International Journal of Networked and Distributed Computing, Vol. 1, No. 4 (November 2013), 196-212

A Checkpoint/Restart Scheme for CUDA Programs with Complex Computation States

Hai Jiang [1] , Yulu Zhang [1] , Jeff Jenness [1] , Kuan-Ching Li [2]

[1] Department of Computer Science,
Arkansas State University, USA
E-mail: {hjiang, yulu.zhang, jeffj}@cs.astate.edu

[2] Department of Computer Science and Information Engr.,
Providence University, Taiwan
E-mail: kuancli@pu.edu.tw

Received 20 February 2013

Accepted 15 July 2013

Abstract

Checkpoint/restart has been an effective mechanism to achieve fault tolerance for many long-running scientific applications. The common approach is to save computation states in memory and secondary storage for execution resumption. However, as the GPU plays a much bigger role in high performance computing, there is no effective checkpoint/restart scheme yet due to the difficulty of the GPU computation state handling. This paper proposes an application-level checkpoint/restart scheme to save and restore GPU computation states in annotated user programs. A pre-compiler and run-time support module are developed to construct and save states in CPU system memory dynamically, whereas secondary storage can be utilized for scalability and long-term fault tolerance. CUDA programs with complicated computation states are supported. State-related variables dissipated in various memory units are collected. Both stack and heap are duplicated at application level for state construction. Experimental results have demonstrated the effectiveness of the proposed scheme.

Keywords: GPU, CUDA, checkpoint/start, fault tolerance.

1. Introduction

High Performance Computing (HPC) systems are usually used to solve more complex problems and many long-running HPC applications are more likely to encounter failures than regular applications. Both hardware and software failures may cause the loss of hours or days of computation[1]. Therefore, fault tolerance is a quite important issue in HPC and many data centers provide tools to support this[2,3]. The most popular fault tolerance technique is checkpoint-restart, which periodically saves the computation state into checkpointing files in stable storage units. Once a failure happens, the early-saved computation state will be loaded and the execution will be resumed from the last checkpoint.

The Graphics Processing Unit (GPU), originally used for computer graphics, has become a new computing platform for parallel and high performance computing. The NVIDIA GPU product families such as Tesla, Fermi and Kepler were designed from the ground up for parallel computing/programming and offer exclusive high performance computing features[4]. Because of their significant performance,

many top supercomputers have adopted GPUs for scientific applications. Fault tolerance is also critical to these GPU applications, especially in large scale GPU clusters[5]. However, there is no mature checkpoint/restart mechanism in the GPU computing field.

Checkpoint/restart for traditional CPU computations can be accomplished at three levels: kernel, library and application levels where computation states are acquired or constructed[6]. However, these strategies cannot be applied to GPUs since they are treated as devices and GPU computing states are not available to users[7]. Once a kernel is launched to the GPU, it will remain there until the execution is finished. Therefore, GPU applications cannot be stopped and re-scheduled. Also, if GPU jobs crash, they have to be re-run from the very beginning. Fault tolerance is not supported so far. Moreover, GPU job scheduling is implemented in the drivers which are under control of vendors such as NVIDIA and AMD. Even operating system vendors cannot solve the scheduling and fault tolerance issues. Therefore, without operating system and system call/library support, we can only adopt application-level checkpoint/restart approach for GPUs.

This paper intends to propose a checkpoint/restart scheme that consists of a precompiler and a run-time support module. The precompiler transforms the user's source code so that the run-time support module can be properly invoked to collect, store and reload the computation state. This paper makes the following contributions:

- A new infrastructure is developed so that the precompiler transforms both CPU and GPU source code and inserts library calls for the run-time support module to construct computation states dynamically.
- Data structures and mechanisms are proposed to collect computation states represented by variables spread in GPU registers, local memory, shared memory and global memory. States are saved in page-locked CPU memory for performance, which can be moved to secondary storage for scalability.
- Experiments have been conducted to determine the type of page-locked memory for state storage and overall checkpoint/restart overheads. The results have demonstrated the effectiveness of the proposed scheme.

The remainder of this paper is organized as follows: Section 2 briefly introduces CUDA programing and GPU memory hierarchy. In Section 3, the proposed checkpoint/restart scheme is described in detail. In Section 4, experimental results are provided for further system analysis. The related work is given in Section 5. Finally, the conclusion and future work are given in Section 6.

2. Background

2.1. CUDA Programming

Although the GPU was initially developed for computer graphics, developers found it to be quite powerful for data parallel applications. Subsequently, applications started using graphics API such as OpenGL[8] and DirectX[9] to achieve parallel processing indirectly. In 2006, NVIDIA released CUDA (Compute Unified Device Architecture) as a new programming paradigm to help utilize NVIDIA GPUs in a much easier way[10]. C/C++ and FORTRAN programmers can write programs in familiar languages while using the GPU directly. The CUDA platform includes run-time and driver APIs.

A CUDA program consists of one or more execution stages distributed on either the host (CPU) or the device (GPU). The stages with rich data parallelism are implemented in device code, whereas other stages are implemented in host code. The NVIDIA C compiler (nvcc) separates host and device code during the compilation process. For C programs, the host code is further compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written in an ANSI C extension with keywords for labeling data-parallel functions, called kernels, and their associated data structures[12]. The device code is typically further compiled by nvcc and executed on a GPU device.

A typical CUDA program contains four steps. First, data is copied from host memory to GPU memory which is most likely the global memory. Second, the CPU code launches a process on the

GPU by calling a kernel function. Third, the GPU executes the kernel function in parallel among GPU cores. Fourth, the computation results are copied back from GPU global memory to host memory.

2.2. GPU Memory Hierarchy

There are several types of memory available on the GPU, making GPU checkpoint/restart processing quite complicated. User defined variables can be spread out in registers, local memory, shared memory, global memory, constant memory and texture memory. Most CUDA applications only access global, local and shared memory. Therefore, texture and constant memory will not be discussed here.

- Registers: Accessible for read and write, local to each thread. This is the fastest memory available, but it is very limited per thread.
- Local memory: Accessible for read and write, private to individual threads. It saves the auto variables that the register spills.
- Shared memory: Accessible for read and write by all threads in a block. The amount of shared memory required for a block effects the level of occupancy obtainable.
- Global memory: Accessible for read and write operations by all thread processors in one grid. However, access to this memory is very slow [11].

2.3. Checkpoint/Restart

Checkpoint/restart is one of the primary techniques for achieving fault tolerance for long-running scientific applications. Usually, the snapshot of the application state is constructed and saved in secondary storage periodically. Once a failure occurs, the application state is reloaded in order to resume the execution from the last saved checkpoint. For applications on the CPU, checkpoint/restart can be accomplished at three levels: kernel, library and application levels[13].

At the kernel level, the operating system can construct the state[14]. This process is usually transparent to application developers[15]. The representative systems of offering kernel level checkpoints include V-System[16] and Charlotte[17].

At the library level, library functions provide checkpointing and recovery system calls to perform extraction of computation states. For instance, in the BLCR (Berkeley Lab Checkpoint/Restart) library, method *cri_syscall()* is used to invoke the *ioctl()*, which allows BLCR to save register contents onto the stack for resuming execution[19,20].

At the application level, applications checkpoint themselves and restart from a checkpoint position[21]. Since the source code needs to reconstruct the state, it might require a complete program rewriting or extensive modification[22,23].

