A portable and adaptable fault tolerance solution for heterogeneous applications

Nuria Losada, Basilio B. Fraguela, Patricia González, María J. Martín

{*nuria.losada, basilio.fraguela, patricia.gonzalez, mariam*}*@udc.es*
Grupo de Arquitectura de Computadores, Universidade da Coruña, Spain
(+34) 881 011212; (+34) 981 167160 (fax). `Corresponding author: Nuria Losada.`

Abstract

Heterogeneous systems have increased their popularity in recent years due to the high performance and reduced energy consumption capabilities provided by using devices such as GPUs or Xeon Phi accelerators. This paper proposes a checkpoint-based fault tolerance solution for heterogeneous applications, allowing them to survive fail-stop failures in the host CPU or in any of the accelerators used. Besides, applications can be restarted changing the host CPU and/or the accelerator device architecture, and adapting the computation to the number of devices available during recovery. The proposed solution is built combining CPPC (ComPiler for Portable Checkpointing), an application-level checkpointing tool, and HPL (Heterogeneous Programming Library), a library that facilitates the development of OpenCL-based applications. Experimental results show the low overhead introduced by the proposal and prove its portability and adaptability benefits.

Keywords:
checkpointing, fault tolerance, heterogeneous systems, OpenCL, portability

1. Introduction

The usage of heterogeneous devices has become increasingly popular, as it provides important improvements in runtime and power consumption with respect to approaches solely based on general-purpose CPUs [1]. The November 2016 TOP500 list (`http://www.top500.org/`) clearly shows this tendency, with 86 systems with accelerators in comparison with the 64 on June 2014, which means a 34% increase in the last couple of years. Heterogeneous applications are those capable of exploiting more than one type of computing system, gaining performance not only by using CPU cores but also by incorporating specialized accelerator devices such as GPUs or Xeon Phis. From a fault tolerance perspective, these applications can suffer failures both in the main processor (host) or in the accelerators.

Di Martino et al. [2] have studied the Cray heterogeneous supercomputer Blue Waters, reporting that 1.53% of applications fail due to system problems. Moreover, applications using heterogeneous nodes displayed a higher percentage of failures due to system errors, which also grew larger when scaling. Overall, failed applications noticeably run for about 9% of the total production node hours. These results bring to light the necessity of fault tolerance mechanisms in High Performance Computing (HPC) heterogeneous applications to ensure the completion of their execution.

Checkpointing is a widely used fault tolerance technique in which the computation state is periodically saved to stable storage, allowing the recovery of the application when a failure occurs. This work proposes an adaptable checkpoint-based solution for heterogeneous applications by combining CPPC (ComPiler for Portable Checkpointing) [3, 4], a portable and transpar-

ent checkpointing infrastructure for sequential and MPI parallel applications, and HPL (Heterogeneous Programming Library) [5], a C++ library for programming heterogeneous systems on top of OpenCL [6], a widely adopted standard for heterogeneous computing. The proposal tolerates fail-stop failures in the host and/or in the accelerator devices used by the application. The main contributions of this work are:

- An application-level checkpointing solution for heterogeneous applications that applies a host-side approach (checkpoints are performed by the host between kernels invocations) and allows recovering from both host and devices failures.
- A low-overhead checkpointing protocol which minimizes host-device data transfers during the checkpoint operation.
- A highly portable restart process that allows applications to be recovered on different machines. The recovery is completely independent of the architecture of the host and the devices, the operating system and any higher level frameworks used.
- A highly adaptable restart process that allows the recovery of the applications adapting the computation to the available number of devices.

To the best of our knowledge no other work provides such portability and adaptability features to the recovery process of heterogeneous applications. In heterogeneous supercomputers, these features enable the completion of the applications even when those resources that were being used are no longer available. Also, the approach can improve the exploitation of the resources,

allowing applications to be started in those that are available and to later continue their execution using a more appropriate or powerful set of resources.

This paper is structured as follows. Section 2 comments upon the main characteristics of heterogeneous computing and HPL main characteristics. Section 3 introduces CPPC. The proposed portable and adaptable fault tolerance solution for heterogeneous applications is described in Section 4. The experimental results are presented in Section 5. Section 6 covers the related work. Finally, Section 7 concludes this paper.

2. Heterogeneous computing

Among the large number of frameworks for the development of applications that exploit heterogeneous devices, OpenCL is the most widely supported and, thus, the one that provides the largest portability across different device families.

