CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications

Hiroyuki Takizawa*,†, Kentaro Koyama*, Katsuto Sato*, Kazuhiro Komatsu*, and Hiroaki Kobayashi*‡

* Tohoku University
Sendai, Miyagi 980-8578, Japan
E-mail: {tacky@, kentaro@sc., katuto@sc., komatsu@sc., koba@isc.tohoku.ac.jp
† Japan Science and Technology Agency, CREST

Abstract—In this paper, we propose a new transparent checkpoint/restart (CPR) tool, named CheCL, for high-performance and dependable GPU computing. CheCL can perform CPR on an OpenCL application program without any modification and recompilation of its code. A conventional checkpointing system fails to checkpoint a process if the process uses OpenCL. Therefore, in CheCL, every API call is forwarded to another process called an API proxy, and the API proxy invokes the API function; two processes, an application process and an API proxy, are launched for an OpenCL application. In this case, as the application process is not an OpenCL process but a standard process, it can be safely checkpointed. While CheCL intercepts all API calls, it records the information necessary for restoring OpenCL objects. The application process does not hold any OpenCL handles, but CheCL handles to keep such information. Those handles are automatically converted to OpenCL handles and then passed to API functions. Upon restart, OpenCL objects are automatically restored based on the recorded information. This paper demonstrates the feasibility of transparent checkpointing of OpenCL programs including MPI applications, and quantitatively evaluates the runtime overheads. It is also discussed that CheCL can enable process migration of OpenCL applications among distinct nodes, and among different kinds of compute devices such as a CPU and a GPU.

I. INTRODUCTION

Recently, numerous researchers have demonstrated that graphics processing units (GPUs) can accelerate various kinds of applications [1]. GPUs have high floating-point operation rates and memory bandwidths. Therefore, use of GPUs as accelerators, so-called GPU computing or GPGPU, has become very popular in the field of high-performance computing (HPC). However, as GPU computing is still emerging, several important tools commonly used in HPC have not been available for GPU computing yet. One of such tools is checkpoint/restart (CPR).

CPR writes the state of a running process to a checkpoint file, from which the process state can be restored. Lots of CPR implementations [2][3][4] have been developed for CPUs but not for GPUs, even though CPR is important to enhance the dependability of GPU computing systems. Dependability will be a major concern of future GPU computing from several reasons. One is that GPUs are increasingly used in large-scale systems of many computing nodes, such as TSUBAME [5], to execute long-running simulations, resulting in a high probability of facing hardware/software failures during the execution. Another reason is that a GPU computing application is also executed on a commodity PC, which may not be designed for GPU computing and sometimes becomes unstable due to insufficient cooling capability.

Furthermore, CPR plays an important role in process migration. For process migration, a snapshot of a running process is taken on a node, and then resumed on a different node. CPR for GPU computing applications is useful for dynamic job scheduling in a GPU cluster system. Using CPR, a job scheduler may migrate a running process from one computing node to another so as to improve the performance in terms of the total system throughput, the turnaround time, and/or the energy efficiency. Such a dynamic job scheduling mechanism will be important especially for a large-scale heterogeneous computing system with various GPUs and other accelerators.

In our previous work [6], CheCUDA has been developed as a CPR tool for CUDA, which is the de facto standard programming framework for current GPU computing [7]. In CheCUDA, all CUDA resources are once deleted before checkpointing, and then restored after checkpointing. This approach assumes that a CUDA process becomes "checkpointable" when all the CUDA resources are deleted. Although this works fine for the systems used in [6], it strongly relies on the underlying checkpointing system and the implementation of the CUDA library and runtime system. Therefore, a more stable approach is necessary to safely take a snapshot of a GPU computing process.

A subtle but significant issue with CheCUDA is that it is not transparent to application codes, i.e., it needs to recompile the codes, and supports only CUDA Driver API, even though most CUDA applications are developed with CUDA Runtime API. As a result, its applicable areas are severely limited. Transparent checkpointing is highly preferable for supporting a wide range of GPU computing applications.

In this paper, we develop a transparent CPR tool for high-performance and dependable GPU computing, named CheCL, which can perform CPR on a GPU computing...
application without any modification and recompilation of its code. As the name implies, CheCL is designed for OpenCL, which is a new programming framework for GPU computing [8]. CheCL adopts an API proxy technique for completely decoupling an application process from the OpenCL implementation. Moreover, CheCL is designed to be transparent to the application process, by implicitly and appropriately exchanging data between an application process and the OpenCL implementation while recording necessary information for restoring the GPU state. This paper demonstrates that CheCL can safely checkpoint OpenCL processes by using the API proxy, and support various OpenCL applications due to its transparent checkpointing capability. Then, the timing overheads required for the capability are evaluated. After that, we discuss process migration using CheCL.

