Two-Level Checkpoint/Restart Modeling for GPGPU

Supada Laosooksathit[1], Nichamon Naksinehaboon[2], Chokchai Leangsuksan[3]

Department of Computer Science, College of Engineering and Science
Louisiana Tech University, Ruston 71272, USA

[1] sla036@latech.edu
[2] nna033@latech.edu
[3] box@latech.edu

***Abstract*—Due to the fact that the reliability and availability of a large scaled system inverse to the number of computing elements, fault tolerance has become a major concern in high performance computing (HPC) including a very large system with GPGPU. In this paper, we propose a checkpoint/restart mechanism model which employs two-phase protocol and a latency hiding technique such as CUDA streams in order to achieve a low checkpoint overhead. We introduce GPU checkpoint and restart protocols. Also, we show experimental results and analyze the influences of the mechanism, especially in a long-running application.**

I. Introduction

Due to the ability to accelerate computation by parallelism, graphic processing units (GPUs) have long been deployed for graphic applications for decades [1]. Nowadays, supercomputing applications has increasingly explored power of GPU and the cluster of GPUs for non-graphic applications [2] [3]. Nevertheless, the more elements there are in the system, the lower reliability [4].

System failure can cause interruptions to jobs running on the system at time. For this reason, fault tolerance research on HPC has been actively discussed in recent years [4].

Checkpoint/restart is a fault tolerance mechanism that has been used in many system platforms. Instead of restarting computation from the beginning when a failure occurs, with checkpoint/restart, a process can be restarted from the last checkpoint on a healthier system [5] [6]. However, an overhead due to the checkpoint causes lower performance of the system [7].

We introduced our two-phase checkpoint work from [7] which presents a novel model of checkpoint/restart utilizing CUDA streamed and GPU virtualization. However, in this paper, our focus is particularly on checkpointing and restarting between GPU and CPU. We leave the checkpoint/restart on the CPU side to the underline checkpoint/restart mechanism.

Our experimental results reveal the performance improvement due to the latency hiding checkpoint/restart mechanism in three different aspects; checkpoint overhead, restart overhead, and wasted time. Our simulation shows the influence of the latency hiding checkpoint/restart mechanism to the performance of the long-running application.

The rest of this paper is organized as follows: The related works are discussed in Section II. Section III describes the streamed checkpoint/restart protocols. Section IV describes the experiments to study the behavior of both streamed and non-streamed checkpoint/restart mechanism. The simulation results are presented in Section V. Finally, the conclusion of this paper is stated in Section VI.

II. Related Works

The checkpoint/restart mechanism is a tool to improve the application resilience. By saving the state and considerable data in the checkpoint file, the process can be restarted from that state later. Therefore, the recomputing time - the time spent to re-execute the work due to a failure, can be reduced. There are many methods that have been implemented for checkpoint/restart [5] [8] [9] [10].

BLCR [9] is a checkpoint/restart mechanism for Linux systems. It is developed for checkpointing at the operating system-level, which allows the system preemption. Periodic and preemptive checkpointing can be used in response to the precursors of a possible failure.

VCCP [5] provides transparent checkpoint/restart mechanism for virtual machines (VMs). It uses a hypervisor based coordinated checkpoint/restart protocols so that the guest OS does not have to be changed. There are two protocols in hypervisor-based checkpoint/restart mechanism; VM checkpoint protocol and VM restart protocol. The experimental results indicate that VCCP generates less than 1 percent of additional overhead for non-communication intensive parallel applications.

For GPGPU fault tolerance, CheCUDA [6] is a checkpoint/restart mechanism for NVIDIA's CUDA (Compute Unified Device Architecture). However, it results in reduction of system performance due to the checkpoint overhead, particularly, for a large data set. This cost is mainly produced by memory transfer.

HiAL-Ckpt [11] is also a checkpoint/restart mechanism for GPGPU. However, it is implemented based on Brook+ programming language and allows the programmer to do checkpoint at the application level. Although its idea of

hierarchical checkpoint is similar to our previous work [7], the overhead optimization is not considered.

NVIDIA's CUDA has introduced a latency hiding technique, called stream. Since the GPU is idle during the memory transfer, this technique allows overlapping between kernel execution and memory transfer [12]. The performance models of CUDA stream are presented in [13]. They show that stream can reduce kernel execution time in case that the application consumes more time on memory transfer, and can eliminate some memory transfer time when the device is occupied on kernel execution. It also suggests that CUDA stream is most utilized when the data of the application are independent because stream can be applied on both host-to-device and device-to-host memory transfers.

III. PROTOCOLS

This section presents checkpoint and restart protocols of streamed checkpoint/restart (CPR) mechanism. Since a GPU system is a heterogeneous system, checkpointing and restarting require transferring checkpoint data between the CPU (host) and the GPU (device). We introduce a two-phase checkpoint/restart protocol to provide resilience in heterogeneous computing applications. Basically, the protocol must first checkpoint GPU kernel states to CPU memory and then from a source CPU memory to a target memory system or reliable storage. We, however, focus on the latency hiding checkpoint/restart protocol on the GPU side that is enhanced from [7].

