

# CheCUDA: A Checkpoint/Restart Tool for CUDA Applications

Hiroyuki Takizawa\*, Katsuto Sato\*, Kazuhiko Komatsu† and Hiroaki Kobayashi†

\* Graduate School of Information Sciences

Tohoku University

6-3 Aramaki-aza-aoba, Sendai, 980-8578 Japan

E-mail: tacky@isc.tohoku.ac.jp, katuto@sc.isc.tohoku.ac.jp

† Cyberscience Center

Tohoku University

6-3 Aramaki-aza-aoba, Sendai, 980-8578 Japan

E-mail: komatsu@sc.isc.tohoku.ac.jp, koba@isc.tohoku.ac.jp

**Abstract**—In this paper, a tool named CheCUDA is designed to checkpoint CUDA applications that use GPUs as accelerators. As existing checkpoint/restart implementations do not support checkpointing the GPU status, CheCUDA hooks a part of basic CUDA driver API calls in order to record the status changes on the main memory. At checkpointing, CheCUDA stores the status changes in a file after copying all necessary data in the video memory to the main memory and then disabling the CUDA runtime. At restarting, CheCUDA reads the file, re-initializes the CUDA runtime, and recovers the resources on GPUs so as to restart from the stored status. This paper demonstrates that a prototype implementation of CheCUDA can correctly checkpoint and restart a CUDA application written with basic APIs. This also indicates that CheCUDA can migrate a process from one PC to another even if the process uses a GPU. Accordingly, CheCUDA is useful not only to enhance the dependability of CUDA applications but also to enable dynamic task scheduling of CUDA applications required especially on heterogeneous GPU cluster systems. This paper also shows the timing overhead for checkpointing.

**Keywords**—Graphics processing units, compute unified device architecture, checkpoint/restart

## I. INTRODUCTION

GPU computing, which uses graphics processing units (GPUs) as accelerators, has become very popular in the field of high-performance computing (HPC), because GPUs have high floating-point operation rates and memory bandwidths. So far, many researchers have reported that various scientific and engineering applications can significantly be accelerated using GPUs[1]. However, GPU computing has just emerged, and several functions commonly used in HPC have not been supported yet in GPU computing. One of such unsupported functions is checkpoint/restart(CPR).

Lots of CPR implementations have been developed[2], [3], [4], [5]. However, as far as we know, there is no CPR implementation that supports checkpointing the GPU status, even though such a CPR tool can significantly enhance the dependability of GPU computing applications. Dependability will be a major concern of future GPU computing, because a GPU computing application is often executed on a commodity PC, which is not designed for GPU computing at all and sometimes becomes unstable due to the insufficient cooling capability and so on.

The CPR capability for GPU computing applications is also useful for dynamic task scheduling of heterogeneous GPU cluster systems, in which each computing node may have a different GPU. The CPR capability allows a GPU computing process to migrate from one computing node to another so as to improve the performance in terms of the total system throughput, the turn-around time, and/or the energy efficiency.

In this paper, we design and develop a CPR tool, named *CheCUDA*, for CUDA (Compute Unified Device Architecture) applications. CUDA is the most popular programming environment for GPU computing. This paper demonstrates that our prototype implementation of CheCUDA can correctly achieve CPR of CUDA applications. This paper also evaluates the overhead required for supporting CPR.

## II. RELATED WORK

### A. Checkpoint/Restart

Checkpointing is to write the status of a running process to a checkpoint file, from which the process status can be restored. Checkpointing 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 [5].

Berkeley Lab Checkpoint/Restart(BLCR) package [2] is one of actively developed CPR implementations for Linux systems. Based on custom kernel modules, BLCR supports checkpointing a wide range of applications including multi-threaded and distributed applications.

As with many other implementations, BLCR supports callback functions to provide a way to achieve CPR of a process, which needs to use resources not being checkpointed. Therefore, such a callback function is invoked when the running application is almost to be checkpointed or right after restarting. In addition, BLCR achieves task migration between two different nodes if the same shared libraries are available on both of them; a CUDA application may resume based on a checkpoint file generated on another node. Using these functions of BLCR, this paper presents a CPR tool for CUDA applications.