The rest of this paper is organized as follows. Section II briefly reviews GPU computing with OpenCL. Then, Section III presents CheCL, which is a CPR tool for OpenCL applications. Section IV discusses the timing overhead of CheCL through some evaluation results. Section V describes related work. Finally, Section VI gives concluding remarks and our future work.

II. GPU Computing with OpenCL

NVIDIA’s CUDA [7] is currently the de facto standard programming framework for GPU computing that incorporates some additional keywords into the standard C/C++ programming language. CUDA allows a programmer to access GPUs without tricky graphics programming techniques used in GPU computing with graphics APIs such as OpenGL and DirectX. Therefore, CUDA has played a very important role for popularization of GPU computing. However, CUDA is available only for NVIDIA’s GPUs, not for other GPUs and accelerators.

OpenCL is a new programming standard open to other vendors. OpenCL enables a programmer to access various GPUs and accelerators in a unified way. In OpenCL, a CPU usually works as a host that controls a compute device such as a GPU. A host and a compute device have their own memory spaces, host memory and device memory, respectively. A typical OpenCL application running on a host first initializes a compute device, allocates a device memory chunk, copies host memory data to the allocated chunk, invokes a special function processing the device memory data with the compute device, and retrieves the computation results from the device memory.

In OpenCL, there are several OpenCL objects such as contexts, command queues, programs, kernels, and memory objects [8]. A context is analogous to a CPU process, and a command queue is used to schedule execution of kernels and memory operations. A program can include several kernels that are special functions running on a compute device. A memory object is bound to a memory region on the device memory. Every object is referenced with a unique handle, and any operation on an object must go through the object’s handle.

There are two reasons for the existing CPR systems to fail on OpenCL processes. One is that existing CPR systems such as [2][3] are designed to restore only the CPU states, but not the GPU states, i.e. OpenCL objects. Although the existing CPR systems can restore the value of a handle, the restored handle is not actually bound to any OpenCL object. As a result, an OpenCL object that existed before checkpointing no longer exists after restarting.

The other reason is that several special devices are mapped to the memory space of an OpenCL process by the GPU device driver. Since the existing CPR system does not know how to handle those memory-mapped devices, it fails to checkpoint the process when accessing the devices. Generally, accessing unknown devices results in implementation-dependent behavior. In CheCUDA [6], a CUDA process is assumed to become checkpointable only by deleting all the CUDA resources. However, there are Linux kernel versions, for which CheCUDA does not work or causes kernel crashes.

III. Checkpoint/Restart of OpenCL Applications

This paper proposes a CPR tool, named CheCL, for CPR of OpenCL applications. CheCL works with a conventional CPR system that is developed for writing the host memory image of a process into a checkpoint file. CheCL is responsible for saving and restoring OpenCL objects.

CheCL intercepts API calls in any OpenCL applications on the system by replacing the OpenCL shared object, i.e., libOpenCL.so, with the CheCL version shared object. The installed CheCL is then configured to use a renamed version of the original libOpenCL.so for actual API calls. It is also possible to make CheCL optional to users, by switching those two shared objects. Anyway, CheCL does not need any modification and recompilation of application codes, and can achieve completely transparent checkpointing with some runtime overheads to be discussed later.

In CheCL, the following two techniques are used to achieve CPR of OpenCL applications.

- **API proxy** that is a proxy mechanism to make an application process checkpointable by completely decoupling the process from OpenCL implementation. In addition, by providing an API proxy for each OpenCL implementation, an application process can transparently use multiple OpenCL implementations; an application process running with one OpenCL implementation can restart with another OpenCL implementation.
- **CheCL objects** that are wrapper classes of OpenCL objects to implicitly record the necessary information for restoring the GPU state. In CheCL, an application process does not know OpenCL handles, but the pointers to CheCL objects called CheCL handles. CheCL
handles are converted to OpenCL handles and then passed to the API proxy.

A. A Proxy Mechanism

CheCL employs an API proxy technique shown in Figure 1 to decouple an application program from OpenCL implementations such as OpenCL libraries and runtime systems. When the CheCL version library is dynamically loaded by an application program, the OpenCL application is executed by at least two processes, an application process and an API proxy.

Every API call of the application process is forwarded to the API proxy, which is another process automatically forked by the application process. Then, the API proxy invokes the actual API function. In this case, the API proxy is an OpenCL process, and some special devices are mapped to its memory space. On the other hand, the application process is itself a standard process just communicating with the API proxy. This means that its memory space does not have any devices unknown to a checkpointing system, and thus it is safely checkpointable.

