU is a coprocessor to the main system, each application can be naturally divided into *host* part and *device* part. The *host* code runs on a CPU and *device* code on the GPU.

Checkpointing a GPU kernel comes with two well-known problems: saving/restoring the GPU context and interrupting a long-running kernel. The CUDA runtime implicitly creates an underlying context for communication between the host process and device. Once created, the context remains attached to the host process for its lifetime. If a process is checkpointed while maintaining an active GPU context, restart from that checkpoint will fail because the restored context will no longer be valid at the device. As CUDA does not provide technical information or an API for context management and FTI does not preserve process states, this work is not concerned with the GPU context save/restore problem. FTI requires the application programmer to indicate what data needs checkpointing and upon restart, the data is automatically restored from the last checkpoint.

CUDA does not expose an API to send an interrupt to a running GPU thread. Nevertheless, threads can be instrumented to interrupt themselves; this is done by ensuring that the first step of a thread's execution is to check a host-controlled flag for permission to continue. All threads within a block wait until an appointed thread reads and updates a block local variable with the flag's value. Afterwards, all threads proceed to make the conditional check whether to continue using the block local variable. Executed blocks are kept track of via a boolean array so that when the kernel is resumed only unexecuted blocks proceed. The kernel's argument list is adjusted to include the boolean flag and array.

After such a modification a kernel can be interrupted by setting the flag on. This signals the kernel to return as early as possible. After the kernel returns, the boolean array of executed blocks is examined. If all blocks have executed then the kernel is complete; otherwise, the kernel will be relaunched. For more details refer to [1].

Note that this approach does not work for kernels with explicit intra-block synchronisation.

3 FTI

FTI is a multilevel checkpointing library for large scale supercomputers. At extreme scale, supercomputers suffer from frequent failures due to the increased

number of components. As scientific applications grow in scale, they are more prone to failures forcing to restart the execution. At the same time, they also use more data, and therefore the state to be saved upon a checkpoint is also increasing. This leads to an I/O bottleneck that could render scientific applications unable to make progress. To alleviate this problem, FTI makes use of multiple storage levels, including the global parallel file system (GPFS), as well as local storage inside the compute nodes. In particular FTI has four levels of checkpointing, providing a good trade-off between resilience and performance

In addition, to the multilevel aspect of checkpointing with FTI, the library also provides a feature to make use of spare cores for resilience aspects. This is particularly useful in systems with accelerators like GPUs, where each GPU is usually linked to one CPU core, and the other CPU cores are idle. This is often the case with GPU applications that execute the most compute intensive parts on the GPU leaving no work for the CPUs. Thus, FTI makes use of the idle CPU cores to accelerate data transfer to the GPFS as well as other resiliency tasks.

All the complexity of erasure coding, asynchronous transfer and managing multiple storage levels is hidden by FTI behind a simple interface that can be summarized in only four functions:

- `FTI_Init`: This function initializes FTI with the configuration provided by the user in the configuration file.
- `FTI_Protect`: This function is used to tell to FTI which are the variables that need to be checkpointed.
- `FTI_Snapshot`: This function actually takes the checkpoint according to the frequency provided in the configuration file.
- `FTI_Finalize`: This function frees the memory and clean up the different storage levels.

FTI also includes a few more function for low-level control of the checkpoints and for checkpointing on specific formats such as HDF5, among others. While running GPU applications with long-executing kernels there are other additional changes that need to be done to perform intra-kernel checkpointing. This will be described in the next section.

4 Extending FTI

In this section we describe both of our extensions to FTI and demonstrate what happens when applying the extended library to an MPI application with GPU kernels.

As FTI is only capable of snapshotting data residing on the host, our first extension adds support for checkpointing GPU data. This extension is orthogonal to kernel interruption, as it liberates a programmer from manually copying device data to the host before each checkpoint. When the extension is used on its own, checkpoints have to occur outside of kernel executions.

