Vol.2010-HPC-126 No.18
2010/8/4

MPI-CUDA Applications Checkpointing

Nguyen Toan[†] Tatsuo Nomura[†] Hideyuki Jitsumoto[††]
Naoya Maruyama[†] Toshio Endo[†] Satoshi Matsuoka[† ††† ††††]

We describe a method to checkpoint MPI applications that use GPUs as accelerators. As current MPI checkpointing tools such as LAM/MPI and Open MPI do not support checkpointing states on GPU, it is a big hindrance for users who want to develop hybrid MPI CUDA applications running on large-scale clusters with high rate of failure. Here we propose a method to checkpoint MPI CUDA applications by integrating Open MPI, BLCR and our CUDA checkpointer. Our CUDA checkpointer hooks CUDA Runtime API calls to record data on GPU for backup during checkpoint/restart sessions and we integrate this checkpointer into the BLCR checkpoint/restart module in Open MPI. In this method, our CUDA checkpointer will monitor and record CUDA resources used on the GPU during program execution. At checkpointing, it is invoked to checkpoint states on GPU by calling our user-defined callback function in BLCR. As restarting, the CUDA checkpointer will perform restoring data and CUDA contexts on the GPU together with Open MPI's restarting service. Based on this methodology, our implementation demonstrates that MPI CUDA applications in which CUDA Runtime API codes are used can be checkpointed and restarted properly in a transparent way. Our implementation also shows a checkpoint overhead of about 38 seconds in checkpointing a 3D stencil application with size 256x256x600 running on 60 GPU-enabled nodes.

1. Introduction

In recent years, general-purpose computation on graphics processing units (GPGPU) is getting more and more popular in the field of high performance computing (HPC). The GPU has a highly parallel architecture that can help accelerating graphics rendering greatly. Nowadays, the parallel architecture in GPU has been made programmable so that parallel computing can be performed on the GPU. With the considerable speedup gained from GPU computing, recent large-scale HPC systems, especially supercomputers, are using more and more GPUs to achieve better performance. As a result, many adoptions of hybrid programming such as MPI-CUDA programming, which mixed MPI and CUDA code in an effective way to take advantage of highly parallel computations are being conducted.

Concerning GPGPU's reliability, fault tolerance, including checkpoint/restart, needs to be urgently supported. Currently there are a lot of checkpoint/restart implementations available but so far none of them supports checkpointing applications running on GPUs. This is a critical obstacle for programmers who want to develop long running GPU applications such as dynamic fluid simulation on large-scale HPC systems which are usually susceptible to failures.

We present a methodology and a prototype implementation which enables transparent checkpoint in MPI CUDA applications by integrating Open MPI, BLCR and our CUDA checkpointer. Our CUDA checkpointer monitors and records data used on the GPU during the application execution for backup or restoration at checkpoint/restart. At checkpointing, our CUDA checkpointer is invoked by the registered callback function in BLCR to checkpoint the states on the GPU. At restarting, our CUDA checkpointer restores the backup data and the CUDA context on the GPU together with Open MPI's restart functionality. Based on this methodology, our prototype implementation verifies that MPI CUDA applications such as 3D stencil can be checkpointed and restarted properly. Our implementation also shows that a 3D stencil MPI CUDA application with size 256x256x600 running on 60 nodes with GPUs can be checkpointed in about 38 seconds.

2. Background

2.1 Checkpoint/restart

Checkpoint/restart is a rollback recovery technique which is widely adopted by many systems. Conceptually, checkpointing is a technique to capture an image (or snapshot) of a running program and write it into disk. When execution of the program aborts due to a system failure, we can restart the program from the last saved state. In the cases of single non-GPGPU process, the state to be saved includes all contents of the address space of the process and CPU registers. In system level checkpointing, kernel level data, such as pending

[†] Tokyo Institute of Technology
[††] University of Tokyo
[†††] National Institute of Informatics
[††††] Japan Science and Technology Agency

signal state and file descriptors are also saved. For parallel applications that consist of several processes, things are more complicated; according to Strom et al, a generic correctness condition for rollback-recovery is that "a system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure" [1]. Therefore, in addition to saving the individual state of application processes, we have to take care of interactions among the processes such as floating MPI messages [2].