B. Data Management for OpenCL Objects

Every handle of OpenCL is an opaque pointer, which is a synonym of a structure pointer. For example, the variable type of a context handle, cl_context, is declared in the header file CL/cl.h as follows:

typedef struct _cl_context* cl_context;

Here, struct _cl_context is defined by OpenCL implementations and not disclosed to application codes. If an OpenCL API function clCreateContext is invoked, the function creates a context and returns its handle. For example,

cl_context ctx = clCreateContext(0, ndev, dev, 0, 0, 0);

Thus, to recreate this context after restarting, CheCL has to know the information how the context was created, such as the arguments given at the object creation API call. In addition, there are several API functions that change the state of OpenCL objects. Therefore, CheCL has to again apply the changes to the restored objects.

To keep such information, CheCL uses a wrapper class instead of an OpenCL object, called a CheCL object. In CheCL, every API function is a wrapper function, which internally communicates with the API proxy to invoke the actual API function, records the actual OpenCL handle and arguments in a CheCL object, and then returns its pointer called a CheCL handle. By returning the CheCL handle, the application code knows only the CheCL handle but not an actual handle value. As a result, CheCL can implicitly change the value of an actual OpenCL handle kept in the CheCL object. This is important because the value of an OpenCL handle might change when the OpenCL object is recreated upon restart.

The handle of an OpenCL object must be retrieved from the CheCL object before being passed to the API proxy as shown in Figure 1. The API proxy assumes that an OpenCL handle, not a CheCL handle, is passed from the application process, and hence just gives it to OpenCL API functions of the underlying OpenCL implementation. In most cases, as the variable type of each argument is already determined by the function signature, it is easy to appropriately convert a CheCL handle to an actual OpenCL handle. However, clSetKernelArg that sets each argument of a kernel function takes only a const void pointer and a size_t value, which are the initial address and size of an argument, respectively. The actual variable type of each argument is unknown. Thus, the wrapper function of clSetKernelArg needs additional information to decide whether a given argument is a CheCL handle.

Although there are several kinds of OpenCL resources as shown in Figure 2, only cl_mem and cl_sampler shaded in the figure can be passed to kernel functions via clSetKernelArg. To surely retrieve handles from those CheCL objects when calling clSetKernelArg, CheCL first parses the source code of each kernel func-
tion declaration when creating a `cl_program` object with `clCreateProgramWithSource`. Then, it records which arguments are supposed to be OpenCL handles. If address space qualifiers such as `__global`, `__local`, and `__constant` are found in the parameter list, the qualified arguments must be OpenCL handles. Similarly, an argument of a special variable type such as `image2d_t`, `image3d_t`, and `sampler_t` is also an OpenCL handle. By recording such arguments in the parameter list of every kernel function, CheCL can appropriately convert CheCL handles to OpenCL handles.

C. Checkpoint and Restart

In OpenCL, command queues work asynchronously, and the relaxed consistency model ensures that the contents of memory visible from different command queues are the same only at explicit synchronization points. Thus, a host and all command queues have to be synchronized before checkpointing.

The synchronization might cause performance degradation if a host is supposed to work concurrently with compute devices. Hence, CheCL provides two modes to decide when to checkpoint a process. One is the *delayed checkpointing mode*. In the mode, once an OpenCL process has received a signal of the Linux system such as SIGUSR1, checkpointing is performed at the next synchronization point. That is, for preventing the overhead of unnecessary synchronization, the checkpointing is postponed until the command queue is synchronized with the host. This mode assumes that one of several API functions such as `clFinish()`, which enforces the synchronization, is located on a potential checkpoint in the source code. The other mode is the *immediately checkpointing mode*. Upon arrival of a signal, CheCL immediately performs synchronization and checkpointing despite the additional synchronization overhead.

CheCL performs the following steps when checkpointing an OpenCL process.

1) Synchronize the host and all command queues to complete all enqueued commands.
2) Copy all the user data in the device memory to the host memory.
3) Checkpoint the process, i.e. writing the current CPU state of the process to a checkpoint file.
4) Delete the copied user data in the host memory to save the memory usage.

On the other hand, the restart procedure is as follows.

1) Restart the process, i.e. restoring the CPU state of the process from a checkpoint file.
2) Fork a new API proxy and recreating OpenCL objects via the new proxy.
3) Send the user data back to the device memory.