Another option to keep the status of a running process is to use virtual machines(VMs), such as Xen [6]. Shi et

```
CUdeviceptr ptr;
cuMemAlloc(&ptr, sizeof(float)*1024);
```

Figure 1. Device memory allocation.

al. have proposed vCUDA [7], which uses VM for CUDA applications. However, CPR is a lighter weight solution than VM. Hence, the snapshot size of a running process generated by CPR becomes much smaller than that by VM, resulting in a smaller overhead for dynamic task migration.

### B. GPU Computing with CUDA

NVIDIA’s CUDA[8] is the most popular programming framework released in 2007 that incorporates some additional keywords into the standard C/C++ programming language for GPU computing. As CUDA allows a programmer to access GPUs via APIs designed for GPU computing, it can handle the GPU hardware without tricky programming techniques, which are required at GPU programming with graphics APIs such as OpenGL and DirectX.

CUDA provides two kinds of APIs; CUDA Runtime APIs and CUDA Driver APIs. The latter is a lower-level API library than the former. To analyze the lower-level behaviors of GPU computing applications, this paper focuses on GPU computing applications written with CUDA Driver APIs, even though our approach will also be applicable to CUDA Runtime APIs.

In CUDA, a GPU is considered a *compute device* and has its own memory called device memory. A CPU is a *host* that controls the compute device. A typical CUDA application running on a host firstly initializes the compute device, allocates a device memory chunk, copies host memory data to the allocated device memory chunk, invokes a special function processing the device memory data, and transfers the results from the device memory.

In CUDA Driver APIs, there are several CUDA resources such as CUDA contexts, CUDA modules, and allocated device memory chunks. Here, a CUDA context is analogous to a CPU process, and a CUDA module is a dynamically loadable package of GPU code and data[9]. We can get a *handle*, which is an identifier bound to such a CUDA resource via API calls. For example, Figure 1 shows the API call that allocates a device memory chunk of `sizeof(float)*1024` bytes, and the handle that is the base address of the memory chunk is passed to `ptr`.

One difficulty in CPR of CUDA applications is that existing CPR systems such as BLCR can resume only the CPU status, but not the GPU status, i.e. CUDA resources. Although the existing CPR systems may restore the value of a handle, the restored handle is not actually bound to any CUDA resource, because a CUDA resource existing at the checkpointing time no longer exists at the restarting time. Moreover, if a CUDA context exists at the checkpointing time, the generated checkpoint file contains the information of the CUDA context. As a

result, restarting from the checkpoint file fails to restore the CUDA context<sup>1</sup>. For example, BLCR fails to restart when doing `mmap()` of the character device of a GPU, e.g. `/dev/nvidia0`. Accordingly, no CUDA context is supposed to exist when a process is being checkpointed. If there is no CUDA context, the process does not have any CUDA resources, and hence an existing CPR system can achieve CPR of the process.

### III. CHECKPOINT/RESTART OF CUDA APPLICATIONS

This section presents a CPR tool, named *CheCUDA*, for CPR of CUDA applications. Generally, it needs large amounts of manpower to maintain a whole CPR package. Moreover, development of CPR directly supporting CUDA will further increase the maintenance cost because CUDA evolves so quickly. Therefore, we decided to implement CheCUDA as an add-on package of BLCR; CheCUDA does not need to modify the original BLCR, which is responsible only to dump the host memory image to a file.

As described in the previous section, no CUDA context is supposed to exist at the checkpointing time, otherwise BLCR fails to restart. Hence, CheCUDA employs a workaround technique to allow BLCR to checkpoint the GPU status. If no CUDA resource exists at the checkpointing, BLCR can successfully restart the process. Therefore, one sure approach to allow a standard CPR implementation such as BLCR to checkpoint a CUDA application is to delete CUDA resources before checkpointing and to restore them right after checkpointing and restarting. In this case, there is no CUDA resource when checkpointing, and hence BLCR can restart the process. The procedure performed by CheCUDA at checkpointing is summarized as follows:

- 1) Copying all the user data in the device memory to the host memory.
- 2) Deleting all existing CUDA resources by destroying CUDA contexts.
- 3) Checkpointing, i.e. writing the current status of the application to a checkpoint file, and also writing the user data, which are copied onto the host memory in Step 1, into the checkpoint file.
- 4) Initializing the GPU and recreating CUDA resources that existed before Step 2.
- 5) Sending the user data back to the device memory.