For GPU applications, however, the computation state is controlled by device drivers and not the OS. Unfortunately, there is no access to the driver source code for inserting state preserving code. Thus, the state cannot be obtained at the kernel level. Moreover, no API is currently provided to extract the state from CUDA, so the library level approach does not work either. As a result, application-level checkpoint/restart is the only option thus far.

3. GPU Checkpoint/Restart Scheme

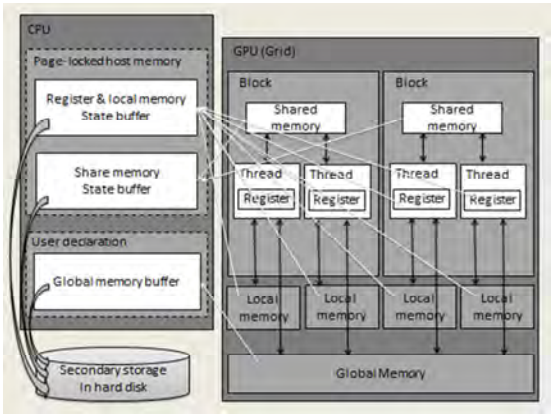

Fig. 1. Collecting and Saving Computation State of GPU Applications

To checkpoint/restart a GPU application, the computation state is the key. Such a state is collected/constructed at a checkpointing event and restored at a later restarting event[24]. The state of a GPU application can be represented by variables declared in the program. However due to the complex memory hierarchy of the GPU, those variables are

spread in different memory locations: register, local memory, shared memory and global memory as shown in Fig 1.

Due to the limited space in GPU global memory, the GPU application state can be saved in the host system memory first. Variables in registers and local memory are copied into one buffer, variables in shared memory are grouped into another one. These two buffers are in page-locked host memory. Variables in global memory are saved back in a user declared buffer. All three buffers can be moved to other machines for computation migration or filed on hard disks for long-term fault tolerance purposes.

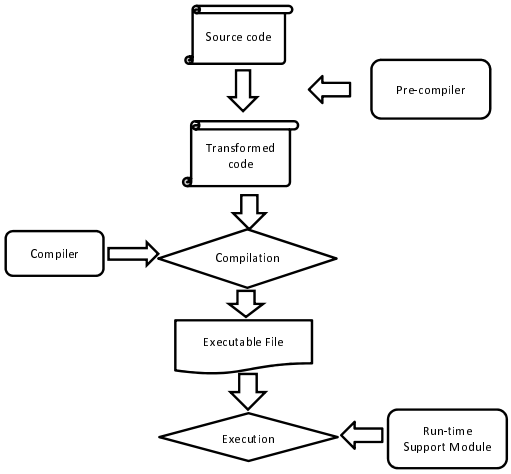

Fig. 2. Execution Process of the GPU Checkpoint/Restart Scheme

The checkpoint/restart of a GPU computation is accomplished through a precompiler at compile time with a run-time support module as shown in Fig. 2. Without the GPU driver's source code, collection/construction must be done at the application level. The precompiler transforms the application source code into a format where the run-time support calls are inserted for constructing the computation state precisely and efficiently. Developers only need to determine the checkpoint positions where the compiler directive "#pragma CPT" should be inserted.

The precompiler conducts the following tasks:

- Buffer allocation: create buffers to hold GPU state in host (CPU) system memory.
- Checkpointing insertion: detect directive "#pragma CPT" and insert the checkpoint statements accordingly [18].
- Threads synchronization: synchronize GPU threads before checkpointing.
- State variable collection: copy variables from GPU local, shared and global memory to pre-allocated buffers [30].
- Checkpoint counter management: maintain a checkpoint counter as a flag to distinguish checkpoint and restart at run-time.
- Library calls insertion: glue the run-time support module and user application together.
- File storage: copy the GPU state to secondary storage for long-term fault tolerance.

The run-time support module is activated through primitives inserted by the precompiler at compile time. It is required to link a run-time library with the user's applications in the final compilation. During the execution, it accomplishes state-related tasks:

- State registration: record the positions and sizes of data variables in local, shared and global memory as well as dynamically allocated memory blocks on the heap.
- State saving: construct and store the computation state consisting of application-level data segment, stack and heap.
- State Restoration: reload the state from buffers to overwrite data variables in different memory locations for resuming execution.

3.1. Pre-compilation

The pre-compilation work handles source code for both host (CPU) and device (GPU) sides for checkpointing and restarting, respectively.

3.1.1. Host-side Code Transformation

The user's original host program is shown in the upper portion of Fig. 3, whereas the pre-compiler modified program is the lower portion.

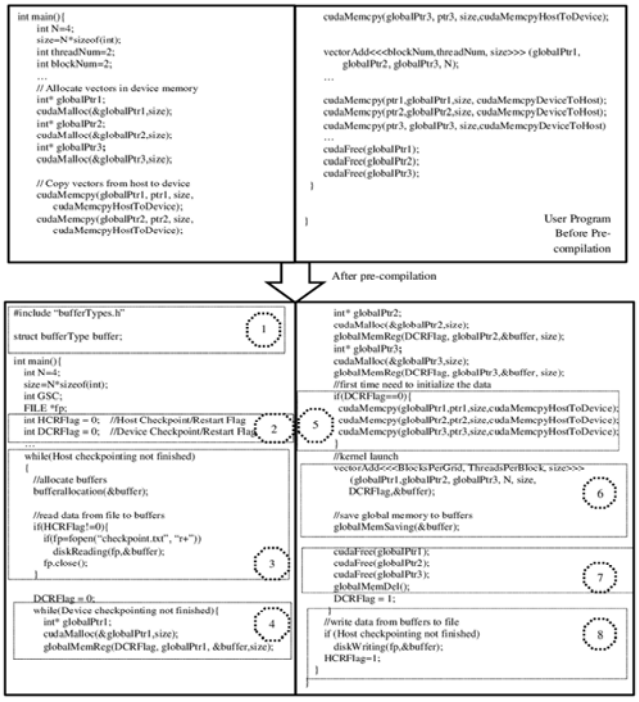

Fig. 3. An Example of Host Side Programs

In Area 1, the header file "*bufferTypes.h*" includes the structure *bufferType* and function prototypes used by the run-time support module. Structure *bufferType* contains the beginning addresses of all state buffers and tables in page-locked host memory. This structure will be used by the run-time support module for state saving and restoration.

In Area 2, flags *HCRFlag* and *DCRFlag* are initialized. These flags are used to indicate if the GPU state currently saved in the CPU memory needs to be written to or reloaded from secondary storage and if the kernel function needs to be launched multiple times (the GPU state will be saved in or reloaded from CPU memory).

In Area 3, the external while loop repeats the host side checkpoint/restart process until the application execution finishes. The data structure *buffer* is initialized and memory is allocated in page-locked host memory for state saving. If secondary storage is used, the computation state from the last checkpoint will be loaded into the buffers in page-locked host memory by calling *diskReading(fp, buffer)* at the beginning of every round if it is not the first time, which is indicated by *HCRFlag*.

In Area 4, the internal while loop repeats the device side checkpoint/restart process until execution is finished. Function *globalMemReg(DCRFlag, globalPtr1, &buffer, size)* saves the beginning address pointed to by *globalptr1*, as well as the size of the GPU global memory block in a linked-list specified in the buffer. Each global memory block is recorded one after another using this function call. If the *DCRFlag* indicates that the current round is not the first round, *globalMemReg* then the state is also restored from the state buffer, i.e., the content of the newly allocated global memory block will be overwritten by the state saved in the previous checkpoint.