There have been many checkpoint/restart implementations such as libckpt [3], Condor checkpoint library [4] and Berkeley Lab Checkpoint Restart (BLCR) system-level checkpointer [5]. BLCR, which we use as a building block, is getting popular since it supports a wide range of applications including multi-threaded ones without modifying application codes or Linux kernel. BLCR is also supported by Open MPI [6] to enable fault tolerance of MPI parallel applications. There exists several classes of parallel checkpointing algorithms: coordinated, uncoordinated and message-driven. Open MPI adopts a coordinated algorithm in order to maintain consistency among processes, and uses BLCR in order to save states of each process. BLCR's another facility, which is relevant to this work, is supporting user-level callback; it allows users to register callback functions that are called in checkpointing and restarting, respectively. We use this facility to save states about GPGPU as described later.

2.2 Multi-GPU application

Our goal is to checkpoint and restart parallel applications that are accelerated by multiple GPUs; we assume that they are based on MPI and each participating process is connected with CUDA GPU(s) as shown in Figure 1.

CUDA (Compute Unified Device Architecture) [7] is a parallel computing environment for GPUs developed by NVIDIA, which extends C/C++ programming language. CUDA GPU mainly consists of many-core streaming multiprocessors (SMs), device memory shared by SMs on the GPU, and scratch pad memory called shared memory per SM. In our context, it should be noted that the device memory is separated from the host memory, thus it is not saved by existing checkpointers. Typical CUDA programs consist of GPU parts and CPU parts, the former is called "kernel functions". In order to write the CPU parts, CUDA provides APIs as follows:

- Allocating memory regions on device memory, e.g. cudaMalloc.
- Sending data from host memory to device memory, and receiving data from device memory to host memory, e.g. cudaMemcpy.
- Invoking kernel functions on the GPU.
- Others, such as synchronization between the CPU and the GPU.

In CUDA architecture, the lifetimes of data on registers and shared memory are a single kernel function. When a kernel function finishes, those data are discarded. On the other hand, that of data on device memory is the whole application execution, and they may survive among several kernel invocations.

Since CUDA itself does not support interaction among multiple GPUs, many HPC researchers have implemented multi-GPU applications based on MPI [8][9]. Figure 1 shows a program linked with both MPI library and CUDA library; the former is for interaction among CPU processes and the latter is for interaction between each CPU process and a GPU. When we want to move some data from the GPU on process 1 to the GPU on process 2, we have to call cudaMemcpy (device to host) and MPI_Send on process 1, MPI_Recv and cudaMemcpy (host to device) on process 2.

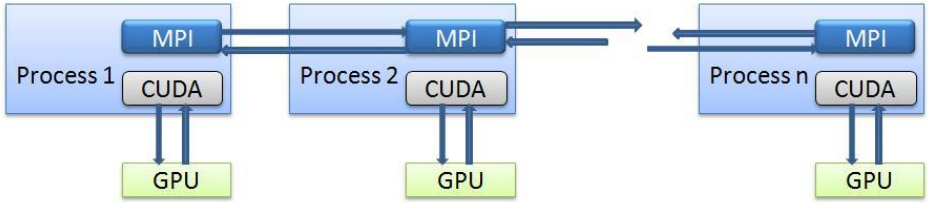

Figure 1 Communication in an MPI CUDA program

3. Checkpointing multi-GPU application

This section discusses issues that arise in checkpointing multi-GPU applications. Our start point is Open MPI with BLCR, although we consider our method is applicable to other baseline checkpointers and MPI implementations. Considering the application structure described in Section 2.2, we have to save the following information to achieve checkpoint/restart multi-GPU applications:

a) States of processes on CPUs
b) States on GPUs
c) States of interaction among processes
d) States of interaction between GPUs and corresponding CPUs

As already described, a) and c) is covered by Open MPI with BLCR, thus we focus on b) and d). Our basic policy is that we reduce kinds of GPU data to be saved in order to simplify the implementation. According to it, we decided to avoid making snapshots of GPUs when kernel functions are running. The reason is that when we try to make a snapshot during kernel functions, we have to save contents of registers and shared memory. When we make s snapshot when no kernel is running, we only need to save contents of device memory.