Thus, CheCL remembers all the OpenCL objects that existed before checkpointing, and restores them after restarting. To do this, a database is managed to hold the pointers to all CheCL objects. Whenever a CheCL object is allocated, its handle is registered in the database. Using the database, CheCL copies the data of every OpenCL object to the host memory before checkpointing, so that the data are stored in the checkpoint file and thereby CheCL can recreate all the OpenCL objects right after restarting.

To recreate an OpenCL object, some other kinds of OpenCL objects are generally required. Figure 2 also shows the dependency among OpenCL resources. For example, creation of a `cl_mem` object requires a `cl_context` object, and the `cl_context` object itself also depends on `cl_platform` and `cl_device` objects. Therefore, CheCL restores OpenCL objects in the following order, and deletes them in the reverse order.

1) `cl_platform_id`
2) `cl_device_id`
3) `cl_context`
4) `cl_command_queue`
5) `cl_mem`
6) `cl_sampler`
7) `cl_program`
8) `cl_kernel`
9) `cl_event`

Most OpenCL objects can be restored by calling the object creation API functions with the information kept in CheCL objects. However, an event object associated with a particular command cannot be recreated; there is no API function to create an arbitrary event object. An event object of an enqueued command is used to block other commands until the command is completed. As shown in Figure 3, hence, the event objects that existed before checkpointing might be used to block the commands to be executed after restarting. It is too costly to re-execute all the commands associated with event objects. Therefore, CheCL needs to trick the underlying OpenCL implementation as all the event objects look restored.

In CheCL, all the command queues are empty when a checkpoint is taken. This is preferable also for easily handling event objects. As all command queues are empty, all events associated with event objects are completed.
In this case, those event objects never block the others. Therefore, although some event objects of the commands executed before checkpointing may be used after restarting, they do not need to be actually associated with those commands. CheCL gets a dummy event object by calling \texttt{clEnqueueMarker}, which immediately returns with an event object created by the underlying OpenCL implementation. Although the dummy event object is associated with the \texttt{clEnqueueMarker} call and thus never blocks the others, it can be used instead of the event object that existed before checkpointing.

IV. EVALUATION AND DISCUSSIONS

This section evaluates the timing overheads of CheCL presented in Section III. In the evaluation, 19 benchmark programs are selected from sample OpenCL codes in NVIDIA GPU Computing SDK 3.0. Each benchmark code consists of GPU computation, CPU computation, and their result comparison. In our evaluation, the CPU computation and result comparison parts are removed from each code in order to avoid underestimating the timing overhead in the GPU computation part. 12 benchmark programs in the SHOC benchmark suite[9] version 0.9.1 are also used. In addition, we have ported three CUDA programs, \texttt{cp}, \texttt{mri-fhd}, and \texttt{mri-q}, in the Parboil benchmark suite[10] to OpenCL for the evaluation.

In our current implementation of CheCL, Berkeley Lab Checkpoint/Restart (BLCR) version 0.8.2 [2] is used to dump the host memory to a checkpoint file, Clang/LLVM version 2.7 [11] is used to parse device codes, and Open MPI version 1.4.2 [12] is used for MPI applications. The evaluation results are obtained using four PCs whose specifications are summarized in Table I. In the table, PCIe Perf.(HtoD) and PCIe Perf.(DtoH) are the average bandwidths for sending 32MB data from the host memory to the device memory, and from the device memory to the host memory, respectively. File Write/Read Perf. indicates the average bandwidth of writing/reading a file to/from the file system in the sequential block I/O mode that is measured with Bonnie++ [13].

A. Runtime Overhead Evaluation

We first evaluate the runtime overheads of CheCL itself that are required for forwarding every API call to an API proxy and implicitly recording necessary information for CPR. Those overheads degrade the performance of a process even if a snapshot of the process is not taken at all.

In the following evaluation, the average execution time of every benchmark program linked with CheCL is calculated based on three simulation runs, and is compared to that with the original OpenCL. During the execution, checkpointing is not performed and hence the difference in execution time between the two versions indicates the timing overheads for the API proxy and internal data management of CheCL.

Figure 4 shows the evaluation results on NVIDIA OpenCL and AMD OpenCL. Here, the average execution time of each of the programs linked with CheCL is normalized by that with the native OpenCL library. In the case of AMD OpenCL, each program is executed on the CPU and the AMD GPU. Sample programs in NVIDIA SDK are not portable and thus cannot run with AMD OpenCL even if they are compiled without CheCL. For example, \texttt{oclSortingNetwork} can run on the CPU but not on the AMD GPU, because the number of work items (threads) in the x-dimension of a work group is limited to 256 in the AMD GPU and to 1024 in the CPU.