In Area 5, if the *HCRFlag* indicates that the current round is the first round, the original data will be copied into GPU global memory.

In Area 6, three new parameters are required for the launched kernel. Parameter *size* represents the number of bytes in a dynamically specified shared memory block. Parameters *DCRFlag* and *buffer* are used by the runtime support module on the device side. After the kernel launch, function *globalMemSaving(&buffer)* saves the data from GPU global memory into the state buffers on the CPU side.

In Area 7, at the end of the inner while loop, function *globalMemDel()* is called to de-allocate the memory block in GPU global memory. In addition, the flag *DCRFlag* is set to 1.

In Area 8, *diskWriting(fp,&buffer)* writes the entire state saved in the buffers (including all variables in global, shared and local memory, and the registers) into the checkpointing file on secondary storage. In addition, the *HCRFlag* is updated to 1 before the external loop ends.

3.1.2. Device-side Code Transformation

The user's original device program is shown in the upper portion of Fig. 4, whereas the pre-compiler annotated program is shown in the lower portion.

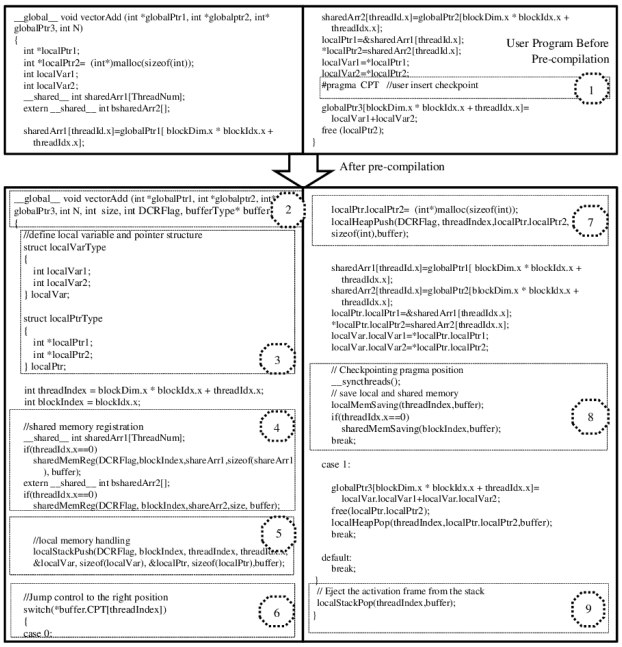

Fig. 4. An Example of Device Side Programs

In Area 1, the user can insert a directive "*#pragma CPT*" to instruct the pre-compiler to insert checkpointing statements.

In Area 2, three new parameters are added to the function parameter lists. Parameter *size* represents the size of dynamically allocated block in GPU shared memory. Parameter *DCRFlag* determines if the current kernel launch is a first-time execution or not. If not, the previous computation state should be restored first. Parameter *bufferType *buffer* passes the address of the buffer structure where the runtime support module can get clues to save all kinds of computation states in page-locked memory on the CPU side.

In Area 3, the pre-compiler collects all local variables in two newly defined data structures: *localVar* contains all non-pointer variables and *localPtr* is for pointers. The GPU *nvcc* compiler and driver will decide to place these variables' in registers or local memory. The reason for gathering pointers in one data structure is to update them easily when the computation is resumed.

In Area 4, shared memory is registered. Each array in shared memory is treated as a memory block. The function call *sharedMemReg(DCRFlag, blockIndex, shareArr1, sizeof(shareArr1), buffer)* registers the memory block into a list by passing the block index, address and size of the array. However, it is not necessary for every thread to register since the shared memory is shared by all blocks. Thread 0 of each block is responsible for the registration of shared memory. If the *DCRFlag* indicates that the current kernel launch is not a first-time execution, the saved data in the buffer is copied back to overwrite the newly allocated memory block.

Programmers can also dynamically allocate shared memory schemes by declaring shared memory variables as *extern __shared__ int bsharedArr2[].* Then the size of the array in the number of bytes is passed in as the third argument of the kernel configuration *Kernel<<< gridDim, blockDim, size >>>*. In order to register the size of the dynamically allocated memory block, the size is also a required parameter in the kernel function's parameter list as shown in Area 1.

In Area 5, the content of local memory, such as the stack and heap for each thread is handled. Function *localStackPush(DCRFlag, blockIndex, threadIndex, threadIdx.x, &localVar, sizeof(localVar), &localPtr, sizeof(localPtr), buffer)* is called to communicate with the run-time support module. Parameters *blockIndex*, *threadIndex* and *threadIdx.x* are passed in to identify the corresponding block and thread. Then the beginning address and size of the two aforementioned data structures *localVar* and *localPtr* will be recorded as they represent the stack for this particular thread. Therefore, each thread's stack is duplicated in the application level. All information can be traced out through a data structure pointed to by the parameter buffer based on which run-time support module can reconstruct the computation state dynamically. If *DCRFlag* indicates that the current kernel launch is not a first-time execution, function *localStackPush* also restores/loads the state from the buffer in page-locked host memory to overwrite the local stack frame, i.e., the contents of data structures *localVar* and *localPtr*.

In Area 6, a switch statement is inserted to dispatch an execution to each labeled point according

to the value of *CPT[threadIndex]*. Pragma counter *CPT* determines where execution should begin, in particular, when to resume a partially executed kernel function. Case 0 is for the first-time execution of a program. In addition, the *CPT* value for each thread needs to be updated after each partial execution.

In Area 7, users can call *malloc()* to allocate memory blocks in the heap (global memory). Each thread gets one small part of the big block. Usually, these are addressed by local pointers (by each thread). The memory block allocated by *malloc()* logically belongs to the thread that creates it. So these memory blocks are treated as if they "were" in local memory (actually in the heap of global memory). Each dynamically allocated memory block is registered to the runtime support module through the function call *localHeapPush(DCRFlag, threadIndex,localPtr.localPtr2, sizeof(int),buffer)* so that each thread's heap will be duplicated at the application level during the state construction period.

In Area 8, after the checkpointing directive in the original code, all the threads in the same blocks are synchronized. Then all memory blocks including stack and heap in local and shared memory are saved into buffers in page-locked host memory. Only thread 0 in each block is necessary to save the memory blocks in shared memory for that particular thread block.

In Area 9, before the kernel returns, function *localStackPop(threadIndex,buffer)* pops the current activation frame from its stack to finish the execution.