Normally, a GPU works as a co-processor of a CPU. First, the data set has to be transferred from the host memory to the device memory. This process is called host-to-device memory copy. Once host-to-device memory copy finishes, the kernel on the device is invoked. After the kernel execution finishes, the result data are transferred back from the device memory to the host memory which is called device-to-host memory copy. Any failure that may occur while the kernel is executing will cause a loss of kernel computation. The GPU CPR mechanism potentially improves the application performance for a long-running kernel [7].

Fig 2 illustrates the streamed checkpoint protocol. The checkpoint overhead is a result of host memory allocation and device-to-host memory copy after a thread synchronization. With streams, the kernel continues executing while the checkpoint data are copied to the host. Once the memory copy finishes, the host dumps all the checkpoint data to a checkpoint file which is handled by the underline CPR mechanism. When using streams, the data set is chopped into chunks. The first chunk is copied to the host before the next kernel is invoked. Then, the subsequence chunk is copied while the kernel is executing after the first chunk transfer completes [13].

Fig 1 shows the streamed restart protocol. When a failure occurs while the kernel is executing, the device application context, including the device memory is destroyed. Therefore, to restore the application, the device context has to be recreated along with device memory, and redo host-to-device memory copy again. The restart process begins by reading the

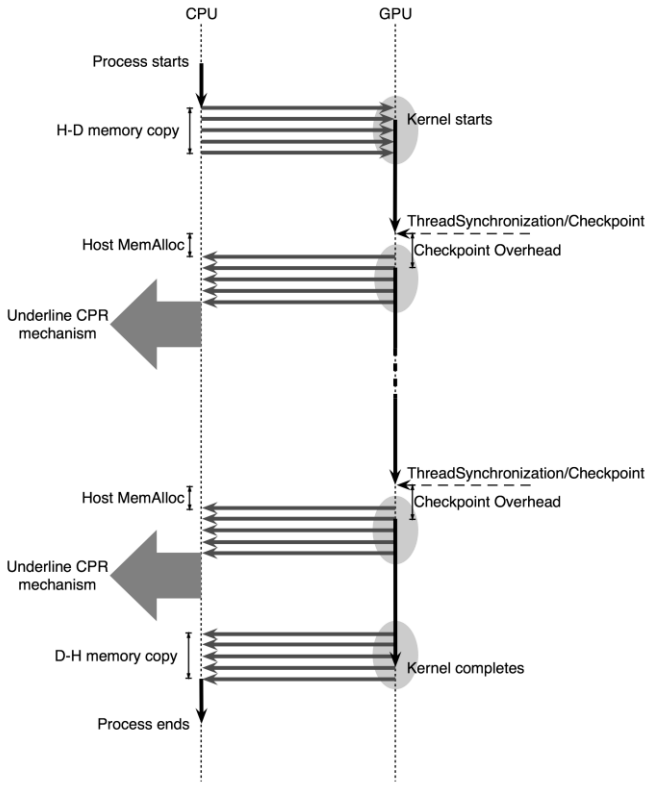


Fig 1. The checkpoint protocol for GPU streamed CPR

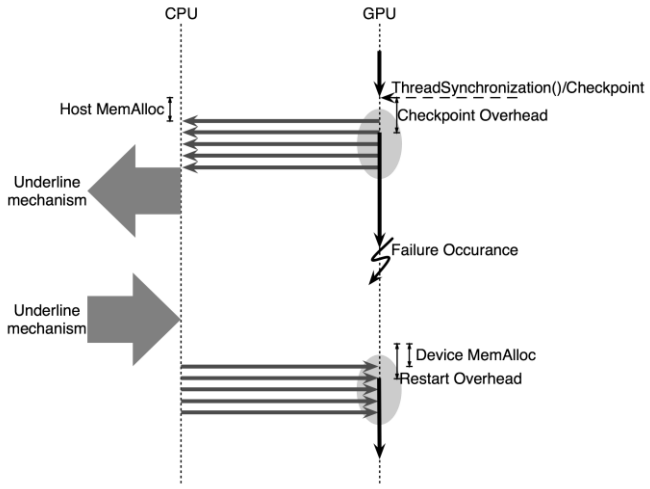


Fig 2. The restart protocol for GPU streamed CPR

checkpoint file to host memory. Since reading the checkpoint file is handled by the underline mechanism and the duration of this process depends on the speed of reading from the hard drive, I/O or networked storage, we do not consider it as the GPU restart overhead on the device. The restart overhead includes the device memory allocation and host-to-device memory copy. CUDA streams are beneficial by starting the kernel execution while the rest of checkpoint data is being transferred. The experiments based on our CPR protocols are presented in the next section.

IV. EXPERIMENTS

To study the cost of checkpoint and restart processes, we simulate the behavior of the protocols presented in the previous section and time the checkpoint overheads, restart overheads, and wasted time - the aggregate of checkpoint overhead, restart overhead and recomputing time, due to a failure. We do experiments on various sizes of a large array addition application with three types of GPU CPR mechanism: non-streamed, 4-streamed, 8-streamed.