At restarting, CheCUDA reads the checkpoint file, and then performs the above procedure from Step 4. To recover all the CUDA resources in Step 4, CheCUDA has to know how to recreate the resources. For example, the size of a device memory chunk is required to restore the chunk with `cuMemAlloc` (see Figure 1). In CheCUDA, when calling `cuMemAlloc`, a different function with the same name, i.e. a *wrapper function*, is first invoked to implicitly record

<sup>1</sup>CUDA contexts can be created separately and attached independently to different CPU threads. See a sample code in CUDA SDK named *threadMigration* for more details. Although `cuCtxPopCurrent` allows a CUDA context to be “floating” and not attached to any CPU thread, even such a CUDA context cannot exist at the checkpointing time.

such information. In the wrapper function, the original `cuMemAlloc` is invoked to actually allocate a device memory chunk, and then the base address and the size of the chunk are kept in a list, called a *resource management list*. CheCUDA manages the list to record all the allocated device memory chunks. The entry of an allocated device memory chunk is registered to the resource management list when calling `cuMemAlloc`, and removed from the list when calling `cuMemFree`. Based on the resource management list, CheCUDA can know the size of every device memory chunk allocated at checkpointing, and recreate it when restarting a CUDA process. In a similar way, most CUDA resources can be restored at restarting by recording how the resources are created.

In CheCUDA, a `CUdeviceptr`-type variable to hold a device memory address is also replaced with a *wrapper class* object. Every wrapper class object is registered to a list, referred to as an *object management list*. The entry of a wrapper class object is implicitly registered to the list by the constructor of the class, and removed by the destructor. If the base address of a device memory chunk reallocated at restarting has been changed from the previous one, the object management list is used to update the old base address stored in wrapper class objects.

Furthermore, CheCUDA has to restore the user data of every device memory chunk. Hence, in Step 1, CheCUDA duplicates all the user data into the host memory so as to write them to the checkpoint file in Step 3. In Steps 1 and 5, the resource management list is used to duplicate the user data of each device memory chunk. CheCUDA provides wrapper functions and classes for resource allocation API calls and handles to implicitly maintain the resource management lists and the object management lists of CUDA resources.

One remaining topic is when the checkpointing is carried out. This is important for CheCUDA due to three reasons. One is that CheCUDA's checkpointing according to the above procedure is time-consuming compared to the conventional ones, which do not consider the GPU status. Another is that a sequence of API calls might have to be atomic; checkpointing during the sequence might lead to a restart failure. The other reason is that CPU and GPU must be synchronized before checkpointing and the synchronization induces a certain performance overhead. Since GPU and CPU are always synchronized when `cuCtxSynchronize()` and some data transfer functions are invoked, CheCUDA performs checkpointing at such a synchronization point to avoid causing an extra synchronization overhead. If a CUDA program linked with CheCUDA has received `SIGUSR1` that is one of signals available on a Linux system, checkpointing is performed at the first synchronization since then. Upon arrival of `SIGUSR2`, CheCUDA immediately performs synchronization and checkpointing despite the overhead and the restart failure risk. In addition, CheCUDA provides user-defined non-preemptive checkpointing at a specific point in the source code, where a checkpointing function, `ckptSelf()`, is inserted.

```
#include "CheCUDA.h"
int main(int argc, char** argv)
{
    CUdevice hDev;
    CUcontext hCtx;

    // CUDA initialization
    cuInit(0);
    cuDeviceGet(&hDev, 0);
    cuCtxCreate(&hCtx, 0, hDev);

    ... CUDA code ...

    // CheCUDA's checkpointing
    ckptSelf();

    ... CUDA code ...

    cuCtxDetach(hCtx);
    return 0;
}
```

Figure 2. A simple CUDA code with CheCUDA.

Figure 2 shows a sample CUDA code with `ckptSelf()`. Basically, CheCUDA can achieve CPR of a CUDA program without modification of the original CUDA code, because wrapper functions and wrapper classes implicitly work. CheCUDA performs checkpointing in response to a signal, `SIGUSR1` or `SIGUSR2`. Only if a programmer needs to specify when to do checkpointing, `ckptSelf()` is inserted into the CUDA code.