As shown in the results, CheCL can properly execute all the benchmark programs except those non-portable ones. Although the results of serial-version SHOC benchmark programs are shown in this figure, CheCL can work also for MPI-version programs. Accordingly, these results clearly indicate that CheCL can support various OpenCL features without any modification and recompilation.

CheCL needs the additional execution time for initialization, which automatically forks an API proxy when the CheCL-version OpenCL shared object is dynamically linked. Since the total execution times of most benchmark programs are short, the initialization overhead of approximately 0.08 seconds can be seen and thus the ratio of runtime overhead to total execution time becomes large in some programs. As the initialization is performed only once during the execution, the initialization overhead is usually negligible in a practical long-running application.

In \texttt{oclBandwidthTest}, \texttt{oclSimpleMultiGPU}, \texttt{BusSpeedDownload}, \texttt{BusSpeedReadback}, and \texttt{Triad}, the time for data transfers between the host and the device is dominant in the total execution time. The results of those benchmarks suggest that the API proxy

\begin{table}[h]
\centering
\caption{System Specifications}
\begin{tabular}{|l|l|}
\hline
CPU & Intel Core i7 920 (DDR3 12GB) \tabularnewline
NVIDIA GPU & NVIDIA Tesla C1060 (GDDR5 4GB) \tabularnewline
AMD GPU & AMD Radeon HD5870 (GDDR5 1GB) \tabularnewline
Chipset & Intel X58/ICH10R \tabularnewline
NIC & Gigabit Ethernet 1000BASE-T \tabularnewline
File Write Perf. & RAM disk: 4840 MB/sec \tabularnewline & Local: 106 MB/sec \tabularnewline & NFS: 21.2 MB/sec \tabularnewline File Read Perf. & RAM disk: 4840 MB/sec \tabularnewline & Local: 106 MB/sec \tabularnewline & NFS: 21.2 MB/sec \tabularnewline PCIe Perf. & HtoD: 5.35 GB/sec \tabularnewline & DtoH: 4.87 GB/sec \tabularnewline OS & CentOS 5.5 Linux \tabularnewline & Kernel Version: 2.6.18-194.11.3.el5 \tabularnewline Driver Ver. & NVIDIA: Forceware 256.40 \tabularnewline & AMD: fglrx 10.7 \tabularnewline\hline
\end{tabular}
\end{table}
approach makes the data transfer slower. This is due to the inter-process communication between an application process and its API proxy. For example, to send some data in the memory space of an application process to the device memory, the data must be first copied to the memory space of the API proxy and then sent to the device memory. Because of the extra memory copy between two processes, the sustained data transfer bandwidth is obviously reduced in some cases.

Furthermore, in a short period, some programs such as Scan and Stencil2D invoke API functions many times without any time-consuming computation. As a result, the overheads of the API proxy approach are exposed in the total execution time.

Although the above extreme programs are included, the average runtime overhead is only 10.1% of the total execution time when using NVIDIA OpenCL, 19.0% when using AMD OpenCL for the GPU, and 12.2% when using AMD OpenCL for the CPU.

In general, an OpenCL application should be designed so that the kernel execution time consumes a large part of the total execution time, minimizing the data transfers between the host and the device. For such a well-designed application, the performance degradation induced by CheCL will be acceptable. In practical uses, CheCL can be optional to users; considering the mean-time-to-failure and the runtime overheads, a user may decide whether CheCL is enabled or not.

B. Timing Overheads for Checkpointing

In the following, the timing overheads additionally induced by CheCL for checkpointing a running process are evaluated. For the evaluation, the procedure of checkpointing by CheCL is decomposed into four phases: synchronization, preprocessing, writing, and postprocessing. These phases correspond to four steps of the checkpointing procedure described in Section III-C. In the synchronization phase, a host and all compute devices wait until all enqueued commands are completed. In the preprocessing phase, all user data in the device memory are copied to the host memory. In the writing phase, a snapshot of the running process is written to a file by BLCR. In the postprocessing, the copies of the user data in the host memory are deleted to save the host memory usage. Accordingly, the writing time is required by BLCR to dump the host memory data, and the others are additional overheads induced by CheCL.

In each benchmark program, checkpointing is performed once after every kernel execution and the timing overheads for the four phases are obtained. We do not use the benchmark programs that do not execute any kernel, such as oclBandwidthTest, BusSpeedDownload, BusSpeedReadback, and KernelCompile.