Another reason is that we do not have generic ways to suspend running kernel functions. We also avoid snapshots when CPU-GPU communication is running; thus we can simply ignore effects of d).

With the above discussion, we can derive the following simple strategy. When checkpointing is started, we wait until all kernel functions and CPU-GPU communication finish. Then we replicate all the contents of GPU device memory into host memory of the corresponding CPU process --- GPU states are now included in CPU states. In order to "hook" the checkpointing to evacuate GPU states, we harness the callback facility of BLCR. Then we can invoke the original checkpointer of Open MPI+BLCR to save all the states. In restarting, we require the inverse operations; the original system first reads the contents of the snapshot and recovers CPU states. Next, our checkpointer moves contents of the replica of GPU states back to device memory, and then restarts the application execution.

Although this strategy is fairly simple, we have found a number of problems that make implementing our checkpointer very challenging. Some of them come from characteristics of the current CUDA runtime, and others come from those of Open MPI and BLCR.

- We do not have standard ways to scan all used regions on device memory. We need to scan (1) all the data objects allocated by cudaMalloc, and (2) all the code objects of kernel functions.
- Also we do not have standard ways to keep addresses of data objects constant across restarting. During restarting, if we call cudaMalloc in order to recover each data object, we may obtain different device addresses than before. In C/C++ programs, change of addresses of data objects raises a lot of problems.
- We have noticed that a structure called "the CUDA context" has to be considered. This structure is implicitly made by CUDA runtime on host memory and used to keep track of states of the graphics device driver. It uses the mmap systemcall in order to touch /dev/nvidia0. However, BLCR fails to recover the mmap'ed regions.
- When Open MPI starts checkpointing, all the processes are suspended by signals, rather than user-level polling. Thus we have to postpone delivery of signals if kernel functions are running on GPUs.

Next section will describe the detailed implementation of our checkpointer and how we solve these problems.

4. Implementation

4.1 CUDA checkpointer

We observe that if a CUDA context exists at checkpointing time, BLCR fails to restart CUDA application from the checkpoint file properly. For example, BLCR fails to restart when doing mmap() of the GPU character device, e.g. /dev/nvidia0. To prevent such situation, we propose a scheme for checkpointing CUDA applications as follows:

(i) Step 1: Copy all the user data in the device memory to the host memory and destroy CUDA context.
(ii) Step 2: Have a CPU checkpointer like BLCR do checkpointing the CPU states.
(iii) Step 3: Copy copied data on CPU back to GPU and restore CUDA context.

As it is difficult to control huge amount of threads during the GPU kernel execution, at checkpointing time, cudaThreadSynchronize() is called to guarantee the GPU kernel finishes its execution before the above steps are performed. Moreover, in order to back up and restore data and CUDA contexts on the GPU properly, it is necessary to keep track of all the data being used on the GPU and relating CUDA resources. This involves managing the data objects and code objects on the GPU.

4.1.1 Managing data objects on GPU

Since the cudaMalloc API in CUDA Runtime returns a different GPU memory address each time it is called, we need to be able to track all data objects and a deterministic memory allocator to keep GPU memory addresses unchanged during a checkpoint

We build a constructor named cudartMemoryContext which is used to allocate nearly all GPU memory. Then we create an object list which each object contains the information of the base address and the size of the allocated memory region to manage the CUDA resources. Each time the function cudaMalloc, which is used to allocate a chunk of memory on GPU, is called, our tool automatically records the information relating to that memory region in an object and store it in the object list. Also instead of letting CUDA randomly allocate new memory regions, we manage to arrange these regions in a continuous order like an array, so that we can reallocate them correctly when restarting. By doing this, memory addresses of data on GPU can be kept unchanged during the checkpoint and therefore CUDA applications can be restarted properly.