#### IV. EVALUATION AND DISCUSSIONS

This section evaluates a prototype implementation of CheCUDA presented in Section 3. Our current CheCUDA implementation provides necessary wrapper classes and functions for hooking a part of basic CUDA Driver APIs to show the feasibility of CPR for CUDA applications. In this paper, BLCR version 0.81 is used for obtaining a snapshot of the host memory. The following evaluation results are obtained using the PCs listed in Table I. In the table, HDD Perf. means the average bandwidth of writing a 8GB file measured with a benchmark program named `Bonnie++` [10]. PCIe Perf.(HtoD) and PCIe Perf.(DtoH) are the average bandwidths for sending 64MB data from the host memory to the device memory, and from the device memory to the host memory, respectively.

##### A. Runtime Overhead for Resource Management

We first evaluate the runtime overhead induced by use of wrapper classes and functions in CheCUDA to maintain the necessary information for recreating CUDA resources. Figure 3 shows a part of a simple code used in the runtime overhead evaluation that just iterates allocation and deallocation of device memory chunks. Two executable files are generated by compiling and linking the simple code with the original CUDA library and CheCUDA, respectively. That is, the difference in execution time between the two executables indicates the runtime overhead of CheCUDA.

In the cases where many CUDA resources and their handles are used in a code, the runtime overhead of CheCUDA increases because CheCUDA implicitly keeps the information about them. Figure 4 shows the evaluation results obtained changing `nHandles` in Figure 3. The

Table I  
SYSTEM SPECIFICATIONS

	Desktop1	Desktop2	Laptop
CPU	Intel Core 2 Quad Q6600	AMD Phenom II X4 940	Intel Core 2 Duo T7250
GPU	NVIDIA GeForce GTX280	NVIDIA GeForce 8800 GTX	NVIDIA GeForce 8600M GT
Main Memory	4GB	2GB	2GB
Video Memory	1GB	768MB	256MB
HDD Perf.	49 MB/sec	46 MB/sec	33 MB/sec
PCIe Perf.(HtoD)	2.42 GB/sec	2.60 GB/sec	2.39 GB/sec
PCIe Perf.(DtoH)	1.52 GB/sec	2.36 GB/sec	0.45 GB/sec
OS	CentOS 5.3 (x86_64), kernel 2.6.18-128.1.14.el5		
CUDA	CUDA 2.2, driver 185.18		

```

CUdeviceptr d_A[nHandles];
unsigned int size_A = 256 * sizeof(float);

for(unsigned int h=0;h<nHandles;h++){
// allocate device memory
cuMemAlloc( &d_A[h], size_A );
}
for(unsigned int h=0;h<nHandles;h++){
// deallocate device memory
cuMemFree(d_A[h]);
}

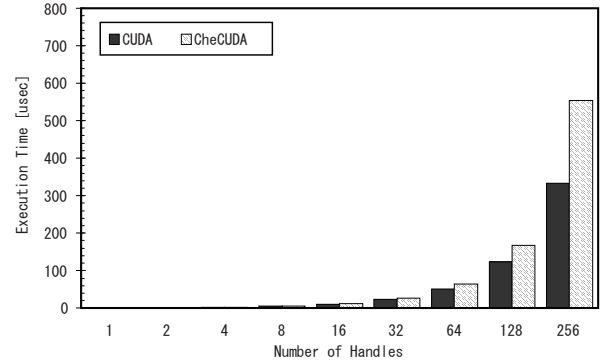
```

Figure 3. The code used for runtime overhead evaluation.

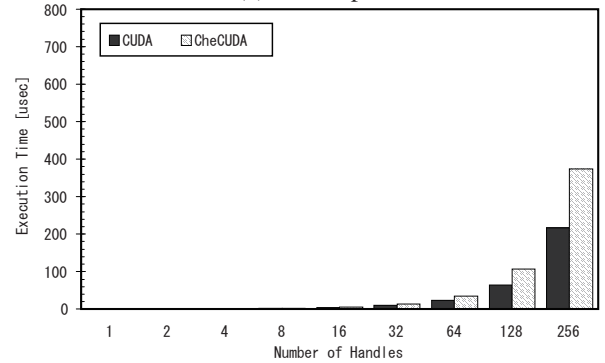
results demonstrate that use of wrapper functions increases the execution time of each API call by at most several microseconds. The overhead becomes smaller when the CPU performance is higher, because the wrapper class management is done by CPU. Allocation of each CUDA resource is usually performed only once and does not become a dominant factor in the total execution time. Accordingly, the runtime overhead of CheCUDA has only a small impact on performance in practical uses.