When checkpointing with FTI, its API prescribes to pass a pointer reference to the data to be checkpointed. Therefore, handling data residing on the GPU implies

determining whether the reference is a valid host or device pointer. Conveniently, the CUDA API provides the `cudaPointerGetAttributes` function which makes it possible to distinguish host and device pointers. For device pointers, a device to host transfer is made prior making a snapshot, and correspondingly on restart, the data is copied back to the device. Note that Unified Virtual Addressing (UVA) and Unified Memory (UM) introduced in CUDA versions 4.0 and 6.0 correspondingly, present a programmer with a coherent view of host and device memory [12], which in principle should eliminate the need to distinguish between host and device pointers. No special handling is required for coherent memory spaces as their pointer attributes will indicate them to be host accessible. FTI will checkpoint the data as normal with the driver instead performing automatic transfers from the GPU.

4.1 Adding Kernel Suspension To FTI

Our second extension adds the ability to perform checkpoints during kernel execution. This is achieved by rewriting the kernel so that it becomes interruptible.

Implementation of this feature comes with a two of challenges. Firstly, apart from the annotated data, at each checkpoint FTI saves some metadata that is used during the restore. Per convention, metadata is stored separately enabling easier analysis of the actual data. This convention has to be respected when saving data used to recreate the GPU kernels. Thirdly, per FTI convention, the number of MPI processes launched in every application must be a multiple of the number of nodes being used. This requirement has to do with how FTI has been designed to recover from failures. For each group there is a designated head process to which all other group processes must send their metadata when it is time to checkpoint. Secondly, every process within a group can launch multiple GPU kernels.

To ensure an interrupted kernel is correctly restarts it is necessary to track which blocks have executed. This information constitutes the metadata of an interruptible kernel. When it is time to checkpoint, the metadata of every interruptible kernel needs to be transferred to the host process. Then the host process needs to send the data of each kernel to the head of its group, the group head is then responsible for determining at which level the checkpoint is being made where it finally persists the metadata. The restart process is more straightforward and loads all of the checkpoint metadata at each level and each process is told at what level it should retrieve the metadata.

Extending FTI to interrupt kernels is essentially a matter of exposing 3 simple macros via the API to rewrite the kernels as necessary. The API macros are as follows:

1. `FTI_Protect_Kernel`
2. `FTI_Kernel_Def`
3. `FTI_Continue_Check`

`FTI_Protect_Kernel` is responsible for rewriting the kernel so that it is continuously launched and interrupted until all blocks have executed. Two key

arguments required by this macro is a unique ID for the kernel, and the interval of time to wait before issuing an interrupt signal. The ID is useful for FTI to internally manage and restore the kernel's metadata. Two additional arguments are added to the kernel's argument list; the boolean flag for signalling the kernel and the boolean array for keeping track of executed blocks. `FTI_Kernel_Def` rewrites the kernel's definition to add corresponding parameters for the signal and boolean array. Lastly, `FTI_Continue_check` adds code to check the boolean flag to determine whether the kernel should continue. It also updates the boolean array with the block ID if the block is allowed to proceed.

How Kernels Are Protected This section highlights the main steps of protecting a kernel. A protected kernel is a kernel that can be interrupted during its execution so that a checkpoint can be made. A description of the macros is provided in Section 4.1 The primary purpose of `FTI_Protect_Kernel` is to initialize the kernel's metadata and to repeatedly execute the kernel until all blocks are complete.

Initialization When `FTI_Protect_Kernel` rewrites a kernel launch, it first makes a call to `FTI_kernel_init` which initializes an object of type `FTIT_KernelInfo` with information on how to interrupt, checkpoint and restart the kernel. For efficiency, a kernel's metadata is initialized once and cleared and reused as necessary if the kernel with same ID is launched again. The initialisation call is made irrespective of normal application execution or failure, if a kernel has associated metadata, this metadata will be restored.

Execution The step after initialisation wraps the kernel launch in a loop which only terminates after it has been executed by all MPI processes. For every iteration, each process broadcasts their complete status to all other processes. The launch loop's terminating condition is orchestrated this way to prevent the application from hanging during a checkpoint. The process for FTI to make a checkpoint includes calling an MPI collective function which requires the participation of *all* processes. If some processes were to complete their kernels early, and allowed to leave the launch loop, other processes creating a checkpoint while within the loop would wait indefinitely.