3.1.3. Header file bufferType.h

Header file bufferType.h includes a structure type bufferType and the prototype of the functions in runtime support module as shown in Fig. 5.

```
//bufferTypes.h

struct bufferType
{
  void* globalMemBuf;          //the beginning address of global memory buffer
  int sizeOfGMB;               //the size of global memory buffer
  void* sharedMemBuf;          //the beginning address of shared memory buffer
  int sizeOfSMF                //the size of shared memory buffer
  void* localStackBuf;         //the beginning address of local stack buffer
  int sizeOfLSB                //the size of local stack buffer
  void* localHeapBuf;          //the beginning address of local heap buffer
  int sizeOfLHB;               //the size of local heap buffer
  void* globalBufPtr;          //the pointer to record current global memory buffer postion
  void** sharedBufPtr;         //the pointer array to record current shared memory buffer postions
  void** localStackPtr;        //the pointer array to record current local stack buffer postions
  void** localHeapPtr;         //the pointer array to record current local heap buffer postions
  long** globalMemBlock;       //the pointer of global memory block table
  long*** sharedMemBlock;      //the pointer array of shared memory block tables
  long*** localMemBlock ;      //the pointer array of local memory block tables
  void* globalMemList;         //the pointer of global memory list head
  void** sharedMemList;        //the pointer array of shared memory list heads
  void** localMemStack;        //the pointer array of local memory stack heads
  void** localMemHeap;         //the pointer array of local memory heap heads
  int threadNum;               //the number of threads per block
  int blockNum;                //the number of blocks per grid
} ;

void bufferallocation(bufferType* buffer);
void globalMemReg(int DCRFlag,void* globalPtr,bufferType *buffer, int size);
void sharedMemReg(int DCRFlag,int blockIndex,void* shareArr,int size, bufferType *buffer);
void localStackPush(int DCRFlag,int blockIndex,int threadIndex,int threadIdx.x, void*  localVar, int
sizeOfVar, void* localPtr, int sizeOfPtr, bufferType *buffer);
void localHeapPush(int DCRFlag, int threadIndex,void* ptr, int size, bufferType *buffer);
void globalMemSaving(bufferType* buffer);
void sharedMemSaving(int blockIndex, bufferType* buffer);
void localMemSaving(int threadIndex,bufferType* buffer);
void diskReading(FILE *fp, bufferType *buffer);
void diskWriting(FILE *fp, bufferType *buffer);
void globalMemDel();
void localHeapPop(int threadIndex, void* ptr, BufferType* buffer);
```

Fig. 5. The content of the file *bufferTypes.h*

Structure type *bufferType* includes the beginning addresses of the buffers, the sizes of the buffers, the pointers to record the positions in the buffers, the beginning addresses for the tables, the heads of the data structure, the thread number per block and the block number per grid.

Buffers include *globalMemBuf*, *sharedMemBuf*, *localStackBuf* and *localHeapBuf*, whereas *sizeOfGMF*, *sizeOfSMF*, *sizeOfLSB* and *sizeOfLH*B are used to record their sizes, respectively. *SharedMemBuf, localStackBuf* and *localHeapBuf* are physically one-dimensional buffers but logically two-dimensional ones. By dividing the *blockNum* from *sizeOfGMF*, we can get the offset of the row for each block in the *SharedMemBuf*. By dividing the (*blockNum*threadNum*) from *sizeOfLSB* and *sizeOfLHB*, we can get the offset of the row for each thread in the *localStackBuf* and *localHeapBuf*.

After the state is saved into the buffer, it is necessary to record the current position in the buffer so that the state for the next memory block will not rewrite the current memory blocks' states. Pointer

globalBufPtr records the current position in *globalMemBuf*. However, since *sharedMemBuf*, *localStackBuf* and *localHeapBuf* are two-dimensional buffers, *sharedBufPtr*, *localStackPtr* and *localHeapPtr* are pointer type arrays where each element in the array corresponds to each block or thread.

Tables include *globalMemBlock*, *sharedMemBlock* and *localMemBlock*. They are used to keep track of the new address, old address and size of each memory block. Table *globalMemBlock* maintains the information of global memory blocks. It can be treated as a two-dimensional long data type array, and all the addresses will be transferred into long integers and inserted into the table. However, there are multiple tables for shared memory blocks and local memory blocks since each block and thread maintain its own table. Therefore, *sharedMemBlock* and *localMemBlock* are arrays of the pointers pointing to each table.

Data structures include *globalMemList*, *SharedMemList*, *localMemStack* and *localMemHeap*. Global memory only needs a single linked-list to record the memory blocks and *globalMemList* is the head of the list. However, every block in shared memory maintains its own list and every thread maintains its own stack and heap. Therefore, *SharedMemList*, *localMemStack* and *localMemHeap* are arrays of pointers pointing to the head of the list of each block, stack and heap of each thread.

Variable *threadNum* represents the number of thread per block and *blockNum* represents the number of block per grid.

The functions with corresponding prototypes are responsible to allocate the buffers, register, save and restore the state, and store the state into files. These are used in the run-time support module.

3.2. Run-time Support Module

The run-time support module is activated through the library calls inserted by the pre-compiler into the user code. Mainly, this module is responsible to register, save and restore the computation state. In addition, other support functions provide for buffer allocation, file management, and data structure deletion.

3.2.1. Buffer Allocation

Function *bufferallocation(bufferType *buffer)* allocates the memory for buffers and arrays. Buffer *globalMemBuf* is allocated within normal CPU memory with *sizeOfGMB*. Buffer *SharedMemBuf*, *localStackBuf* and *localHeapBuf* are allocated within page-locked host memory.

CUDA provides library calls to allocate and free page-locked (also known as pinned) host memory blocks through *cudaHostAlloc()* and *cudaFreeHost()*. Such memory blocks are opposed to regular pageable memory blocks allocated and de-allocated by system calls *malloc()* and *free()*. The advantages of page-locked host memory include:

- Copies between page-locked host memory and device memory can be performed concurrently with kernel execution for some devices [30].
- On some devices, page-locked host memory can be mapped into the address space of the device, eliminating the need to copy data back and forth between host and device memory.
- For systems with a front-side bus, bandwidth between the host memory and device memory is higher if the host memory is allocated as page-locked.

Variable *globalBufPtr* is a pointer in CPU memory and *globalMemList* is will be allocated dynamically later in the registration step. However, *sharedBufPtr*, *localStackPtr*, *localHeapPtr*, *globalMemBlock*, *sharedMemBlock*, *localMemBlock*, *SharedMemList*, *localMemStack* and *localMemHeap* are all allocated on page-locked host memory so that they can be accessed by the device side. Among them, table *globalMemBlock* is allocated with size of *(the memory block number * 3 * sizeof(long))*. Pointer arrays *sharedBufPtr*, *sharedMemBlock* and *SharedMemList* are allocated with the size of *(blockNum * sizeof(void*))* while *localStackPtr*, *localHeapPtr*, *localMemBlock*, *localMemStack* and *localMemHeap* are allocated with the size of *(threadNum * sizeof(void*))*.

3.2.2. State Registration

In order to keep track of the memory blocks in global, shared and local memory, these memory blocks are registered into special data structures: a linked list for global memory, a list for shared memory per block, a stack and a heap for local memory per thread. The beginning address and size of each memory block are registered into a single node and added to the data structure through the registration function calls inserted by the pre-compiler. These two data structures simulate real memory I/O on the GPU. In addition, the addresses and sizes of the memory blocks are recorded in tables for further pointer updates.