2.1. OpenCL

Figure 1 depicts the general hardware model for heterogeneous computing provided by OpenCL, further described using OpenCL terminology. The sequential portions of the application run on the host and they can only access its memory. The parallel tasks are called kernels and they are expressed as functions that run in the attached devices at the request of the host program. Each device has one or more processing elements (PEs). All of the PEs in the same device run the same code and can only operate on data found within the memory of the associated device. PEs in different devices, however, can execute different pieces of code. Thus, both data and task parallelism are supported. Also, in some devices the PEs are organized in groups, called

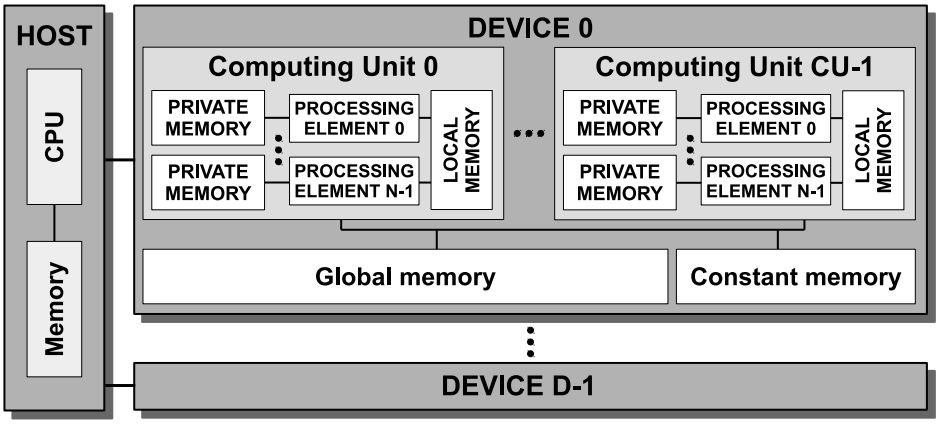

Figure 1: OpenCL hardware model. It is comprised of a host with a standard CPU and memory, to which a number of computing devices are attached.

computing units in Figure 1, which may share a small and fast scratchpad memory, called local memory.

Regarding the memory model, the devices have four kinds of memory. First, the global memory is the largest one and can be both read and written by the host or by any PE in the device. Second, a device may have a constant memory, which can be set up by the host and it is read-only memory for its PEs. Third, there is a local memory restricted to a single group of PEs. Finally, each PE in an accelerator has private memory that neither the other PEs nor the host can access.

Since the device and host memories are separated, the inputs and outputs of a kernel are specified by means of some of its arguments. The host program is responsible for the memory allocations of these arguments in the memory of the device. As well, the host program must transfer the data between the host and the device memory for the input arguments, and vice versa for the output arguments.

2.2. HPL

The Heterogeneous Programming Library (HPL) [5], available under the GNU General Public License (GPL) at `http://hpl.des.udc.es`, supports the previously described model on top of the OpenCL standard. The library provides three main components to users:

- A template class `Array` that allows the definition of the variables that need to be communicated between the host and the devices.
- An application programming interface (API) that allows inspecting the available devices and requesting the execution of kernels.
- An API to express the kernels. Kernels can be written using a language embedded in C++, which allows HPL to capture the computations requested and build a binary for them that can run in the chosen device. Another possibility is to use HPL as an OpenCL wrapper [7], enabling the use of kernels written in native OpenCL C in a string, just as regular OpenCL programs do, and thus, easing code reuse.

The data type `Array`<*type*, *ndim*[, *memoryFlag*]> represents an *ndim*-dimensional array of elements of the C++ type *type*, or a scalar for *ndim*=0. The optional *memoryFlag* specifies one of the kinds of memory supported (Global, Local, Constant, and Private). By default, the memory is global for variables declared in the host and private for those defined inside the kernels. Variables that need to be communicated between the host and the devices are declared as Constant or Global `Arrays` in the host, while those local to the kernels can be declared Private inside the kernels or Local both in the host and inside the kernels.

```
void kernel_1(Array<int, 1, Global> a1,/*INPUT*/
   Array<int, 1, Global> a2,/*INPUT*/
   Array<float, 1, Global> tmp,/*OUTPUT*/
   Array<float, 1, Constant> b,/*INPUT*/
   Array<int, 0, Global> i);/*INPUT*/

void kernel_2(Array<int, 1, Global> a2,/*OUTPUT*/
   Array<float, 1, Global> tmp,/*INPUT*/
   Array<int, 1, Local> c,
   Array<float, 1, Global> a3);/*INPUT&OUTPUT*/

Array<int, 1, Global> a1(N), a2(N);
Array<float, 1, Global> a3(N), tmp(N);
Array<float, 1, Constant> b(M);
Array<int, 1, Local> c(M);

int main()
{
   [...]
   /* kernel_1 and kernel_2 are associated to their
   OpenCL C kernels using the HPL API (not shown) */
   [...]

   for(i = 0; i < nIters; i++){
       eval(kernel_1)( a1, a2, tmp, b, i);
       eval(kernel_2)( a2, tmp, c, a3);
   }

   [...]
}
```

Figure 2: Example of an HPL application where two different kernels are invoked `nIters` times.