Figure 5 shows the results to evaluate the average timing overheads, and also the average checkpoint file size of each benchmark program. From the figure, it is obvious that there is a strong correlation between the total checkpoint time and the checkpoint file size, and the correlation coefficient is 0.99. This is because writing a checkpoint file to the hard disk is much more time-consuming than the others such as the data transfers between the host and the device in preprocessing. Note that, as shown in Table I, the bandwidth of writing data into a hard disk is much lower than the bandwidth of sending data via the PCI-Express bus.

Unlike CheCUDA [6], CheCL does not destroy the existing OpenCL objects in the preprocessing phase. Since CheCL does not need to recreate OpenCL objects in the postprocessing phase, the postprocessing time is negligible in all the benchmark programs, resulting in a lower checkpointing overhead. This is one important advantage of the API proxy approach over CheCUDA because recovery of OpenCL objects may take a long time. This will be further discussed in Section IV-C.

In some benchmark programs such as MaxFlops, the synchronization phase also consumes a large part of the execution time of the whole checkpointing procedure. This is because CheCL must wait until all the enqueued commands are completed. In the evaluation, to show the synchronization overhead, at least one uncompleted kernel execution...
command always exists in the queue when the process is checkpointed. The synchronization time would grow with the number of uncompleted time-consuming commands in the queue. In the case where the CPU is idle while the GPU is executing those commands, the checkpointing is just postponed and thus the synchronization time does not increase the total execution time of the OpenCL application. However, in the case of an OpenCL application simultaneously using both the CPU and the GPU, the execution times of those processors are serialized, resulting in performance degradation. Therefore, in such a case, the delayed checkpointing mode described in Section III-C should be used to reduce the synchronization overhead by delaying the checkpointing until the CPU and the GPU need to be synchronized.

The problem sizes of oclFDTD3D and oclMatVecMul are determined at runtime based on the memory size available on the device. As shown in Table I, the memory size of AMD Radeon HD5870 used in this evaluation is smaller than those of the others. On the AMD GPU, therefore, those programs are executed with smaller problem sizes, and the checkpoint files become smaller.

Figure 6 shows the checkpoint times of the MPI-version MD program that were measured changing the problem size.
and the number of computing nodes. In this figure, the checkpoint time increases with the problem size, because the memory usage and thus the checkpoint file size increase proportionally with the problem size. The checkpoint files of individual computing nodes, called local snapshots, are aggregated into a global snapshot [14], and stored in an NFS file. Therefore, the checkpoint time also increases with the number of nodes.

All the above results suggest that the additional timing overheads for checkpointing OpenCL objects by CheCL are trivial especially if a process uses a large memory space. It should be noted that the writing time depends mainly on the memory usage but not on the total execution time. This means that its impact on the total execution time becomes relatively smaller for a longer-running program. For a practical application program requiring a long execution time and large memory capacity, thus, the additional overheads for checkpointing OpenCL objects would have very little effect on the execution time for the whole checkpointing procedure.

C. Process Migration

We also migrate a running process of an OpenCL application from one PC to another, in order to check if a checkpoint file generated by CheCL contains host-dependent information. In our evaluation, CheCL can enable process migration among distinct PCs if all the libraries and files accessed by the process exist on the destination PC. Even if those PCs are equipped with different GPUs such as NVIDIA GPUs and AMD GPUs, a checkpointed process on one PC can restart on another PC. Therefore, CheCL can enable process migration even in a heterogeneous GPU cluster system, in which each computing node has a different GPU.

In a heterogeneous GPU cluster system, a more powerful GPU should be assigned to a job expected to be more effectively accelerated by the GPU. However, it is not easy to statically determine such an appropriate job schedule, because a new job that is much more effectively accelerated by GPUs might be submitted after the scheduling. As CheCL can enable process migration of OpenCL processes in such a system, it is also useful to realize dynamic scheduling of running jobs to improve the total throughput and/or energy efficiency. Although CheCL induces some runtime overheads as shown in Section IV-A, more aggressive job scheduling enabled by CheCL may be able to improve the total system performance as a result.

In some cases, a running job may better stop using GPUs and give them to other jobs. AMD's OpenCL implementation supports use of CPUs as well as GPUs to comply with the OpenCL specification [8], even though NVIDIA OpenCL does not yet. Thus, we can expect that OpenCL applications can run with and without GPUs in a similar way. In this situation, CheCL allows an OpenCL process to stop using the GPU at runtime by recreating all OpenCL objects so as to use a CPU as a compute device. As a result, an appropriate processor can be selected at runtime for execution of each job so as to improve a given criterion such as performance and energy efficiency [15]. We have confirmed that CheCL with AMD OpenCL can achieve runtime processor selection by changing the compute device from a CPU to a GPU, and vice versa. In runtime processor selection, we do not need to save a checkpoint file in a hard disk but volatile memory devices such as a RAM disk provided by Linux. As shown in Table I, the RAM disk bandwidth of the system is much higher than the hard disk bandwidth; use of the RAM disk can significantly reduce the cost of changing the compute device from one to another.