Checkpoint and restart processes

```
1  // Begin computing
2  Do host-to-device memory copy of array A and
   B.
3  Execute kernel_1() with l iterations.

4  // Checkpoint process
5  Synchronize all threads to prepare for memory
   copy.
6  Allocate host memory for data checkpointing,
   i.e., for array A, B, and C.
7  Do device-to-host memory copy of array A, B,
   and C.

8  Execute kernel_2() with m iterations.

9  // Failure occurrence
10 Free all device memory.

11 // Restart process
12 Reallocate  device memory for array A, B, and
   C.
13 Do host-to-device memory copy of array A, B,
   and C.

14 // Recomputing time
15 Re-execute kernel_2() with m iterations.

16 Execute kernel_3() with n iterations.

17 Do device-to-host memory copy of the result
   array C.
```

Fig 3. The pseudo code imitating GPU streamed checkpoint and restart protocols

Fig 3 illustrates our simulation based on the protocols in Fig 2 and Fig 1. Since the checkpoint process is called at the host, the kernel of iterated array addition is split into three parts: `kernel_1()`, `kernel_2()` and `kernel_3()` which are prolonged by l, m, and n loops of iterative array addition ($C = A + B$), respectively. We assume that `kernel_1()` is the computation before a checkpoint, `kernel_2()` is the computation between the checkpoint and a failure, and `kernel_3()` is the computation after the failure until the application finishes. Then, when a failure occurs, it has to re-compute only `kernel_2()` instead of restarting from the beginning.

TABLE I
THE OVERHEADS OF DOING A CHECKPOINT WITH NON-STREAMED, 4-STREAMED, AND 8-STREAMED CPRS

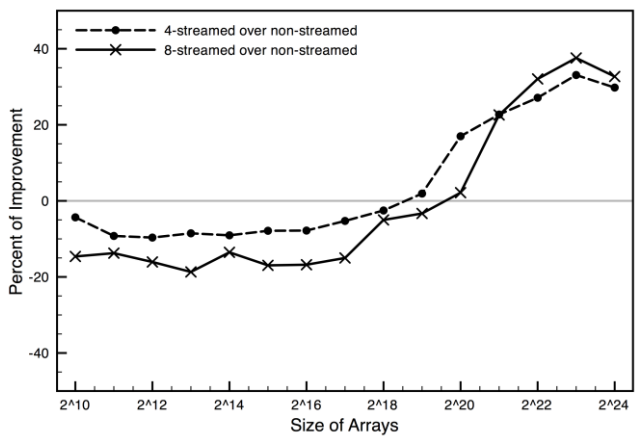
Array Size	Checkpoint Overhead (ms)		
	Non-streamed	4-streamed	8-streamed
2^{10}	5.116	5.337	5.812
2^{11}	5.086	5.555	5.795
2^{12}	5.098	5.590	5.924
2^{13}	5.159	5.600	6.108
2^{14}	5.239	5.713	5.940
2^{15}	5.404	5.829	6.285
2^{16}	5.679	6.123	6.640
2^{17}	6.256	6.586	7.196
2^{18}	7.390	7.577	7.770
2^{19}	9.579	9.396	9.917
2^{20}	14.475	12.016	14.171
2^{21}	22.726	17.568	17.613
2^{22}	41.075	29.938	27.930
2^{23}	80.754	54.034	50.436
2^{24}	147.704	103.702	99.220

TABLE II
THE RESTART OVERHEADS DUE TO A FAILURE OCCURRENCE ON AN APPLICATION WITH NON-STREAMED, 4-STREAMED, AND 8-STREAMED CPRS

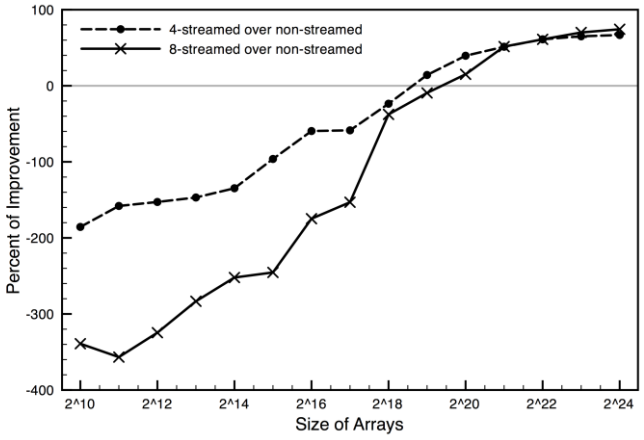
Array Size	Restart Overhead (ms)		
	Non-streamed	4-streamed	8-streamed
2^{10}	0.437	1.248	2.099
2^{11}	0.490	1.264	2.160
2^{12}	0.491	1.241	2.111
2^{13}	0.502	1.240	1.923
2^{14}	0.521	1.223	1.862
2^{15}	0.577	1.132	2.088
2^{16}	0.716	1.141	1.965
2^{17}	0.975	1.546	2.451
2^{18}	1.427	1.763	1.982
2^{19}	2.403	2.063	2.622
2^{20}	4.330	2.620	3.682
2^{21}	8.140	3.963	3.931
2^{22}	15.826	6.160	6.163
2^{23}	31.178	10.899	9.322
2^{24}	61.751	20.537	15.904