Global Memory Registration

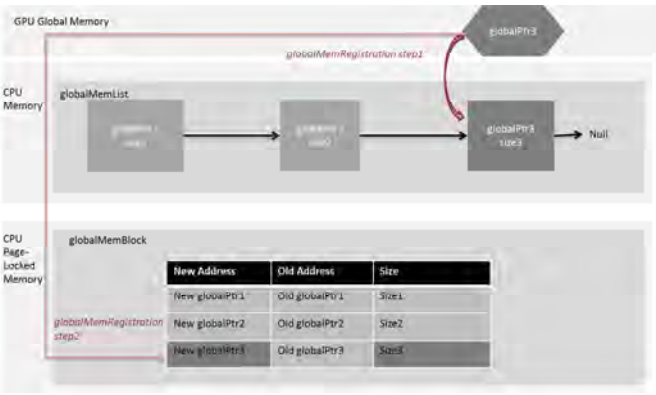

Fig. 6. Process Layout of Global Memory Registration

Function *globalMemReg(int DCRFlag,void* globalPtr,bufferType *buffer, int size)* accomplishes global memory block registration in two steps as shown in Fig. 6. The global memory block's beginning address and size are passed by parameter *globalPtr* and *size*. Parameter *bufferType *buffer* contains the address of the head of *globalMemList* and *globalMemBlock*. Parameter *DCRFlag* will be used to indicate restoration.

- In step 1, as the example shown in Fig. 6, a new node is added to the tail of the *globalMemList* to record the address and size of the memory block allocated at *globalPtr3*. Because the global memory blocks are allocated by the program executed on the host side, in order to save the space for global memory and page-locked host memory, the list *globalMemList* is in CPU memory. Pointer *buffer->globalMemList* is assigned to point to the head of this list.
- In step 2, the information about the memory block is also maintained in table *globalMemBlock*, which keeps track of the old and new addresses as well as the sizes of the memory blocks. When the function *globalMemReg()* is called, the address of the memory block *globalptr3* is written into the first column. The size is inserted into the third column. Since the run-time support module will search the table during restoration, it is maintained in the page-locked host memory for both sides' access.

Shared Memory Registration

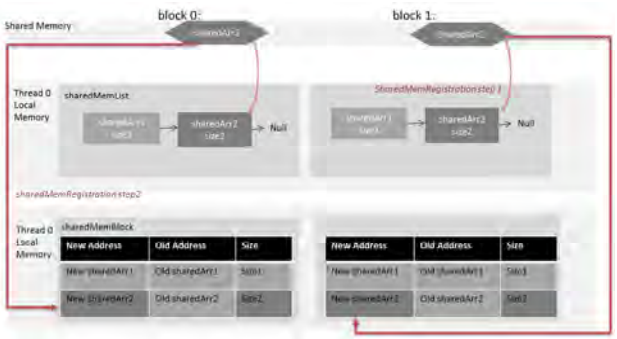

Fig. 7. Process Layout of Shared Memory Registration

Function *sharedMemReg(int DCRFlag,int blockIndex,void* shareArr,int size, bufferType *buffer)* helps register memory blocks in shared memory in two steps as shown in the example in Fig. 7. The beginning address of an array and its size are passed by parameters *sharedArr* and *size*. The block ID is specified by *blockindex* to distinguish each thread block.

- In step 1, a new node is added to the tail of the list *sharedMemList* to record the address and size of array *sharedArr2*. Because the shared memory is very critical for dynamically allocated memory

and all threads in the block share the same memory, the list is maintained in the local memory of thread 0 in each block.

- In step 2, the information about the memory block is also recorded in table *sharedMemBlock*, which performs the same functionality as *globalMemBlock*. The only difference is that each thread block has its own table in the local memory of thread 0 in each block.

In the main data structure *buffer*, two fields, *buffer->sharedMemList* and *buffer->sharedMemBlock*, are allocated on page-locked host memory as two pointer type arrays. Each element points to the beginning address of the *sharedMemList* and *sharedMemBlock* for each block by matching the array index with the block index. The *sharedMemList* and *sharedMemBlock* are allocated in local memory logically but allocated on the heap of global memory physically.

Local Memory Registration

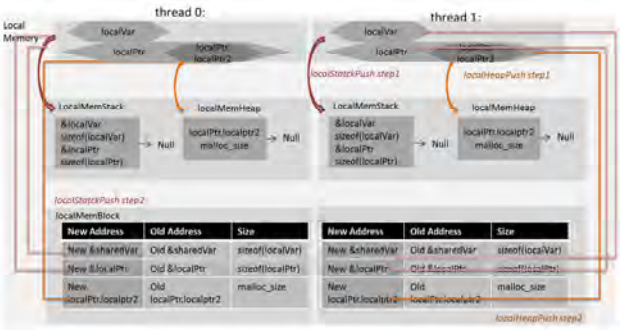

Fig. 8. Process Layout of Local Memory Registration

Local memory registration includes two operations: *localStackPush* and *localHeapPush*. Function *localStackPush(int DCRFlag,int blockIndex,int threadIndex,int threadIdx.x, void* localVar, int sizeOfVar, void* localPtr, int sizeOfPtr, bufferType *buffer)* pushes the static local memory blocks onto the stacks as shown in Fig. 8. All the variables and pointers are collected and gathered into structures *localVar* and *localPtr* at pre-compilation time. The addresses of these two structures and sizes are passed by the parameter list. Function *localHeapPush(int DCRFlag, int threadIndex,void* ptr, int size, bufferType *buffer)* pushes the dynamically allocated memory blocks in local memory onto the heaps. The beginning address and size of the memory block are passed by ptr and size. Moreover, thread index is necessary for both functions because each thread is registered separately. Both functions take two steps to complete the registration:

- In step 1, *localStackPush* gathers the addresses for structure *localVar*, *localPtr* and their sizes in a node then pushes them onto the stack, whereas *localHeapPush* pushes the address and size of the dynamic memory block pointed by the pointer field such as *localPtr.localPtr2* onto the heap. Each thread maintains its own stack and heap in its local memory.
- In step 2, the addresses and sizes for all the static and dynamic memory blocks are recorded into the same table *localMemBlock*. Each thread maintains its own table. All other details of the table are the same as table *globalMemBlock*.

In the common data structure *buffer*, three fields: *buffer->localMemStack*, *buffer->localMemHeap* and *buffer->localMemBlock* are allocated in page-locked host memory as three pointer type arrays. Each element points to the beginning address of the *localMemStack*, *localMemHeap* and *localMemBlock* for each thread, respectively, by matching the array index with thread index. The *localMemStack*, *localMemHeap* and *localMemBlock* for each thread are allocated in local memory logically but on the heap of global memory physically.

3.2.3. *State Saving*

Memory blocks in global memory will no longer be available once they are de-allocated. In addition, the memory blocks in local and shared memory are not accessible to the outside world after the kernel exits. In order to keep such state data in these memory blocks, the run-time support module helps save them into host memory.

Global Memory Saving

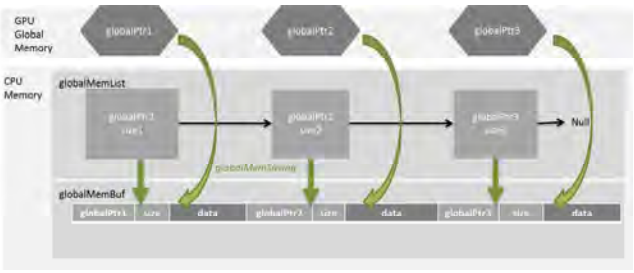

Fig. 9. Process Layout of Global Memory Saving

Function *globalMemSaving(bufferType* buffer)* saves the state in each memory block in global memory into buffer *globalMemBuf* in CPU memory as shown in Fig. 9. The beginning address of the buffer and the head of the list are contained in *bufferType *buffer*.

The run-time support module traverses the whole list, saves the address and size in the node to the buffer first, and then uses *cudaMemcpy()* API to copy the data inside the memory block into the buffer. CUDA provides this API to make the global memory accessible from the host, and the flag *cudaMemcpyDeviceToHost* allows memory copy from global memory to CPU memory.