4.1.2 Managing code objects on GPU

Besides restoring data objects in CUDA programs, code objects also need to be restored properly. In CUDA Runtime, this is done by implementing wrapper functions of _cudaBinRegisterFatBinary and _cudaRegisterFunction to control registered information in the GPU kernel.

```
int my_callback_handler(void *arg) {
    Back_up_data_on_GPU_and_destroy_CUDA_context();
    cr_checkpoint(0);
    Restore_backup_data_back_to_GPU_and_recreate CUDA_context();
    return 0;
}
```

Figure 2 A prototype code of our custom callback handler

4.1.3 Limitation of our CUDA checkpointer

The current implementation of our CUDA checkpointer has several limitations. First, not all CUDA objects, including texture bindings, streams, and events, are checkpointed, and thus applications using such objects would fail to restart. We see that they could also be supported in a similar way to CUDA memory objects, and we are currently extending our checkpointer for more complete support of the CUDA functionalities. Second, the CUDA API provides a custom memory allocator for host memory that allows allocated chunks to be pinned down so that DMA transfers between host and GPU memory can be run with minimal overheads. Pinned-down chunks reside in system memory, for which we use BLCR to save and restore. Our experiments with BLCR, however, fail to restart applications with pinned-down memory, and thus we are unable to use the low-overhead host-GPU transfers in CUDA applications. Extending BLCR to support checkpointing of pinned-down memory chunks could solve this problem; however, this may not be a reasonable approach, since the details of the way such memory is allocated may be hidden in the binary-only GPU driver code. Another issue is supporting the CUDA Driver API, which is a variant of the CUDA API discussed in this paper and providing more fine-grained low-level accesses to GPU devices. While few CUDA applications are written in the Driver API, some highly performance-conscious ones exploit features that are only available in the low-level API. We believe that the framework described in this paper could be similarly applied to the low-level API.

4.2 MPI, BLCR and CUDA checkpointer integration

In order to attach our CUDA checkpointer to BLCR, we make use of the user callback function in BLCR. For users want to interact with checkpoint/restart, BLCR provides a way to register user-level callback functions, which are triggered at checkpointing time and continue when a restart is initiated. Moreover, since Open MPI uses signal to invoke checkpoint, we attach our CUDA checkpointer by register a callback handler which will be called at Open MPI's checkpointing time to checkpoint the states on GPU. Since BLCR executes all user callback functions during the checkpoint, the states on GPU are guaranteed to be saved in the checkpoint and hence restored on the GPU at application restarting or continuing time. Figure 2 shows a prototype implementation of our callback handler which performs three steps in Section 4.1.

However, integrating the CUDA checkpointer into Open MPI+BLCR is not simple. Below are some problems arising from this integration and our solution to these problems.

a) Signal interruption during CUDA API execution

Since CUDA APIs are not guaranteed to be Async-Signal-Safe, the CUDA checkpointer which is invoked inside the signal handler may influence the CUDA APIs' proper behaviors when they are interrupted by the Open MPI's checkpoint signal. In order to prevent such unwanted interruption, we propose a signal masking scheme to all CUDA APIs as follows:

i) Before each CUDA API execution, the signal handler is modified to perceive the arrival of the checkpoint signal.
ii) After each CUDA API execution, the signal handler is returned back to the original one which handles checkpointing.
iii) If the checkpoint signal arrives in the middle of signal masking, it will be sent to the current thread to perform checkpointing.

b) CUDA API execution in signal handler

Since some CUDA APIs used in our CUDA checkpointer such as cudaMemcpy (host to device and device to host), cudaThreadSynchronize() are not likely to perform properly in the signal context, we conducted tests to verify their operations. We observe that these APIs operate properly under signal contexts like Sync-Signal-Safe functions. This means our CUDA checkpointer integrates well with Open MPI+BLCR and thus guarantees proper checkpoint operation.