Figure 2 shows an example of an HPL application. The host code invokes the kernels with the HPL function `eval()`, specifying with arguments the kernels inputs and outputs. These arguments can be `Arrays` in global, constant or local memory, as well as scalars. Global and constant `Arrays` are initially stored only in the host memory. When they are used as kernel arguments, the HPL runtime transparently builds a buffer for each of them in the required device if that buffer does not yet exist. Additionally, the library automatically performs the appropriate data transfers to ensure that all the kernels inputs are in the devices memory before their execution. Local `Arrays` can be also used as kernels arguments. In this case, the appropriate buffers will be allocated in the devices, however, no data transfers

will be performed, as Local `Arrays` are invalidated between kernel runs. As for the output arrays, necessarily in global memory, they are only copied to the host when needed, for instance, when the `Array data()` method is used. This method returns a raw pointer to the array data in the host. When this method is called, HPL ensures that the data in the host is consistent by checking if any device has modified the array, and only in that case HPL transfers the newest version of the data from the device to the host. In fact, HPL always applies a lazy copying policy to the kernels arguments that ensures that transfers are only performed when they are actually needed.

3. CPPC overview

CPPC [4] is an open-source checkpointing tool available under the GPL at `http://cppc.des.udc.es`. It is implemented at the application level, and, thus, it is independent of the operating system and any higher-level framework used.

CPPC appears to the final user as a compiler tool and a runtime library. The CPPC compiler instruments the user-provided source files, adding calls to the CPPC library. This is illustrated in Figure 3, where the code inserted by the compiler is shown in boldface. The CPPC instrumentation performs the following actions:

- **Configuration and initialization:** at the beginning of the application the routines `CPPC_Init_configuration()` and `CPPC_Init_state()` configure and initialize the necessary data structures for the library management.

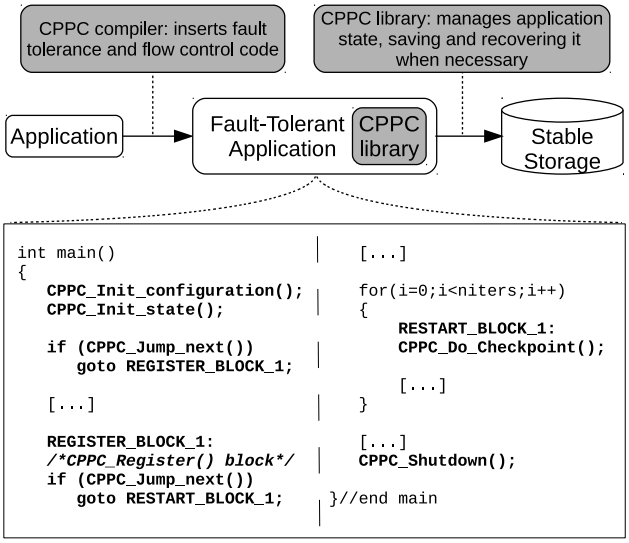

Figure 3: At compile time the CPPC source-to-source compiler automatically transforms a code into an equivalent fault-tolerant version by adding the instrumentation code (marked in boldface in the figure).

- **Registration of variables:** the routine `CPPC_Register()` explicitly marks the variables necessary for the successful recovery of the application, for their inclusion in checkpoint files. During restart, this routine also recovers the values from the checkpoint files to their proper memory location.

- **Checkpoint:** the `CPPC_Do_checkpoint()` routine is placed inside the most computationally expensive loops. A checkpoint file will be generated every N calls to this function, N being user-defined.

- **Shutdown:** the `CPPC_Shutdown()` routine is added at the end of the application to ensure the consistent system shutdown.

Checkpoint file sizes are reduced by storing user variables and by using a

liveness analysis to save only those variables that are essential for the application recovery. Besides, CPPC applies the zero-blocks exclusion technique, which avoids the storage of memory blocks that contain only zeros. Moreover, CPPC provides multithreaded dumping, overlapping the checkpoint file writing with the computation of the application, and thus, reducing the checkpointing overhead.