In process migration, another important factor is the time for resuming OpenCL objects because it is included in the migration cost. Figure 7 shows the breakdown of the object recreation time on restart. It is clearly indicated that a considerable part of the time is spent for resuming cl_mem and cl_program objects that are abstractions of memory chunks and device programs, respectively. For resuming cl_mem objects, the user data that existed in the device memory before checkpointing must be sent to the device memory. Because of the data transfer from the host to the device, resuming memory objects is time-consuming and the time is in proportion to the data size. On the other hand, for resuming cl_program objects, each program must be recompiled and the recompilation can potentially require a long time. Figure 7 indicates that the breakdown of the object recreation time strongly depends on the underlying OpenCL implementation. In AMD OpenCL, the recompile time is often longer than NVIDIA OpenCL. Especially, the recompilation of S3D takes a long time because it uses 27 program objects. Although the time for recreating cl_platform_id and cl_context objects can be seen in NVIDIA OpenCL, it is negligible in AMD OpenCL.

Accurate estimation of the process migration cost is necessary to achieve dynamic job scheduling and runtime processor selection. As shown in Figures 5, 6, and 7, the total migration cost for checkpointing and restarting is roughly proportional to the checkpoint file size, but the time for
recompiling OpenCL program objects to restart a process may not. If the recompilation time is known a priori, the process migration cost can be estimated from the checkpoint file size and therefore from the total memory usage. The process migration time using CheCL is estimated by

\[ T_m = \alpha M + T_r + \beta, \]

where \( T_m \) is the process migration time, \( M \) is the size of a checkpoint file, \( \alpha \) is a system parameter mainly depending on the bandwidth of writing the checkpoint file, \( T_r \) is the recompilation time, and \( \beta \) is a system-specific constant value. As shown in Figure 8, the total of the checkpoint time and the restart time on each compute device can be estimated by the simple prediction model. If the performance difference between two nodes or between two compute devices for a process is large enough to justify the migration cost, the process should be migrated to a higher-performance node or compute device. Therefore, CheCL can be an infrastructure to explore dynamic job scheduling and runtime processor selection mechanisms based on the migration cost prediction.

D. Limitations

Although the current implementation of CheCL already supports various OpenCL features as demonstrated above, there are some limitations.

Based on the kernel function signature, CheCL checks if every argument set by \texttt{clSetKernelArg} is a CheCL handle or not. By parsing the parameter list of each kernel function, CheCL remembers a formal argument with a special keyword, such as \texttt{__global} and \texttt{__local}, that is supposed to receive an OpenCL handle. However, if a user-defined structure including CheCL handles is given to \texttt{clSetKernelArg} as an argument, CheCL overlooks the handles in the structure, even though they must be converted to OpenCL handles. Therefore, the current implementation of CheCL does not support such function arguments yet, and its OpenCL C code parser is under development to check if each user-defined structure includes OpenCL handles. For a similar reason, CheCL does not currently support callback functions that can be set to some OpenCL API functions such as \texttt{clBuildProgram}. CheCL just ignores those callback functions.

Use of \texttt{clCreateProgramWithBinary} is deprecated in CheCL, because the binary code used when being checkpointed is not always valid for the node, on which the process restarts. Besides, if a \texttt{clCreateProgramWithBinary} is used to create a \texttt{cl_program} object, the source code of kernel functions might not be available. In such a case, based on the memory address, CheCL estimates whether a given argument is a CheCL handle or not. However, there is a possibility that CheCL incorrectly converts a given address to another invalid address because the given address may accidentally coincide with the address of one CheCL handle.

If a \texttt{cl_mem} object is allocated with \texttt{CL_MEM_USE_HOST_PTR} option, the OpenCL implementation uses a given host memory region to cache device memory data. This OpenCL feature is available even in the current implementation of CheCL, but usually causes severe performance degradation. This is because the cached copy of such a memory object is sent to the device memory whenever the memory object is passed to a kernel, and the cached copy in the host memory is overwritten by the data in the device memory after the kernel execution. To reduce the performance degradation due to those redundant data transfers, some operating system supports such as used in GMAC [16] would be necessary. In addition, the OpenCL C parser of CheCL should be capable of checking if a memory object is modified by a kernel. This capability is necessary for CheCL to achieve incremental checkpointing of OpenCL objects, in which the data of OpenCL objects are written into a checkpoint file only if the objects are updated. As a result of reducing the data written to a checkpoint file, the checkpoint time will be significantly shortened. These will be examined in our future implementation.