5. Evaluation

To verify our implementation's proper behaviors and evaluate the checkpoint overheads, we conducted some experiments on our GPU clusters, named Raccoon, and on the supercomputer TSUBAME at Tokyo Tech[11] with the hardware and software specifications shown in Table 1. We verified that MPI CUDA applications can be checkpointed and restarted properly through checkpoint/restart commands in Open MPI. Also, we measured the runtime of the application together with the overall overhead incurred during a checkpoint. This overall checkpoint overhead includes the following runtime of these procedures:

(a) Pre-processing: includes 1) the waiting time for all kernel functions and CPU-GPU communication finish and 2) the cost for transferring contents of GPU device memory into host memory of the corresponding CPU process.

(b) BLCR overhead: is the runtime of BLCR to checkpoint states on the CPU.
(c) Post-processing: includes 1) the overhead of the original system reading the contents of the snapshot and recovering CPU states and 2) the overhead of our checkpointer moving contents of the replica of GPU states back to device memory and restarting the application execution.
(d) Others: are the overheads incurred in Open MPI's checkpointing and resuming operations.

For the overheads in (a),(b),(c) we take the highest sum calculated among the local nodes and the overall checkpoint runtime is taken from the application runtimes in normal execution and execution with checkpoint.

5.1 CUDA checkpointer microbenchmark

Firstly we evaluate the checkpoint overheads in our CUDA checkpointer. The checkpoint target is a simple CUDA program which allocates raw data with size varying from 100 MB to 1000 MB. The checkpoint is performed on a single machine on Raccoon. As shown in Figure 2, the overhead pre-processing and post-processing is very small, while the runtime of BLCR increases linearly with the data size.

Table 1 Experiment environment

	Raccoon	TSUBAME
CPU/Memory	Intel i7 920 2.67GHz (4core/8thread) 12GB Memory	AMD Opteron 880 2.4GHz x 8(16core) 32GB Memory
GPU/Memory	Tesla C2050, 2.6GB Memory	Tesla S1070, 4GB Memory
BLCR	0.8.2	0.8.1
CUDA	3.0	2.3
Open MPI	1.4.2	1.4.2
Network	Infiniband 4x DDR	Infiniband 4x SDR
Local Disk	SSD	HDD

Table 2 The overhead in "Others" with varying problem size and # of processes

Problem size	# of process	BLCR (sec.)	Others (sec.)
500x500x500	50	25.1	307
500x500x500	40	25.7	273
400x400x400	40	14.5	87.9

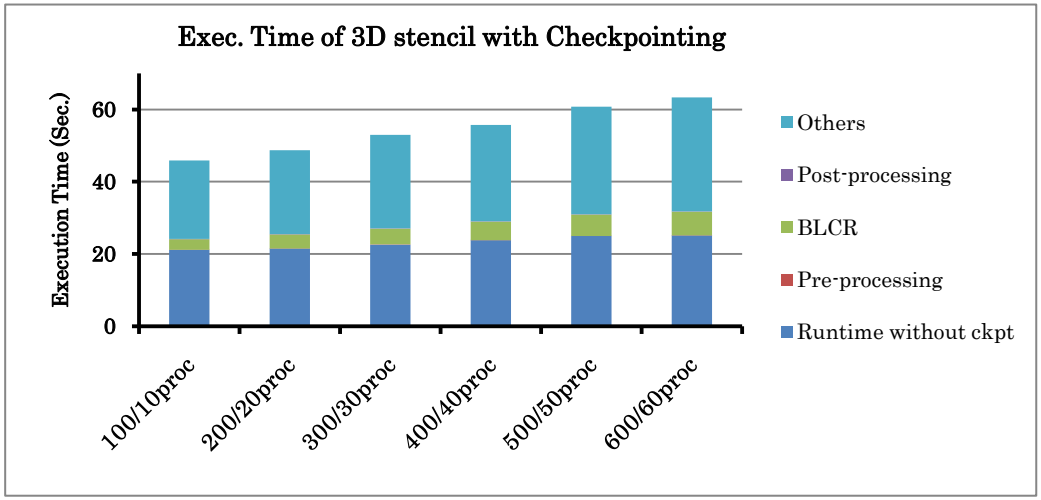

Figure 3 Execution time of 3D stencil with checkpointing. 100/10proc means experiment running on 10 processes with data size on Z-axis equal to 10