Shared Memory Saving

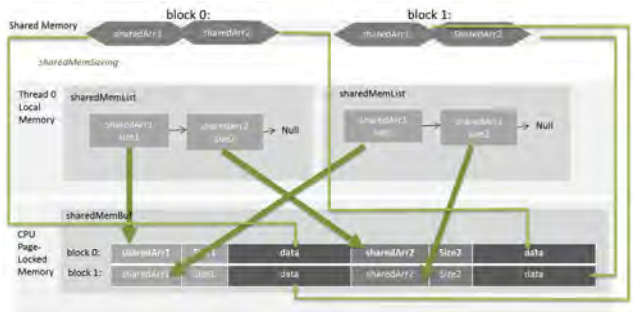

Fig. 10. Process Layout of Shared Memory Saving

Function *sharedMemSaving(int blockIndex, bufferType* buffer)* saves the state in each memory block in shared memory into a two dimensional buffer *sharedMemBuf* as shown in Fig. 10. The buffer is allocated in page-locked host memory so that both the CPU side and the GPU side can access it. Each row of the buffer is dedicated to back up the state of one thread block. Parameter *blockIndex* helps distinguish each block and *bufferType* buffer* contains the beginning address of *sharedMemBuf* and the heads of the lists.

Thread 0 in each block is responsible of saving the shared memory state for the whole block. It traverses the *sharedMemList* in local memory: first saves the address and the size of the memory block into corresponding row in the buffer, and then copies the data inside the memory block into the buffer through *memcpy()*.

Local Memory Saving

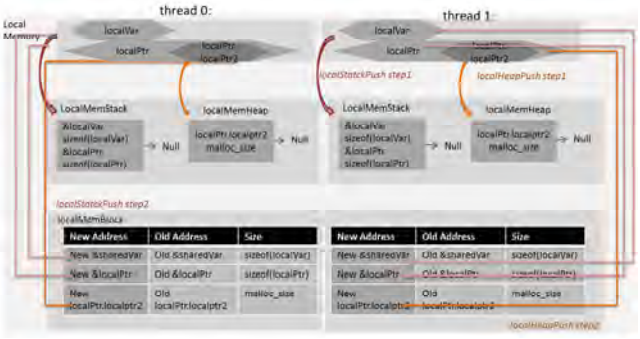

Fig. 11. Process Layout of Local Memory Saving

Function *localMemSaving(int threadIndex,bufferType* buffer)* saves the state in each thread in local memory into two two-dimensional buffers *localStackBuf* and *localHeapBuf* as shown in Fig. 11. Each row of the buffers is dedicated to back up the local memory state of a thread. In addition, both buffers are allocated in page-locked host memory for accessibility from both the CPU and the GPU sides. Parameter *threadIndex* corresponds to thread ID and *bufferType* buffer* contains the beginning address of buffer *localStackBuf* and *localHeapBuf*, as well as the heads for the stacks and heaps. The saving process includes stack saving and heap saving.

Each thread traverses the stack and saves every node into the corresponding buffer location. Because there are two pairs of address and size in each

node, they are saved in order. First, the address and size for *localVar* are assigned to buffer *localStackBuf*. Second, the data of *localVar* is copied through *memcpy()*. Third, the address and size of *localPtr* are saved to the buffer. Fourth, the data of *localPtr* is copied over. This indicates that each node in the stack needs six slots on the buffer for saving.

The heap is traversed by each thread: the address and size are saved first and then the data in the memory block is copied over by *memcpy()*.

3.2.4. State Restoration

During state restoration, states stored in the buffer are copied back to different GPU memory units.

Global Memory Restoration

Fig. 12. Process Layout of Global Memory Restoration

Global memory restoration is triggered by function *globalMemReg(int DCRFlag,void* globalPtr,bufferType *buffer, int size)*. If *DCRFlag* is detected to be 0, the restoration process will be skipped. Otherwise, the state will be restored. Global memory restoration takes three steps. Using the example shown in Fig. 12:

- In step 1, the address of *globalPtr1* in *globlMemBuf* represents the old address of the memory block during the last checkpointing. The address is recorded in the old address column in table *globalMemBlock*.
- In step 2, the data of old *globalPtr1* memory block is copied back to the newly allocated memory block using function *CudaMemcpy()* with flag *cudaMemcpyHostToDevice*. The beginning address of the new memory block can be accessed from the parameter *void* globalPtr* in function *globalMemReg()*.
- In step 3, the pointer *globalBufPtr* in the buffer is assigned to point to the beginning position of the next memory block so that the next memory block can be restored when *globalMemReg* is called again. The reason an extra pointer is required here is that the user cannot move the original pointer for *globalMemBuf*. The saving process of the next checkpoint will reuse the same buffer from the beginning.

Shared Memory Restoration

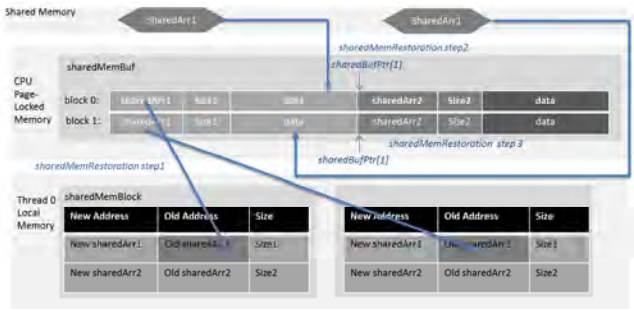

Fig. 13. Process Layout of Shared Memory Restoration

Shared memory restoration is invoked by *sharedMemReg (int DCRFlag, int blockIndex, void* shareArr, int size, void* buffer)* if *DCRLag* is detected to be 1. Thread 0 of each block is responsible for restoring the state from the buffer to the memory block in shared memory. The whole restoration process is also accomplished in 3 steps as shown in Fig. 13:

- In step 1, the address of *sharedArr1* is read from buffer *sharedMemBuf*, and added into table *sharedMemBlock* as the old address of *sharedArr1*. *SharedArr1* of each row corresponds to the old *sharedArr1* of each thread block.
- In step 2, the data in the buffer is copied into the new memory block (with the same size) using *memcpy()*. The beginning address of the new

memory block is passed by parameter *sharedArr* in the function *sharedMemReg()*.

- In step 3, *sharedBufPtr* is an array of pointers. Each element records the current location of each row in the buffer after one memory block is recovered so that the next memory block's restoration can start from the current position.

Local Memory Restoration

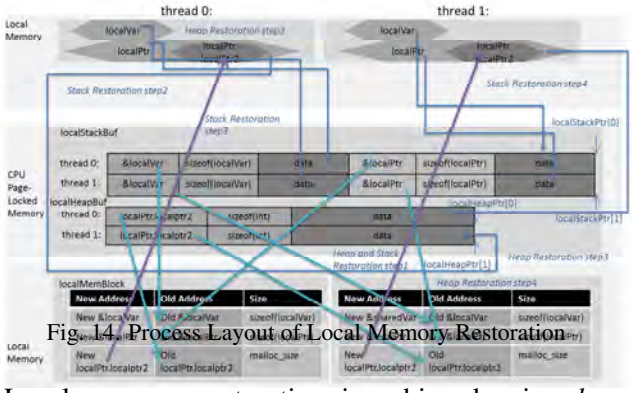

Fig. 14. Process Layout of Local Memory Restoration

Local memory restoration is achieved using *localStackPush(int DCRFlag, int blockIndex, int threadIndex, int threadId, void* localVar, int sizeOfVar, void* localPtr, int sizeOfPtr, bufferType* buffer)* when *DCRFlag* is detected to be 1. The states in *localStackBuf* and *localHeapBuf* are written back to the local memory by each thread. The local memory restoration process includes stack and heap restoration as shown in Fig. 14.