TABLE III
THE WASTED TIMES DUE TO A FAILURE OCCURRENCE ON AN APPLICATION WITH NON-STREAMED, 4-STREAMED, AND 8-STREAMED CPRS

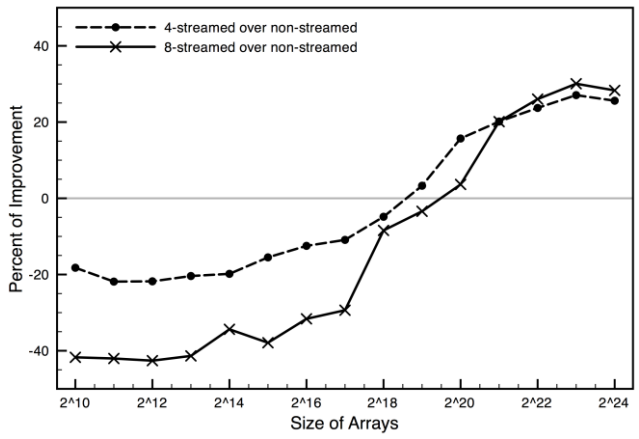
Array Size	Wasted Time (ms)		
	Non-streamed	4-streamed	8-streamed
2^{10}	5.665	6.698	8.025
2^{11}	5.688	6.931	8.067
2^{12}	5.701	6.943	8.147
2^{13}	5.776	6.954	8.146
2^{14}	5.936	7.112	7.978
2^{15}	6.322	7.302	8.714
2^{16}	6.964	7.834	9.175
2^{17}	8.255	9.156	10.671
2^{18}	10.803	11.326	11.738
2^{19}	15.913	15.389	16.469
2^{20}	26.573	22.404	25.620
2^{21}	46.352	37.016	37.029
2^{22}	87.749	66.946	64.941
2^{23}	173.624	126.625	121.450
2^{24}	332.888	247.673	238.557



(a)



(b)



(c)

Fig 4. Percentages of performance improvement in terms of (a) checkpoint overhead, (b) restart overhead due to a failure occurrence, and (c) wasted time due to a failure occurrence

Our experiments are done on the NVIDIA GeForce GTX 295 system which has compute compatibility 1.3. The heuristics of NVIDIA's graphic card with compute compatibility 1.x indicates that the maximum number of blocks in a grid is 65535, and the maximum number of threads in a block is 512 [12]. Then the maximum number of threads there can be is 65535×512, or $2^{25} - 2^{9}$. For the purpose of load balancing and data correctness while invoking the kernel with streams, we vary the size of arrays from 2^{10} to 2^{24}. If the size of arrays reaches 2^{25}, the number of iterations in the kernel will be influenced.

In non-streamed case, the memory copy and the kernel execution are done consecutively. However, in 4-streamed and 8-streamed cases, those instructions are done simultaneously. The checkpoint overhead is obtained by timing host memory allocation and device-to-host memory copy. A restart overhead is obtained by timing device memory reallocation and host-to-device memory copy. The wasted time is obtained by the summation of those overheads and the time of `kernel_2()` execution. The results are shown in Table I to III and Fig 4.

Table I shows the overheads caused by a checkpoint, comparing between non-streamed, 4-streamed, and 8-streamed CPRs. When the size of arrays is less than 2^{19}, 4-streamed CPR does not show advantage over non-streamed CPR. However, as the size of arrays is at least 2^{19}, the checkpoint overhead of 4-streamed CPR is less than that of non-streamed CPR. Moreover, when the size of arrays is greater than 2^{21}, the checkpoint overhead of 8-streamed CPR is the smallest. This corresponds to the graph in Fig 4 (a), which illustrates the percentage of the performance improvement of streamed CPR in term of checkpoint overhead. When the size of arrays is less than 2^{19}, the percentage of improvement is negative since the streamed CPR has no advantage over non-streamed CPR. However, it becomes positive when the size of arrays is 2^{19}. As the size of arrays is 2^{24}, 4-streamed and 8-streamed CPRs gain advantage over non-streamed CPR with the percentages of 29.790 and 32.699, respectively.

Table II describes restart overheads of non-streamed, 4-streamed, and 8-streamed CPRs due to a failure occurrence. It corresponds to the graph in Fig 4 (b). Similar to the checkpoint overhead, the restart overhead of 4-streamed CPR is smaller than non-streamed CPR when the size of arrays is at least 2^{19}. Also, the restart overhead of 8-streamed CPR gains advantage over 4-streamed CPR when the size of arrays is at least 2^{21}. As the size of arrays is 2^{24}, the percentages of improvement of 4-streamed and 8-streamed CPRs are 66.743 and 74.305, respectively. This similarity is because both checkpoint and restart overheads depend mainly on the duration of memory copy.