### B. Timing Overheads for Postprocessing, Checkpointing, and Postprocessing

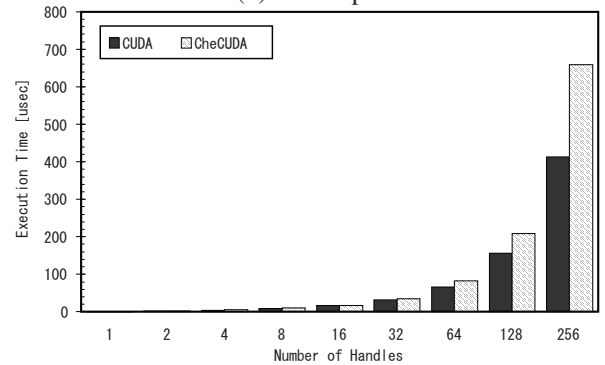
Two sample codes in NVIDIA CUDA SDK version 2.2, called `matrixMulDrv` and `simpleTextureDrv`, are used for the checkpointing overhead evaluation. Similarly to the code in Figure 2, `ckptSelf` are inserted in the original code of `matrixMulDrv`; `ckptSelf` is called after `cuLaunchGrid`, which calculates multiplication of two matrices. As `ckptSelf` internally calls `cuCtxSynchronize`, CPU and GPU are synchronized at the function call, and a checkpoint file is created immediately after the matrix multiplication on GPU is completed. We also modify the `matrixMulDrv` code so as to adjust the matrix size to measure the timing overheads of preprocessing, checkpointing, and postprocessing of the CheCUDA's checkpointing procedure. The preprocessing includes copying user data to the host memory and deleting CUDA resources. In the checkpointing, BLCR writes a snapshot of the running process to a file. The postprocessing indicates recovering CUDA resources and copying back the user data to the device memory. Also, `simpleTextureDrv` that rotates an image loaded from a file is used as another sample application that uses various CUDA resources, such as textures and CUDA arrays. Using these sample applications, the timing overheads for CPR are evaluated.



(a) Desktop1



(b) Desktop2



(c) Laptop

Figure 4. Runtime overhead for CUDA resource management.

Figures 5 and 6 show the evaluation results. Here, the execution times for matrix multiplication and texture image rotation are included in *others* of those figures, respectively. In Figure 5, the execution times for preprocessing and postprocessing are longer than the time for multiplying two matrices when the matrix size is small.

In Figure 6, the execution times for preprocessing and postprocessing are always longer than the time for rotating a texture image, because the execution time for texture image rotation is very short and negligible (usually less than 1 millisecond). Deletion and recreation of CUDA resources consume a considerable part of those additional overheads introduced by CheCUDA. In addition, those additional overheads of CheCUDA slightly increase with the total size of user data, i.e. matrices and texture images, because it is necessary to transfer the data in device memory to host memory before checkpointing, and to send them back to device memory after checkpointing. However, the overhead for checkpointing by BLCR is much larger than the additional overheads for the preprocessing and postprocessing by CheCUDA. This is mainly because writing data to a file on a hard disk is much more time-consuming than sending data between the host memory and the device memory, as shown in Table I. As a result, even in the case of low PCIe performance such as Laptop PC, the BLCR checkpoint consumes a much longer time than the preprocessing and postprocessing by CheCUDA. Accordingly, in practical uses, the additional overheads of CheCUDA for preprocessing and postprocessing are negligible compared to the checkpointing overhead by BLCR.