Heap restoration is completed in a loop containing four steps:

- In step 1, the address of *localPtr.localptr2* in *localHeapBuf* is written into the old address column in the table *localMemBlock*.
- In step 2, a same size memory block in local memory is reallocated using function *malloc()* and the address of the new memory block is recorded into the corresponding new address column in table *localMemBlock*.
- In step 3, the data in the old memory block is copied into the corresponding new memory block.
- In step 4, *localHeapPtr* is an array of pointers. Each pointer keeps track of the current position of the row of buffer *localHeapBuf*.
- Go to step 1 if the buffer has not been entirely restored.

Stack Restoration contains four steps:

- In step 1, the beginning address of *localVar* is written into the old address column in table *localMemBlock*.
- In step 2, the data of the old memory block is copied into the new memory block whose beginning address is passed by parameter *localVar* in function *localStackPush()*.
- In step 3, the pointer structure is restored. This step has to wait until all other memory blocks have been restored and all memory block tables are updated. This is because the pointers in the pointer structure are required to point to the new memory blocks that contain the data of the corresponding old memory blocks. It takes four sub-steps to finish:
  - The address of *localPtr* is written into the old address column in table *localMemBlock*.
  - For each pointer from the *localPtr* structure, compare the pointer with old beginning addresses in the table *globalMemBlock*, *sharedMemBlock* and *localMemBlock* to determine which memory block it is located in.
  - Get the offset by subtracting the beginning address of the old memory block from the pointer.
  - Update the pointer's new pointing address by adding the offset to the corresponding new beginning address of the block, i.e., (original pointer address - old memory block address) + new memory block address.
- In step 4, *localStackPtr* is an array of pointers. Each pointer keeps track of the current position of the row in buffer *localStackBuf*.

3.2.5. File Management

Function *diskWriting(FILE *fp, bufferType *buffer)* writes all buffers into a checkpointing file. First, the size of *bufferType* is written into the file. Second, the buffer itself is written into the file. Third, global memory, shared memory, local stack, and local heap buffers are written into the file sequentially.

Function *diskReading(FILE *fp, bufferType *buffer)* reads the states from the checkpointing file and copies them into buffers. First, the size of *bufferType* is read so that the buffer data structure itself can be read. Second, according to the *sizeOfGMB* in old buffer, the state for global memory is copied into the new allocated buffer passed by parameter *buffer*. Shared memory and local memory buffers are copied back one after another using the same strategy.

3.2.6. Data Structure Deletion

Function *globalMemDel()* deletes the linked list *globalMemList* in CPU memory, whereas function *localHeapPop(int threadIndex, void* ptr, BufferType* buffer)* removes the tail of heap *localMemHeap* in each thread's local memory to return the resources back to the system.

4. Experimental Results

All GPU checkpoint/restart experiments are conducted based on NVIDIA Fermi GPU, Tesla C2050. The vector addition application was selected as the test program. The array sizes used were 256, 512, 1024, 2048, 4096, 8192, 16384 and 32768. The state buffers were allocated in page-locked host memory on the CPU side in four modes: default, portable, write-combining and mapped memory.

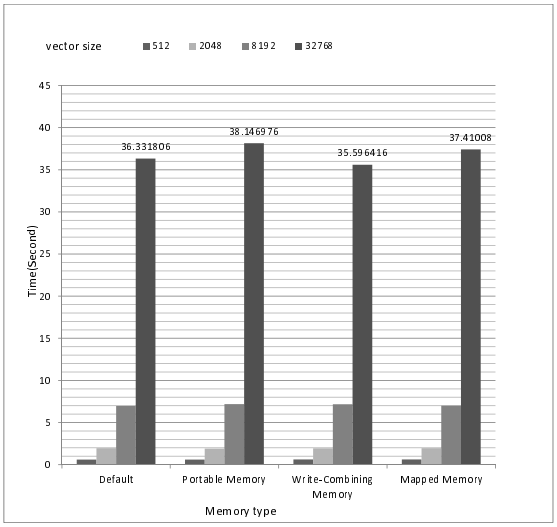

Fig. 15. GPU Checkpointing Overhead

Checkpointing overhead for the vector addition application is shown in Fig. 15. The gap in checkpointing overheads for the four modes grows as the array size increases. Write-combining memory exhibits the best performance. This is because write-combining memory frees up the host's L1 and L2 cache resources, making more cache available to the rest of the application. In addition, write-combining memory does not require snooping the PCI Express bus for data transfers. Compared with default page-locked host memory, write-combining increased the performance of vector addition by 2% for checkpointing on average.

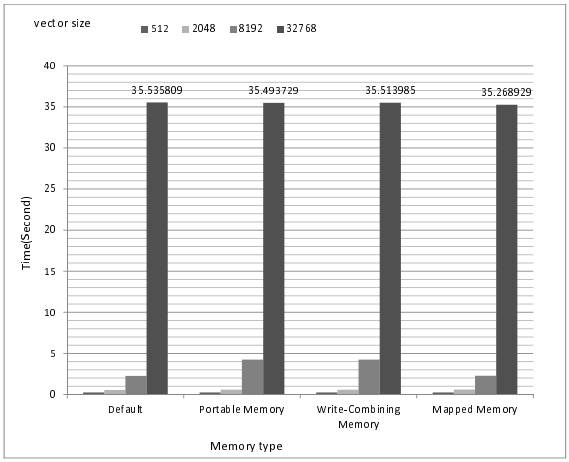

Fig. 16. GPU State Restoration Time

The overheads in restart phase is shown in Fig. 16. Mapped memory exhibits the best performance. Accessing host memory from a kernel directly has several advantages. First, there is no need to allocate a block in device memory, and then copy data back and forth between host and device memory. Second, data transfers are implicitly performed as needed by the kernel. Third, the kernel-originated data transfers automatically overlap with kernel execution for performance gains. Compared with default page-locked host memory, mapped memory increased the restart performance by 1.3% on average.

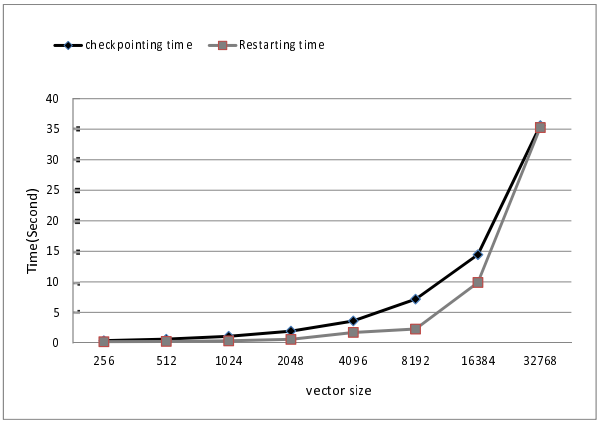

Fig. 17. Checkpoint/Restart Overhead with the Best Mode Combination

Fig. 17 shows the overall performance of the best page-locked memory configuration for different array sizes. Write-combining and mapped memory modes are used for checkpoint and restart, respectively. Obviously, the application-level GPU checkpoint/restart might introduce high overheads for applications with big data sets. However, it is still a feasible mechanism to provide fault tolerance features for long-running scientific applications.