Next, the kernel is launched asynchronously and the host sleeps for the duration of the quantum. When the quantum expires, the host sends a request to the kernel and waits for it to return. After the kernel returns, a count of its executed blocks is made, if the sum is equal to the number of blocks launched the kernel is marked as complete. The number of blocks executed and the kernel's complete status metadata is then updated. Finally, a snapshot is attempted, and based on the configured snapshot frequency a snapshot may or may not be made. The kernel will be relaunched until it is complete.

4.2 What Happens At Checkpoint Time

It is the task of a programmer to decide what application data needs protecting for a successful restart. Therefore, we are mainly concerned with metadata generation, including the kernel one. For each group of processes, the process with group ID zero is identified as the head process. The head process is responsible for accumulating the metadata from all other group processes via standard `MPI_Send` and `MPI_Recv` functions. The transfer of kernel metadata as listed in Section 4.1 is accomplished in the same manner. After receiving the metadata, the head process proceeds to write the information to storage at the matching checkpoint level.

4.3 What Happens At Restart Time

A previously failed application may be restored if at least one checkpoint was successful prior to failure. When executed, FTI will detect the execution as a restart and try to recover the most recent checkpoint data. The corresponding metadata of the recovered checkpoint is also loaded as part of the restart. The initialization phase of FTI triggers the restart process and subsequently calls a setup function for kernel protection. If the setup function detects the application is in recovery mode it attempts to load the metadata for all protected kernels. For an interruptible kernel, the `FTI_Protect_Kernel` macro will rewrite the kernel launch as described, this time however, the kernel's associated metadata will be restored instead of newly allocated. The restored metadata contains the list of executed blocks which the kernel uses to accurately resume.

A previously complete kernel will have its metadata reset so that it can be launched again. However, If there are multiple kernels to be restored, a check is performed to ensure that *all* protected kernels are complete. Since at this point, whether the kernel is being relaunched immediately after failure or again through iteration cannot be determined. For the former case, execution must resume from the incomplete kernel. For the latter case, complete kernels that are not reset will still launch but do nothing since all blocks are marked as complete.

5 Experimental Setup

From LULESH we used a kernel that is called once for each iteration of LULESH's main executing loop. For our experiments only level 1 checkpoints were permitted. The other levels were effectively disabled by configuring their interval to be greater than longest running experiment, this was done as the time taken for checkpointing is not consistent for each level. First we establish then experimentally verify a simple model to determine the frequency at which kernels may be interrupted. Next, we experiment with a simple LULESH configuration of 1 MPI process to determine if kernel interruptions fit our model by triggering interrupts as often

as possible. Finally, we use a real world configuration[4] of multiple MPI processes with heavier application workloads and a practical checkpoint interval.

All experiments were executed using an AMD Opteron 6376 CPU running Scientific Linux Release 7.6 (Nitrogen), kernel version 3.10.0. The system has 1024GB of RAM and an NVIDIA TITAN-XP GPU with 12GB of global memory connected via PCIE x16. For our experiments CUDA version 10.0 was used with driver version 410.79.

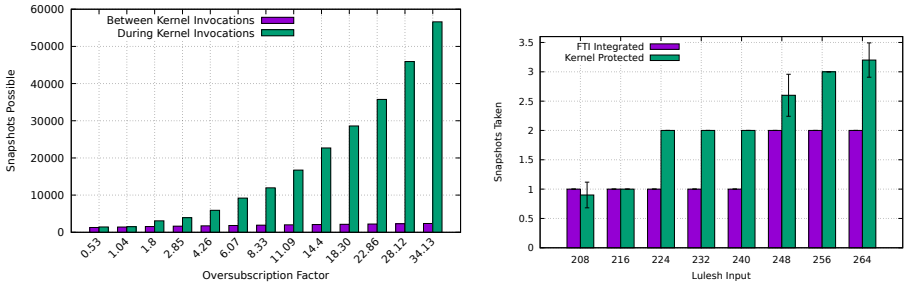
6 Case Study

The primary control flow of LULESH consists of a loop which drives all kernel invocations. For our case study we evaluate three versions of LULESH, a native version with no changes, a version with the FTI library and a version with FTI extended to interrupt kernels. A functionality test is first performed to ensure that LULESH still correctly runs to completion after an interrupt as been triggered from within a kernel. For this test, the native version of LULESH was configured and executed so that its solution was written to a file. The kernel interruptible version of LULESH was then executed with the same configuration and prematurely terminated after at least one checkpoint. It was verified to have restarted correctly via a checksum comparsion against its solution files and the native's solution files.