V. RELATED WORK

CPR is a classic but still important research topic in high-performance and dependable computing. Therefore, numerous studies on effective CPR implementations have been reported so far [4].

In this paper, we use BLCR [2] to write the state of an application process to a file and resume the process from the file. As BLCR can fully access system software resources, it can include the executable and shared libraries of a running process in a checkpoint file. BLCR supports checkpointing a wide range of applications including multi-threaded and distributed applications. Thanks to the BLCR features, CheCL can work for applications written with both OpenCL and Open MPI[12].

Another promising CPR system implementation is Distributed MultiThreaded CheckPointing (DMTCP) [3]. Unlike BLCR, DMTCP is a user-level CPR system that does not use custom kernel modules for checkpointing. Although CheCL’s approach does not require any kernel support, DMTCP fails to checkpoint an OpenCL application with CheCL because DMTCP takes a checkpoint of a process and its children in default and thus fails in checkpointing an API proxy that uses OpenCL. If the API proxy is killed before checkpointing and restarted right after checkpointing, we have confirmed that DMTCP can also be used as the underlying CPR system of CheCL. This makes CheCL more portable because CheCL with DMTCP is available even if BLCR is not installed in the system.
The idea of intercepting API calls to record the state changes can be found in [17]. Wang et al. have proposed a job pause service that records all the state on one side of API calls [18]. These ideas are similar to the API proxy approach proposed in this paper. However, these ideas are applied to checkpointing MPI programs but not to GPU computing programs.

This paper employed CPR for more aggressive job scheduling in a heterogeneous GPU cluster system. Power-aware job scheduling in such a system has also been discussed in [19]. However, they do not consider migration of a running process yet.

Another major approach to such process migration would be to use virtual machines (VMs) such as Xen [20], because the VM technology enables to migrate virtual OS instances across distinct physical hosts [21]. Shi et al. have proposed vCUDA that enables CUDA applications to run on the guest OS on a VM [22]. However, vCUDA needs to redirect any API calls on the guest OS to the host OS, and causes severe performance degradation due to the encode/decode overheads for communication between the guest OS and the host OS.
Recently, some mechanisms capable of transparently using remote compute devices have been developed [23], [24]. Such a capability can also be incorporated into our CheCL implementation by, for example, allowing CheCL wrapper functions to communicate with a remote API proxy via TCP/IP sockets.

VI. CONCLUSIONS

This paper has presented a completely transparent CPR tool for OpenCL applications, named CheCL, which can checkpoint the GPU state without any application code modification and recompilation. Since CPR fails if a running process to be checkpointed uses any OpenCL objects, CheCL prevents an application process from directly accessing OpenCL objects. Every API call is once forwarded to another process called an API proxy that really uses OpenCL objects. Although the API proxy approach induces the runtime overheads, our evaluation results suggest that the overheads would be acceptable in practical applications well-designed for GPU computing. The additional overheads for handling OpenCL objects are quite small, and hardly increase the checkpoint time. Moreover, CheCL is also useful to migrate an OpenCL process from one computing node to another, resulting in performance improvement in terms of the total system throughput, the energy efficiency, and so on. Those mechanisms may be able to compensate the runtime overheads caused by CheCL and eventually improve the total system performance. Consequently, CheCL will become a powerful tool to enhance the dependability of GPU computing systems.

We are planning to apply CheCL to dynamic job scheduling and runtime processor selection, and to explore efficient mechanisms to assign each task to the best compute device among available CPUs and GPUs in the system. We will
also investigate effective ways to overcome the limitations of the current implementation described in this paper. These will be discussed in our future work.

ACKNOWLEDGMENTS

The authors would like to thank Wen-mei W. Hwu and Guochun Shi of the University of Illinois at Urbana-Champaign and Nacho Navarro of Barcelona Supercomputing Center for their valuable suggestions and comments on this work, and also Akira Nukada of Tokyo Institute of Technology for meaningful discussions on programming issues for checkpointing GPU computing applications.

This research was partially supported by Grants-in-Aid for Young Scientists (B) #21700049; NAKAYAMA HAYAO Foundation for Science & Technology and Culture; Core Research of Evolutional Science and Technology of Japan Science and Technology agency (JST); and Excellent Young Researchers Overseas Visit Program of Japan Society for the Promotion of Science (JSPS).

REFERENCES