### C. Task Migration

We also migrate a running process of CUDA applications from one PC to another. Basically, BLCR supports task migration. Hence, in this paper, we examine that a checkpoint file generated by CheCUDA does not contain any host-dependent information, resulting in successful restart of the checkpointed process on another PC. In our evaluation, CheCUDA can achieve task migration among all the PCs listed in Table I. Therefore, it is clearly demonstrated that CheCUDA can enable task migration of CUDA applications. However, CheCUDA may fail in task migration in some situations not evaluated in this paper. For example, task migration fails if a checkpoint file is generated on a 32-bit Linux PC and then used to restart on a 64-bit Linux PC. BLCR terminates with an error message to notify that the checkpoint file does not match the kernel architecture. Since CheCUDA is an add-on package for BLCR, it always fails in task migration if BLCR cannot restart the task. Although this paper uses only three PCs to test the task migration capability, we will further investigate the requirements for successful CPR of CUDA applications, using various hardware configurations, operating systems, shared libraries, and applications.

## V. CONCLUSIONS

This paper has presented a CPR tool for CUDA applications, named CheCUDA, to checkpoint the GPU status. CheCUDA is designed as an add-on package for BLCR, which is one of the most popular CPR tools actively developed for checkpointing only the CPU status. As a result, CheCUDA can be developed separately from BLCR. Since CPR fails if a running process to be

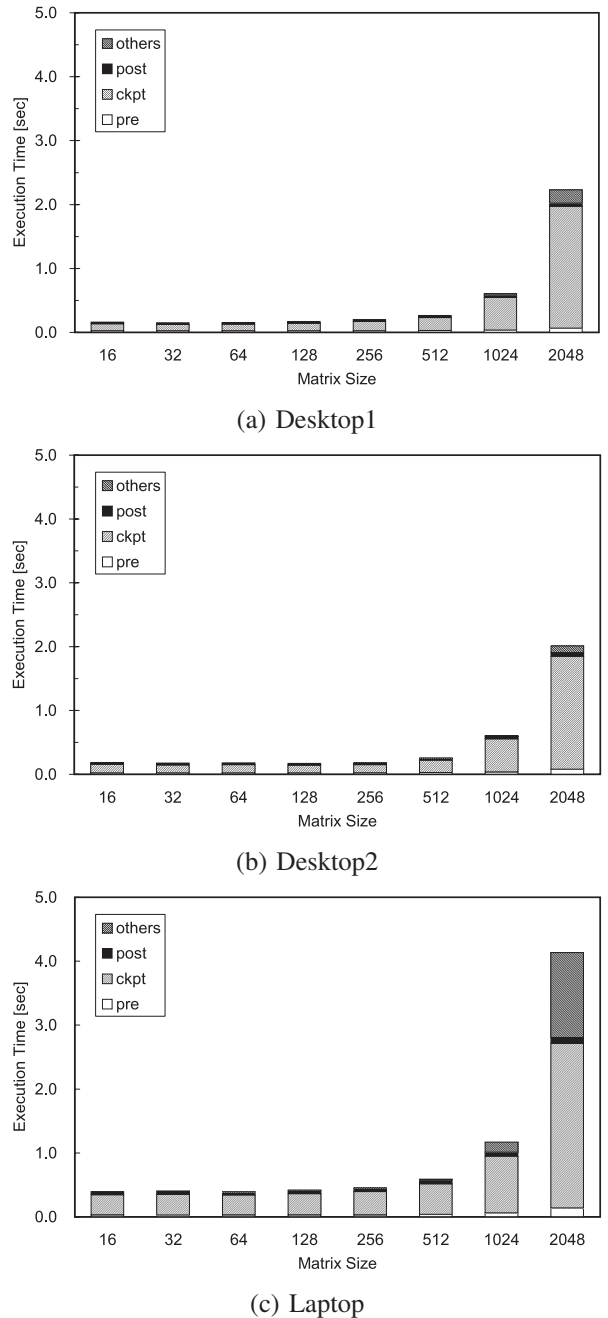


Figure 5. Timing Overheads for `matrixMulDrv`.

checkpointed uses CUDA resources, CheCUDA employs a workaround technique that once deletes all CUDA resources and recovers them after checkpointing. Thus, CheCUDA enables CPR of CUDA applications. Although the CheCUDA's approach obviously increases the time required for checkpointing, our evaluation results indicate that the additional overheads introduced by CheCUDA are negligible. Therefore, in practical uses, CheCUDA will become a powerful tool to enhance the dependability of CUDA applications with BLCR's checkpointing and small additional overheads. Moreover, it is also available to migrate a CUDA process running on one PC to another, resulting in the performance improvement in terms of the