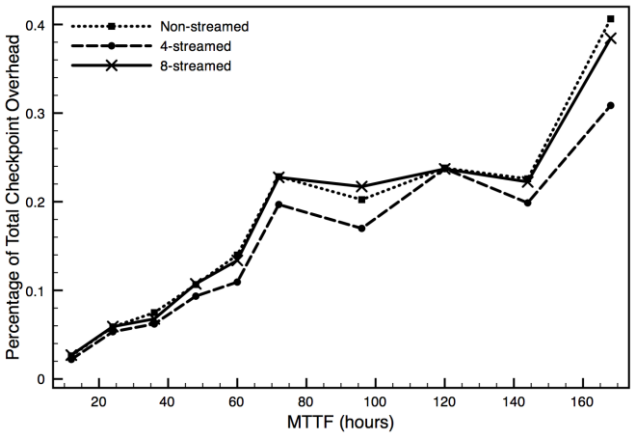
Table III shows the wasted times due to a failure occurrence on an application with non-streamed, 4-streamed, and 8-streamed CPRs. This table corresponds to the graph in Fig 4 (c). Since wasted time is the aggregate of checkpoint overhead, restart overhead, and recomputing time, the performance improvement in the aspect of wasted time is similar to the checkpoint and restart overheads as we have already stated.

In our experiments, we study the benefits of streamed CPR mechanism with only one checkpoint and one failure. In the next section, the simulation and its results will be presented in

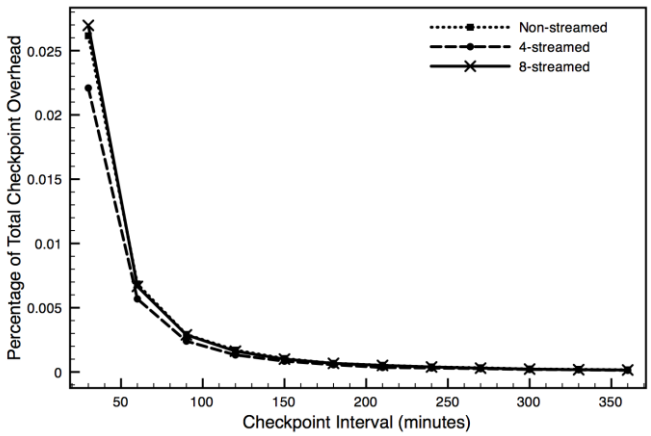
order to study the benefit of streamed CPR on real-world applications.

V. Simulation

In the previous section, we have shown the performance improvement of streamed CPR with various sizes of arrays. In this section, we study fault tolerance performability of streamed CPR, based on the checkpoint and restart overheads achieved in the experiments.



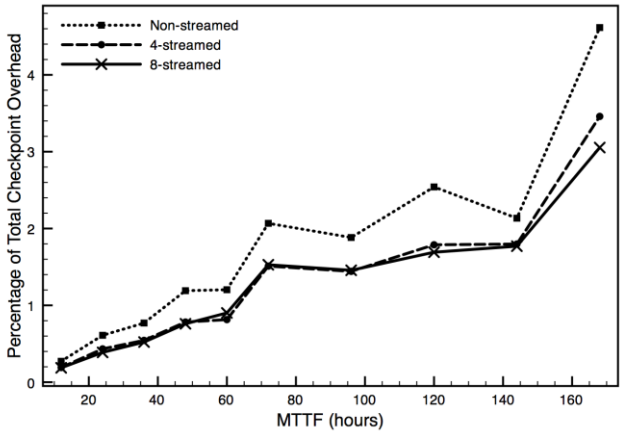
(a)



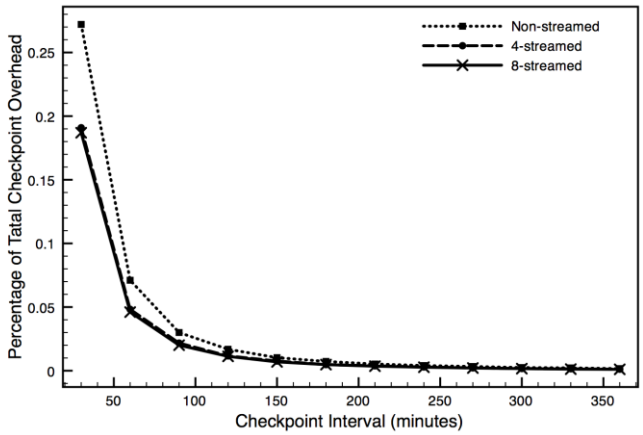
(b)

Fig 6. The percentages of total checkpoint overheads compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs when the size of arrays is 2^{20} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours

In our simulation, despite of various checkpoint intervals, we also concern about mean-time-to-failures (MTTFs). Due to the fact that a large scale GPU cluster system is a heterogeneous system with a significant number of GPUs, such as Tianhe-1A [14], the MTTF of the system depends on the MTTF of some other modules or nodes in the system. Our study varied the MTTFs from 12 hours to 7 days with the checkpoint intervals of 30 and 120 minutes. The application length is fixed at 1000 hours. Since both 4-streamed and 8-streamed CPRs have advantage over non-streamed CPR when the size of arrays is over 2^{19}, we do the simulation for the array sizes of 2^{20} and 2^{24}. The performance is observed in three different aspects; the percentage of total checkpoint overheads and total restart overheads compared to wasted time, and the percentage of wasted time compared to completion time, which is the aggregate of the application length and wasted time. Fig 6 - Fig 5 show the percentages of total checkpoint overheads, while Fig 7 - Fig 8 illustrate the percentages of total restart overheads, and Fig 10 - Fig 11 show the percentages of wasted time.



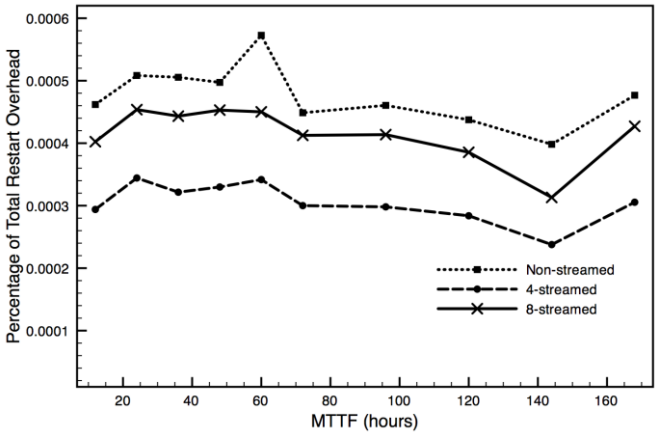
(a)



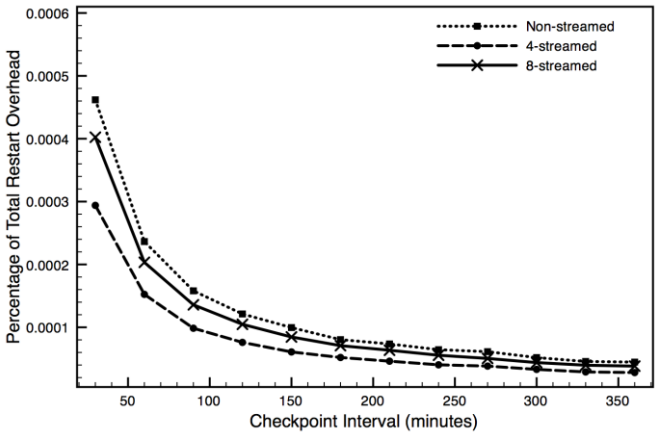
(b)

Fig 5. The percentages of total checkpoint overheads compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours

Fig 6 illustrates the percentages of total checkpoint overheads compared to wasted time when the size of arrays is 2^{20}. As shown in Fig 6 (a), when the MTTF is larger while the checkpoint interval is fixed at 30 minutes, the percentages of total checkpoint overheads grow bigger since the number of failures decreases while the number of checkpoints is more likely the same. On the other hand, when fixing the MTTF at 12 hours and varying the checkpoint intervals, as shown in Fig 6 (b), the larger interval produces fewer checkpoints. As a result, the total checkpoint overheads decrease. Furthermore, the percentages of total checkpoint overheads of 4-streamed

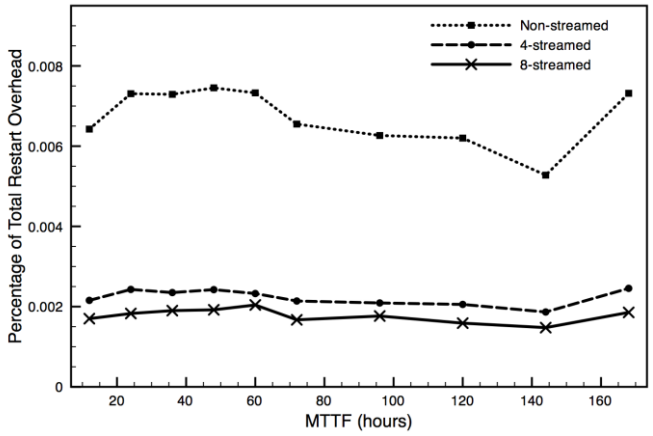


(a)

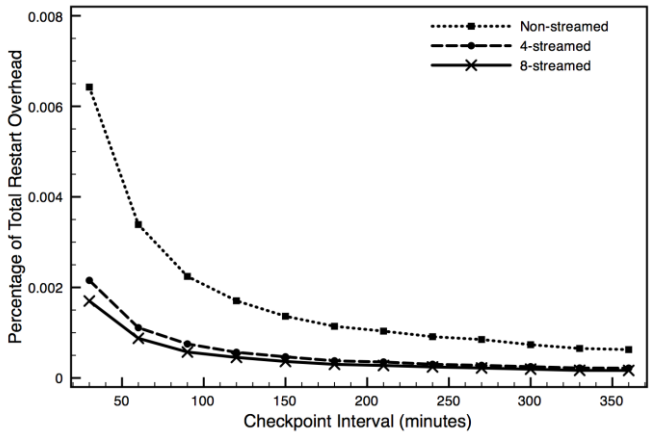


(b)