The goal of our evaluation is to verify that FTI extended with ability to interrupt kernels solves the problem of checkpointing an application with long-running kernels. For the evaluation, the number of possible snapshots is counted and each snapshot is verified to break the kernel after a number of executed blocks. The extension makes it possible to interrupt earlier which minimises the snapshot interval. Our first experiment verifies the simple model we use to interrupt kernels. The second experiment verifies our approach works on a simple configuration of LULESH and the third experiment demonstrates the validity of our approach on a realistic configuration of LULESH. The details of these experiments are presented in this section.

There is a simple model that should indicate how many kernel interrupts are possible due to there being a limit to the number of threads a GPU can simultaneously execute. If a kernel is configured to exceed this limit then the GPU will be oversubscribed. An over-subscription factor of a GPU can be determined by: $\frac{K_n}{N_{gpu}}$, where K_n is the desired number of threads and N_{gpu} is the number of threads the GPU can simultaneously execute. Using this model, we were able to confirm experimentally that the over-subscription factor has a direct influence on the rate of interrupts which decreases the snapshot interval.

[4] `https://github.com/maxbaird/luleshMultiGPU_MPI/tree/integrating-fti-protecting-kernels`



(a) Interrupts Increase with Oversubscription (b) Realistic execution

Fig. 1: Reducing Checkpoint Interval

Figure 1a is the result of the model applied to a simple configuration of LULESH and clearly shows the total number of application snapshots possible increases with the over-subscription factor. The significant increase in snapshots from the application of our mechanism is attributed to the kernel being launched over 1000 times for each data point. While many more checkpoints are possible, the results do not precisely fit the simple model for a couple reasons. Firstly, the quantum needs to be increased between interrupts since previously executed blocks will still be scheduled as normal. Although these blocks will terminate early, they still consume time to perform the termination check. If the quantum is not increased, very few new thread blocks will be scheduled and the kernel will progress very slowly until all unexecuted blocks are randomly scheduled early. The amount the quantum should be increased by very much depends on the duration of N_{gpu}. The quantum should be increased enough to allow for scheduling as many new blocks as possible, but not too large so that few or no snapshots are made. Secondly, all kernels in LULESH are launched asynchronously in their own streams, this means that multiple kernels execute simultaneously on the device which means N_{gpu} represents threads from multiple kernels. In this case, it is possible to achieve more interrupts of individual kernels since a single kernel will not dominate the device. For these two reasons, our model needs to be more complicated but we leave this as future work. Despite this, we demonstrate here that the proposed approach definitely works and clearly improves the problem described in Section 2. The additional memory required for the book keeping of executed blocks does not significantly increase the application's memory footprint. It is less than 3MB for our most oversubscribed execution.

Figure 1 applies FTI with our extension to a full application of LULESH with a snapshot interval of four minutes. The value for the interrupt is represents a more realistic execution of LULESH and does not try to interrupt the application as often as possible, so the results for this figure are less dramatic than that of Figure 1a.

7 Related Work

Checkpoint/Restart (CR) techniques are generally done either through proxy, virtualisation, or can be specifically applied to applications. The work discussed here is thus grouped accordingly.

GPU Proxies CRUM [4] achieves transparent CR by using a proxy process to decouple the application process state from the device driver state. This allows for checkpoints to be made without recording any active driver state. CRUM is geared toward applications with large memory footprints which make use of Unified Virtual Memory (UVM). The proxy process creates a shadow UVM region for each allocation made by the application process and then makes a corresponding real allocation via the CUDA driver. The shadow and real memory regions are kept in sync. This setup is necessary because UVM has no API calls that can be intercepted. However, the restart process is based on the assumption of deterministic memory allocations that are made by the CUDA driver libraries. The CUDA documentation does not guarantee this to always be the case. It also raises the question of what happens if a restart needs to occur on a different device; while the allocations may be deterministic it does not mean they are consistent across devices. CRCUDA [17] and CheCL [19] are proxy based approaches that target CUDA and OpenCL respectively. Like CRUM, CRCUDA is transparent to the application process. Unlike CRUM, CRCUDA does not rely on deterministic memory allocations but instead logs and replays CUDA API calls where BLCR [7] is responsible for saving and restoring the application's state. CRCUDA does not support MPI or applications that make use of UVM. CheCL provdes its own OpenCL library to intercept and redirect API calls to decouple the process from the OpenCL runtime.