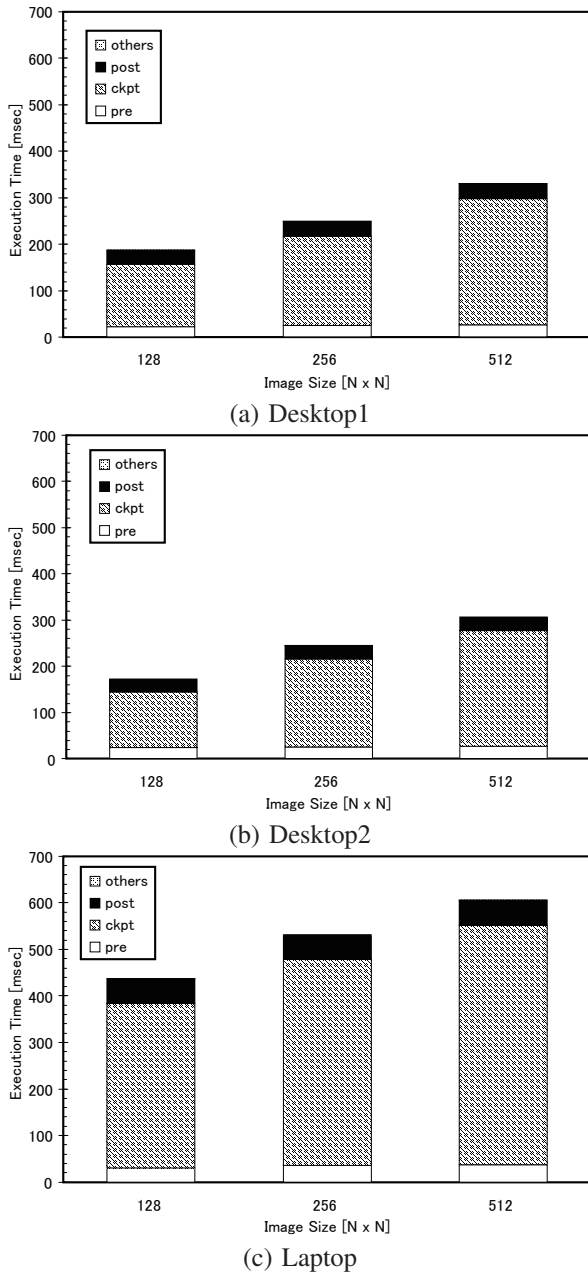


Figure 6. Timing Overheads for simpleTextureDrv.

total system throughput, the energy efficiency, and so on.

Since the current implementation of CheCUDA supports only a part of basic CUDA Driver APIs, we are now developing it to fully support all the CUDA Driver APIs. We will also test CheCUDA under various environments, and unveil the requirements for CheCUDA's CPR. Supporting

CUDA Runtime APIs in addition to Driver APIs will also be one important future work.

#### ACKNOWLEDGMENTS

This research was partially supported by Grants-in-Aid for Young Scientists(B) #21700049 and by NAKAYAMA HAYAO Foundation for Science & Technology and Culture.

#### REFERENCES

- [1] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
- [2] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," in *Proceedings of SciDAC 2006*, June 2006.
- [3] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, May 2009.
- [4] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *Proceedings of Usenix Winter 1995 Technical Conference*, Jan 1995, pp. 213–223.
- [5] E. Elnozahy and J. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97–108, 2004.
- [6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Prat, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 163–177.
- [7] L. Shi, H. Chen, and J. Sun, "vCUDA: Gpu accelerated high performance computing in virtual machines," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–11.
- [8] NVIDIA Corporation, "CUDA Zone – The resource for CUDA developers." [Online]. Available: [http://www.nvidia.com/object/cuda\\_home.html](http://www.nvidia.com/object/cuda_home.html)
- [9] —, "NVIDIA CUDA programming guide, version 2.2.1." [Online]. Available: [http://www.nvidia.com/object/cuda\\_develop.html](http://www.nvidia.com/object/cuda_develop.html)
- [10] R. Coker, "Bonnie++ project page." [Online]. Available: <http://www.coker.com.au/bonnie++/>