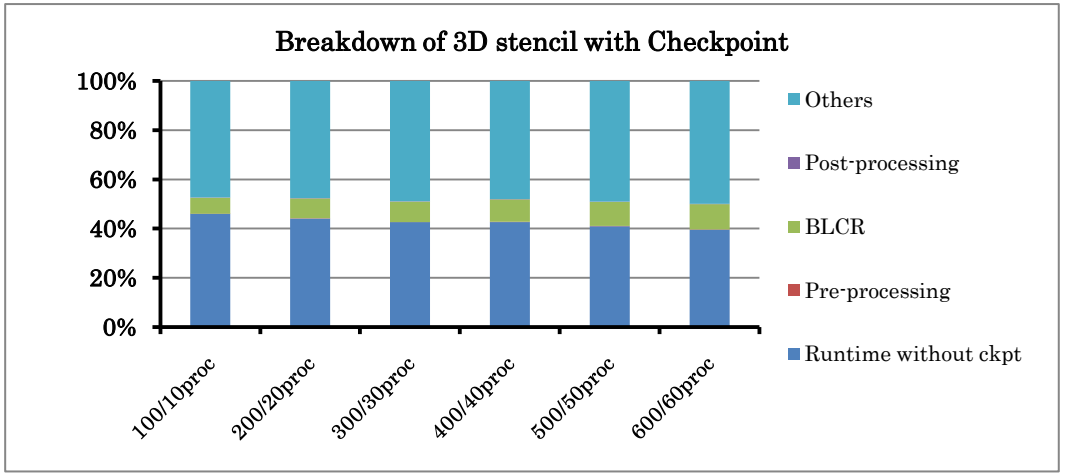

Figure 4 Breakdown of 3D stencil with checkpoint

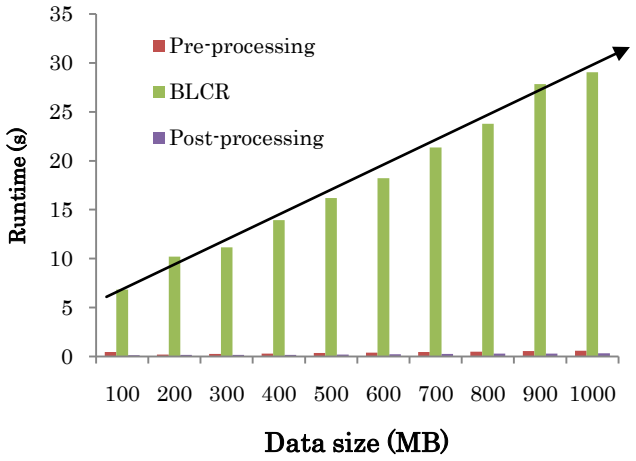

Figure 2 Relation between data size and checkpoint overhead in our CUDA checkpointer

5.2 MPI CUDA checkpoint overhead

Next we conducted weak scaling experiments on TSUBAME to analyze the scalability and overheads of checkpointing a simple 3D stencil MPI CUDA application. Basically, a stencil is a specified set of nearest neighbors for a given point and a stencil code updates every point in a regular grid based on weighted subset of its neighbors. Stencil codes are used in many scientific applications such as diffusion and computational fluid dynamics. For the experiment, we observe how the application runtime changes when the number of processes is varied from 10 to 60 together with the size of the Z-axis in 3D stencil computation varied from 100 to 600. Sizes on X and Y axes are fixed as 256 and one checkpoint is taken per application execution. The experimental result in Figure 3 which illustrates the execution time of 3D stencil program with checkpointing shows that all execution time increase together with the number of processes and the size of Z-axis and hence we cannot get weak scaling property. This is understandable since each process maintains the data of the whole problem in the 3D stencil application. This obviously results in the increase of memory size and file writing time in each process as we have observed in Figure 3.