GPU Virtualisation Virtualising GPUs is quite common as exemplified by work such as [3] [13] [6] [5]. Virtual Machines (VMs) are attractive as they inherently serve as a buffer between the application and the physical device. This decoupling from the hardware makes things easy for checkpointing especially in the realm of CUDA where the GPU context cannot be checkpointed along with the application. A VM approach is similar to a proxy in that API calls need to be redirected. VMGL [10] is marketed as an OpenGL cross-platform GPU independent virtualisation solution with suspend and resume capabilities. Suspend and resume is made possible via a shadow driver which keeps track of OpenGL's context state. While OpenGL is supported by all GPU vendors, in reality it is used chiefly for rendering and not well suited for general purpose GPU computing. vCUDA [16] follows the identical approach of VMGL using CUDA instead of OpenGL. Unfortunately, VMs typically add more overhead via extra communication ultimately degrading performance.

Application Specific CheCUDA [18] and NVCR [11] are currently obsolete CUDA based libraries because they depend on the CUDA context detaching cleanly before a checkpoint. Unfortunately, the CUDA context no longer behaves this way

and is non-reentrant. A restart is not possible if the GPU context was recorded in the checkpoint. CudaCR [15] is a CR library that is capable of capturing and rolling back the state within kernels in the event of a soft error. Similarly for soft errors, VOCL-FT [14] offers resilience against silent data corruption for OpenCL-accelerated applications. VOCL-FT is a library that virtualises the layer between the application and the accelerators to log commands issued to OpenCL so that they may be replayed later in case of failure. HiAL-Ckpt [20] is a checkpointing tool for the Brook+ language with directives to indicate where checkpoints should be made. However the development on Brook+ seems to have stopped with the last official release in 2004. HeteroCheckpoint [8] is a CUDA library and mainly focuses on how efficient checkpoints can be made by optimising the transfer of device data.

8 Conclusions & Future Work

This paper demonstrates a system that makes it possible to checkpoint/restore MPI applications with long running GPU kernels. Its distinctive feature is the ability to take snapshots without the necessity to wait for kernels completion. To our knowledge, none of the existing resilience tools can do this automatically.

The system is based on the FTI library — one of the standard resilience tools; and it is extended with the kernel interruption mechanism that we have described in [1]. As a result, by using the proposed tool, we significantly reduce the minimal interval at which the snapshots can be taken, making it possible to align the snapshot frequency with the MTBF of the system of interest.

We apply our system to the real-word numerical MPI/CUDA application named LULESH. We verify that the proposed system is operational by running a number of snapshot/restores that include GPU data; and we demonstrate that the minimal snapshotting interval actually decreases.

Despite our system being fully operational and production-ready, it comes with a few limitations that immediately guide our future work. Currently, we do not verify that automatic kernel interruption mechanism is safe, assuming that this is a job of a programmer. For example, if a kernel uses explicit intra-block synchronisation, our mechanism may introduce a deadlock. This is less of a problem for CUDA systems prior to the version 9, as intra-block synchronisation was not supported, and use of manual spinlocks are not advised by the manual. Latest CUDA architectures allow for such synchronisations which we would like to attempt to detect by means of analysing CUDA kernels.

Currently, the time we have to wait to interrupt the running kernel is equal to the time it takes to execute one thread of a kernel. If this time happens to be too large, we need to make our interruption mechanism smarter — we can check for interrupts not only at the beginning of each block, but also while the thread is running. This would require a more sophisticated analysis of kernels, that would take into account dataflow and controlflow.

9 Acknowledgement

This work was supported in part by grants EP/N028201/1 and EP/L00058X/1 from the Engineering and Physical Sciences Research Council (EPSRC).