Fig 7. The percentages of total restart overheads compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs when the size of arrays is 2^{20} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours



(a)



(b)

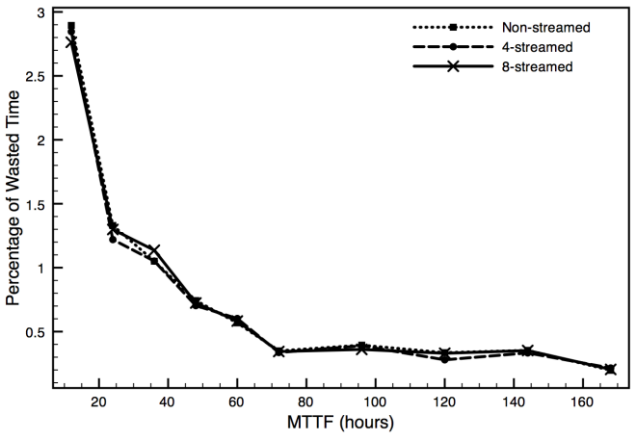
Fig 8. The percentages of total restart overheads compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours

CPR are the smallest in both cases which corresponds to our experimental results in the previous section.

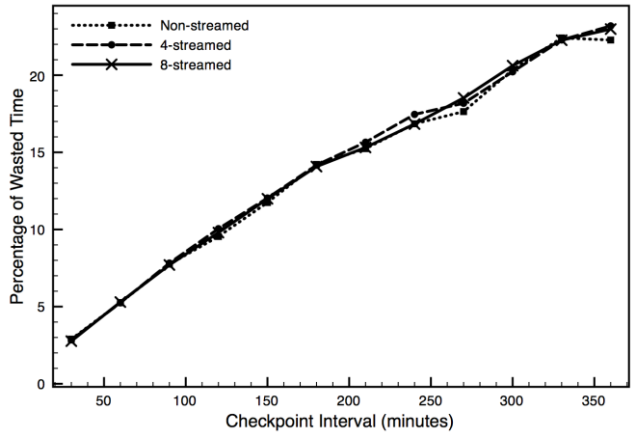
Fig 5 illustrates the percentages of total checkpoint overheads compared to wasted time when the size of array is 2^{24}. According to table I, when the size of arrays is 2^{24}, the checkpoint overhead of 8-streamed CPR is the smallest. As a result, 8-streamed CPR performs better than non-streamed and 4-streamed CPRs in term of total checkpoint overheads, particularly when there are more checkpoints. Similar to the graphs in Fig 6, the CPRs with the checkpoint interval of 30 minutes, shown in Fig 5 (a), produce larger total checkpoint overheads when the MTTF increases. Also, with various checkpoint intervals, shown in Fig 5 (b), the total checkpoint overheads are smaller for the larger checkpoint interval. Moreover, as non-streamed CPR produces the most checkpoint overheads in both cases; the total checkpoint overheads are very small compared to the wasted time with no more than 5 percent.

Fig 7 and Fig 8 illustrates the percentages of total restart overheads compared to wasted time when the sizes of arrays are 2^{20} and 2^{24}, respectively. Since the number of failures depends on MTTF, when the MTTF increases, both restart overheads and wasted time decreases. As a result, the graphs slightly change (as shown in Fig 7 (a) and Fig 8 (a)). On the other hand, when the MTTF is fixed at 12 hours, by varying the checkpoint intervals (as shown in Fig 7 (b) and Fig 8 (b)), the graphs drop due to the increase of checkpoint overheads, resulting in the increase of wasted time. Furthermore, in both cases, streamed CPR obviously performs better than non-streamed CPR. However, the restart overheads do not have much effect to the performance of the application since the percentages of total restart overheads are less than 0.01 percent of wasted time in any cases.

Fig 10 and Fig 11 illustrates the percentages of wasted time compared to completion time, which is the aggregate of wasted time and application length, when the sizes of arrays are 2^{20} and 2^{24}, respectively. When the checkpoint interval is fixed 30 minutes (shown in Fig 10 (a) and Fig 11 (a)), as the MTTF increases, the percentage of wasted time tends to be smaller. However, when the MTTF is fixed at 12 hours

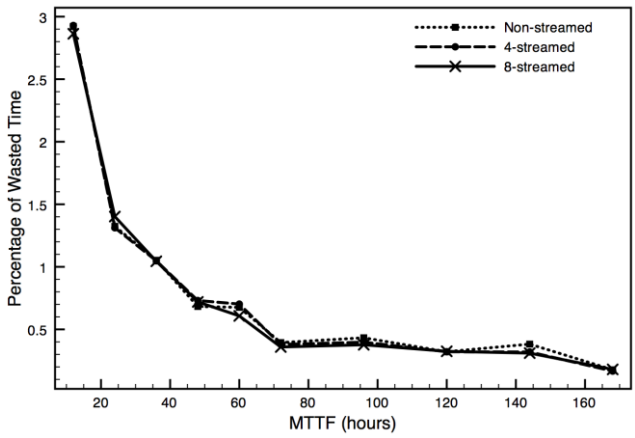


(a)