Besides, the overhead in "Others" which is supposed to be the coordination time occurred in Open MPI's checkpointing occupies about 50% of the overall overhead as shown in Figure 4. Table 2 shows the overhead of "Others" when we change the problem size together with the number of processes. When the problem size is fixed at 500^3, the overhead in "Others" is reduced when the number of processes is decreased; this means that the coordination time in Open MPI probably depends on the number of processes. However, the overhead in "Others" also changes greatly when the problem size changes with the fixed number of processes; this can be inferred that some cost dependent on the problem size may also be included in "Others".

6. Related Work

Regarding checkpoint/restart on GPU, H.Takizawa et al. provides a tool named CheCUDA [10] which allows checkpointing CUDA applications written in CUDA Driver API. CheCUDA manages CUDA resources by indirect handlers while our CUDA checkpointer uses a custom memory allocator to control GPU memory allocation. However, in order to enable checkpointing, users are required to insert CheCUDA's checkpoint function in the program code and have to recompile their programs. Also, CheCUDA does not support CUDA programs which have data with pointer references within the GPU and multi-GPU applications checkpointing. Our tool provides better checkpointability, which supports checkpointing CUDA programs using data with pointer references on GPU, as well as supports checkpointing multi-GPU applications. Also we design our implementation as a shared library which allows users to checkpoint their programs by only setting LD_PRELOAD environment variable to our library. No code modification or recompilation is required. Also the users can checkpoint and restart the application by commands provided in Open MPI such as ompi-checkpoint and ompi-restart.

7. Conclusion and Future Work

We have presented a methodology to checkpoint multi-GPU applications by proposing a framework combining Open MPI, BLCR and our CUDA checkpointer. Based on our prototype implementation, we verified that MPI CUDA applications such as 3D stencil can be checkpointed and restarted properly. We also conducted checkpointing experiments to gain microbenchmark evaluation and breakdown of checkpoint overheads in parallel checkpointing experiments. In this breakdown, we verified that there is an unexpected overhead incurred during the checkpoint which is not included in the writing time of BLCR and depends on the number of processes and memory size.

In our future work, we will contribute a more detailed analysis of this breakdown and optimize our implementation based on this result. More complete support for CUDA functionalities in our CUDA checkpointer will be included.

Vol.2010-HPC-126 No.18
2010/8/4

Acknowledgements This work is partially supported by the JST-CREST Ultra-Low-Power HPC Project, MEXT Grant-in-Aid for Young Scientists (22700047), and NVIDIA CUDA Center of Excellence.

References

1) Robert E. Strom, Shaula Yemini: Optimistic Recovery in Distributed Systems, ACM Trans. Comput. Syst. 3(3), pp.204-226 (1985).
2) E.N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson: A survey of rollback-recovery protocols in message-passing systems, ACM Computing Surveys (CSUR), Volume 34, Issue 3, pp.375-408 (2002).
3) James Plank, Micah Beck, Gerry Kingsley, Kai Li, Libckpt: Transparent Checkpointing under Unix, Usenix Winter Technical Conference, pp.213-223 (1985).
4) Douglas Thain, Todd Tannenbaum, and Miron Livny: Distributed Computing in Practice: The Condor Experience, Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pp.323-356 (2005).
5) Paul H Hargrove and Jason C Duell: Berkeley lab checkpoint/restart (BLCR) for Linux clusters, Proceedings of SciDAC (2006).
6) Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine: The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI, DPDNS 07, 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (2007).
7) NVIDIA Corporation: CUDA Zone – The resource for CUDA developers, http://www.nvidia.com/object/cuda_home_new.html.
8) Phillips E.H., Fatica M.: Implementing the Himeno benchmark with CUDA on GPU clusters, IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp.1-10 (2010).
9) Dana A. Jacobsen, Julien C. Thibault, and Inanc Senocak: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters, 48th AIAA Aerospace Sciences Meeting and Exhibit. Orlando, FL. (2010).
10) H.Takizawa, K.Sato, K.Komatsu, H.Kobayashi: CheCUDA: A Checkpoint/Restart Tool for CUDA Applications, Proceedings of the Tenth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp.408-413 (2009).
11) Tokyo Tech's TSUBAME: http://www.gsic.titech.ac.jp/en/tsubame