References

1. Baird, M., Fensch, C., Scholz, S., Šinkarovs, A.: A lightweight approach to gpu resilience. In: Euro-Par 2018. pp. 826–838. Springer (2018)
2. Bautista-Gomez, L., Tsuboi, S., et al.: Fti: High performance fault tolerance interface for hybrid systems. In: SC '11. pp. 1–12 (2011). https://doi.org/10.1145/2063384.2063427
3. Duato, J., Peña, A.J., et al.: rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In: 2010 International Conference on High Performance Computing Simulation. pp. 224–231 (2010). https://doi.org/10.1109/HPCS.2010.5547126
4. Garg, R., Mohan, A., et al.: Crum: Checkpoint-restart support for cuda's unified memory. In: CLUSTER 2018. pp. 302–313 (2018). https://doi.org/10.1109/CLUSTER.2018.00047
5. Giunta, G., Montella, R., et al.: A gpgpu transparent virtualization component for high performance computing clouds. In: Euro-Par 2010 - Parallel Processing. pp. 379–391 (2010), https://doi.org/10.1007/978-3-642-15277-1_37
6. Gupta, V., Gavrilovska, A., et al.: Gvim: Gpu-accelerated virtual machines. In: ACM Workshop on System-level Virtualization for High Performance Computing. pp. 17–24. ACM (2009), http://doi.acm.org/10.1145/1519138.1519141
7. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters. Journal of Physics Conference Series **46**, 494–499 (2006). https://doi.org/10.1088/1742-6596/46/1/067
8. Kannan, S., Farooqui, N., et al.: Heterocheckpoint: Efficient checkpointing for accelerator-based systems. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 738–743 (2014). https://doi.org/10.1109/DSN.2014.76
9. Karlin, I., Bhatele, A., et al.: Exploring traditional and emerging parallel programming models using a proxy application. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 919–932 (2013). https://doi.org/10.1109/IPDPS.2013.115
10. Lagar-Cavilla, H.A., et al.: Vmm-independent graphics acceleration. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. pp. 33–43. ACM (2007). https://doi.org/10.1145/1254810.1254816
11. Nukada, A., Takizawa, H., et al.: Nvcr: A transparent checkpoint-restart library for nvidia cuda. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. pp. 104–113 (2011). https://doi.org/10.1109/IPDPS.2011.131
12. NVIDIA Corporation: Nvidia cuda compute unified device architecture programming guide version 10.1.105 (2019), https://bit.ly/2EcQ4hN
13. Oikawa, M., Kawai, A., et al.: Ds-cuda: A middleware to use many gpus in the cloud environment. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. pp. 1207–1214 (2012). https://doi.org/10.1109/SC.Companion.2012.146

14. Peña, A.J., Bland, W., et al.: Vocl-ft: introducing techniques for efficient soft error coprocessor recovery. In: SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2015). https://doi.org/10.1145/2807591.2807640
15. Pourghassemi, B., Chandramowlishwaran, A.: cudacr: An in-kernel application-level checkpoint/restart scheme for cuda-enabled gpus. In: CLUSTER 2017. pp. 725–732 (2017). https://doi.org/10.1109/CLUSTER.2017.100
16. Shi, L., Chen, H., Sun, J., et al.: vcuda: Gpu-accelerated high-performance computing in virtual machines. IEEE Transactions on Computers **61**(6), 804–816 (2012). https://doi.org/10.1109/TC.2011.112
17. Taichiro Suzuki, Akira Nukada, S.M.: Transparent checkpoint and restart technology for cuda applications. `https://bit.ly/2DzHGbO` (2016), online; accessed 25-April-2019
18. Takizawa, H., Sato, K., et al.: Checuda: A checkpoint/restart tool for cuda applications. In: 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies. pp. 408–413 (2009). https://doi.org/10.1109/PDCAT.2009.78
19. Takizawa, H., et al.: CheCL: Transparent checkpointing and process migration of OpenCL applications. In: IPDPS, 2011 IEEE International. IEEE (2011). https://doi.org/10.1109/IPDPS.2011.85
20. Xu, X., et al.: Hial-ckpt: A hierarchical application-level checkpointing for cpu-gpu hybrid systems. pp. 1895–1899 (2010). https://doi.org/10.1109/ICCSE.2010.5593819