5. Related Work

State-Carrying Code (SCC) is a software mechanism to achieve computation mobility by saving and retrieving computation states during normal program execution in heterogeneous multi-core/many-core clusters[27]. SCC adopts the application-level thread migration approach. However, SCC was only applied to applications running on the CPU and does not address CUDA programming at all.

Condor[31] has been developed by the University of Wisconsin-Madsion, and is a heterogeneous distribution system that uses idle resources to achieve high throughput in the work pool. Users only need to re-link with the library provided by Condor to use the Condor checkpointing and process migration feature. There is no need to modify the underlying Linux operating system[32]. However, Condor checkpointing technology only supports single-process tasks.

Libckpt[28] inserts a middle layer to initialize the checkpointing process and collect the state information at run-time in order to achieve checkpointing at the user level. When the state needs to be saved, the process' address space information and other information collected by the middle layer are stored to reliable storage. When a state is restored, a new process is started using the original binary file of the application.

One simple GPU checkpoint/restart scheme is proposed by Calhoun *et al.* [18] The purpose is to support preemptive kernel function for flexible GPU scheduling. However, the existing scheme works only with simple CUDA programs and cannot manage a complex GPU memory hierarchy.

CheCUDA is a tool to store the image of host memory into a file [25]. As an additional package of Berkeley Lab Checkpoint/Restart (BLCR) [19], it can handle CUDA applications. However, CheCUDA did not perform well in task migration. When BLCR fails to restart the process, CheCUDA is not able to migrate the task. In order to reduce the overhead of CheCUDA, Supada Laosookasathit proposed a lightweight checkpoint/restart using CUDA streams based on Virtual Cluster Checkpointing Protocol (VCCP) [29]. However, neither of these two checkpoint/restart schemes have dealt with complex GPU computation states directly.

6. Conclusions and Future Work

Checkpoint/restart is an effective scheme for fault tolerance and has been widely used to reduce the overall execution time of long-running applications when failures occur [26]. This paper proposes a new checkpoint/restart scheme for GPU applications. The pre-compiler transforms the user's original program, inserts directives for checkpoint/restart positions, and defines data structures to enable the runtime support system to collect computation states precisely and dynamically. For scalability, computation states are saved in page-locked host memory as well as on the secondary storage. Experimental results have demonstrated the effectiveness of the proposed scheme which can also be used for GPU computation migration between multiple GPU machines.

The future work includes reducing overheads

further and testing the scheme with more large-scale and complex applications.

References

1. J. Hursey, I. T. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in open mpi," *Proc. ACM Intl. Symposium on High Performance Distributed Computing* (2009).
2. I. P. Egwutuoha, S. Chen, D. Levy, and B. Selic, "A fault tolerance framework for high performance computing in cloud," *Proc. IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing* (2012).
3. Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 213–223 (2005).
4. S. Matsuoka, T. Aoki, T. Endo, A. Nukada, T. Kato, and A. Hasegawa, "Gpu accelerated computing-from hype to mainstream, the rebirth of vector computing," *Journal of Physics: Conference Series*, **180(1)** (2009).
5. A. Nukada, H. Takizawa, and S. Matsuoka, "Nvcr: A transparent checkpoint-restart library for nvidia cuda," *Proc. IEEE Intl. Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011).
6. K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," *Proc. ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* (2012).
7. D. Ibtesham, D. Arnold, K. B. Ferreira, and R. Brightwell, "Abstract: comparing gpu and increment-based checkpointing compression," *Proc. ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* (2012).
8. D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*, 7th, Addison-Wesley (2009).
9. A. Sherrod and W. Jones, *Beginning DirectX 11 Game Programming*, 1st, Course Technology Press (2011).
10. S. Cook, *UDA Programming: A Developer's Guide to Parallel Computing with GPUs* , 1st, Morgan Kaufmann (2012).
11. J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley (2010).
12. D. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*, Morgan Kaufmann (2012).
13. D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *Journal of ACM Computing Surveys*, **32(3)**, 241–299 (2000).
14. O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," *Proc. USENIX Annual Tech. Conf.* (2007).
15. M. Bozyigit, K. A1-Tawil, and S. Naseer, "A kernel integrated task migration infrastructure for clusters of workstations," *Computers and Electrical Engineering*, **26**, 279–295 (2000).
16. R. Gioiosa, J. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," *Proc. ACM/IEEE Intl. Conf. on Supercomputing* (2005).
17. M. Bozyigit and M. Wasiq, "User-level process checkpoint and restore for migration," *ACM SIGOPS Operating Systems Review*, **35(2)**, 86–96 (2001).
18. J. Calhoun and H. Jiang, "Preemption of a cuda kernel function," *Proc. ACIS Intl. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing* (2012).
19. P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Proc. SciDAC* (2006).
20. R. Xue, W. Chen, and W. Zheng, "CprFS: a user-level file system to support consistent file states for checkpoint and restart," *Proc. ACM Intl. Conf. on Supercomputing* (2008).
21. R. Arora, P. Bangalore, and M. Mernik, "A technique for non-invasive application-level checkpointing," *The Journal of Supercomputing*, **57(3)**, 86–96 (2011).
22. J. P. Walters and V. Chaudhary, "Application-Level checkpointing techniques for parallel programs," *Proc. Intl. Conf. on Distributed Computing and Internet Technology* (2006).
23. G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," *Proc. ACM SIGPLAN symposium on Principles and practice of parallel programming* (2008).
24. S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-C. Feng, "Transparent Accelerator Migration in a Virtualized GPU Environment," *Proc. IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing* , 124–131 (2012).
25. T. Hiroyuki, S. Katsuto, K. Kazuhiko, and K. Hiroaki, "Checuda: A checkpoint/restart tool for cuda applications," *Proc. Intl. Conf. on Control, Automation and Systems*, **10.1109**, 408–413 (2012).
26. Y. Ji, H. Jiang, and V. Chaudhary, "A heuristic checkpoint placement algorithm for adaptive application-level checkpointing," *Intl. Journal of Applied Science and Technology*, **1** (2011).
27. H. Jiang and Y. Ji, "State-carrying code for computation mobility," *Handbook of Research on Scalable Computing Technologies*, **38**, IGI Global (2009).

28. K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "libhashckpt: hash-based incremental checkpointing using gpuÕs," *Recent Advances in the Message Passing Interface*, **6960**, 272–281 (2011).
29. S. Laosookasathit, N. Naksinehaboon, C. Leangsukan, A. Dhungana, C. Chandler, K. Chanchio, and A. Farbin, "Lightweight checkpoint mechanism and modeling in gpgpu environment," *Proc. Intl. Workshop on System-level Virtualization for High Performance Computing* (2010).
30. NVIDIA. Nvidia fermi tuning guide (2010).
31. T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: a distributed job scheduler," *Beowulf Cluster Computing with Linux* (2001).
32. A. R. Butt, R. Zhang, and Y. Hu, "A self-organizing flock of Condors," *Journal of Parallel and Distributed Computing*, **66(1)**, 145–161 (2001).