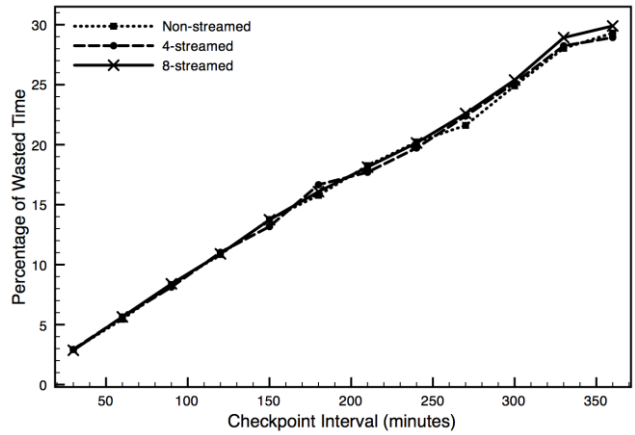


(b)

Fig 10. The percentages of wasted time compared to completion time of the application with non-streamed, 4-streamed, and 8-streamed CPRs when the size of arrays is 2^{20} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours



(a)



(b)

Fig 11. The percentages of wasted time compared to completion time of the application with non-streamed, 4-streamed, and 8-streamed CPRs when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours

(shown in Fig 10 (b) and Fig 11 (b)), as the checkpoint interval increases, the percentage of wasted time also increases due to the more recomputing time. Nevertheless, the overheads of non-streamed, 4-streamed, and 8-streamed CPRs are relatively small comparing to the recomputing time and completion time. Thus, the difference of the wasted time between those three types of CPRs is insignificant.

VI. Conclusion

Even though the checkpoint/restart mechanism can improve the application resilience, it reduces the performance by the cost of doing checkpoint. In this paper, we proposed GPU checkpoint/restart protocol that aims to reduce the fault tolerance overhead. Our experiments have revealed that the streamed CPR can reduce the checkpoint overhead when the size of checkpoint data is large enough. It can also improve the restart process by reducing the restart overhead.

Our simulation has shown that the streamed CPR can reduce the cost of doing checkpoint. However, in the long-running application, since the overhead of both checkpoint and restart processes are relatively small comparing to the recomputing time and the completion time, streamed CPR may not be beneficial on a single GPU system.

Since the proposed checkpoint/restart mechanism needs thread synchronization in order to ensure the data correctness when performing checkpoint, we plan to apply our mechanism with data scheduling in compiler level in order to perform checkpoint at the proper time in the future work.

Acknowledgment

This work was partially supported by the grants CNS-0834483 and EPS-1003897 from the National Science Foundation.

REFERENCES

[1] (2010) General-Purpose Computation on Graphics Hardware. [Online]. http://gpgpu.org

[2] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover, "GPU Cluster for High Performance Computing," *SC Conference*, vol. 0, p. 47, 2004.

[3] Volodymyr V. Kindratenko et al., "GPU clusters for high-performance computing," *CLUSTER*, pp. 1-8, 2009.

[4] HPC Resilience Consortium Wiki. [Online]. http://resilience.latech.edu/

[5] Hong Ong, Natthapol Saragol, Kasidit Chanchio, and Chokchai Leangsuksun, "VCCP: A Transparent, Coordinated Checkpointing System for Virtualization-based Cluster Computing," *IEEE Cluster 2009*.

[6] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi, "CheCUDA: A Checkpoint/Restart Tool for CUDA Applications," in *PDCAT*, 2009, pp. 408-413.

[7] Supada Laosooksathit et al., "Lightweight Checkpoint Mechanism and Modeling in GPGPU Environment," in *4th Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2010)*, 2010.

[8] John W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530--531, September 1974.

[9] Paul H Hargrove and Jason C Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, p. 494, 2006.

[10] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw, "Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification," University of Virginia, Charlottesville, VA, USA, 1997.

[11] Xinhai Xu, Yufei Lin, Tao Tang, and Yisong Lin, "HiAL-Ckpt: A hierarchical application-level checkpointing for CPU-GPU hybrid systems," in *Computer Science and Education (ICCSE)*, 2010, pp. 1895-1899.

[12] NVIDIA, NVIDIA CUDA Programming Guide, 2010.

[13] Supada Laosooksathit, Chokchai Leangsuksun, Abdelkader Baggag, and Clayton Chandler, "Stream Experiments: Toward Latency Hiding in GPGPU," in *Parallel and Distributed Computing and Networks (PDCN)*, 2010.

[14] TOP500 Supercomputing Sites. [Online]. http://top500.org/