The state files generated by CPPC are portable, allowing the execution to restart on different architectures and/or operating systems. Portability is achieved by using HDF5, a portable storage format available at `http://www.hdfgroup.org/HDF5/`, and by avoiding the inclusion in the checkpoint files of non-portable state. This state is recovered through the re-execution of the code responsible for creating such state in the original execution. The compiler automatically identifies both the variables to be checkpointed and the non-portable code to be re-executed upon restart. The application stack is an example of such non-portable state, and it is recovered by recreating the sequence of original function calls that built the stack up to the checkpoint location. Additionally, CPPC incorporates a portability layer to deal with different file systems, being able to track open files at runtime, so that, upon restart, they can be re-opened and re-positioned in the correct part of the file.

The restart phase has two fundamental parts: reading the checkpoint data into memory and recovering the application state. The first step is encapsulated inside the routine `CPPC_Init_state()`. However, the actual reconstruction of the application state is achieved through the ordered re-execution of certain blocks of code: the configuration and initialization block,

variable registration blocks, checkpoint blocks, and non-portable state recovery blocks. The compiler inserts control flow code (labels and conditional jumps) to ensure an ordered re-execution. When the execution flow reaches the `CPPC_Do_checkpoint()` call where the checkpoint file was generated, the recovery process ends and the execution can resume normally.

For more details about CPPC and its restart protocol the reader is referred to [3, 4].

4. Portable and adaptable checkpoint/restart of heterogeneous applications

This section describes our proposal to obtain adaptable fault tolerant HPC heterogeneous applications using CPPC and HPL. First, the design decisions are described and justified, to latter present the implementations details of the proposed solution.

Note that, although the adaptability of an application when its execution starts is a feature of HPL, its adaptability in restarts is exclusively enabled by the design and implementation decisions presented in this Section. The contributions to support adaptability can be summarized in three main points. First, applications are analyzed to determine how they should be modified to be restarted in different device architectures and different numbers of devices. Second, CPPC is extended to make the required changes in the source codes. Third, as the restarted code must be executed in a different way depending whether the application is pseudo-malleable or malleable, an algorithm to detect the kind of adaptability of the application is designed and implemented in the CPPC compiler.

4.1. Design decisions

The first design decision is to determine the optimal location for the checkpoints, i.e. in which points of the application code its state should be saved to stable storage. This proposal is focused on long-running HPC heterogeneous applications to deal with fail-stop failures both in the host CPU and in the accelerator devices. The choice of a host-side checkpointing (placing checkpoints in the host code between kernels invocations) provides several performance, portability and adaptability benefits, further commented in the remain of this section. Moreover, a host-side approach guarantees in most practical situations an adequate checkpointing frequency because the execution times of kernels are not expected to exceed the Mean Time To Failure (MTTF) of the underlying system. This expectation is based both on physical limitations and on practical observations. Regarding the limitations, a very relevant one is that, as can be seen in Table 4, which describes the hardware used in our evaluation, accelerators usually have memories that are considerably smaller than those of regular multi-core servers. As a result, when huge computational loads are executed in them, it is quite common to have to alternate stages of transfers to/from the host memory with stages of kernel computation to be able to process all the data. As for the practical observations, in order to analyze the typical kernel runtimes, we performed an experimental evaluation of the most popular benchmark suites developed with OpenCL: Rodinia [8], SHOC [9], and SNU-NPB [10]. Table 1 shows the results for the four most time-consuming applications in each benchmark suite. The experiments took place in System#1, described in Table 4 of Section 5. Rodinia benchmarks were executed using the default parameters,

Table 1: Maximum kernel times (seconds) of the most time-consuming applications from popular benchmarks suites for heterogeneous computing.

		KERNELS	
		NUMBER	MAX. TIME (SECONDS)
RODINIA BENCHMARKS DEFAULT PARAMETERS	cfd	14004	0.00032
	streamcluster	4833	0.00069
	particlefilter	36	0.91775
	hybridsort	21	0.10381
SHOC BENCHMARKS SIZE 4	SCAN	15360	0.00138
	Spmv	10000	0.00320
	MD	162	0.00901
	Stencil2D	22044	0.00318
SNUNPB BENCHMARKS CLASS B	CG	8076	4.10887
	FT	111	0.51884
	SP	5637	0.05743
	BT	3842	0.10504

while in SHOC the largest problem available (size 4) was used. Finally, in SUN-NPB the configuration used was class B, as it was the largest problem that fits in the device memory. The studied applications execute a large number of kernels whose maximum times range from 0.32 milliseconds in application cfd to 4.1 seconds in CG, thus making host-side checkpointing a very appropriate alternative.

The next step consists in studying which application data should be included in the checkpoint files. The state of a heterogeneous application can be split into three parts: the host private state, the devices private state (data

in the local and private memory of the accelerators), and the state shared among the host and the devices (data in the global and constant memory of the accelerators, which may or may not also be in the host memory). By locating checkpoints in the host code, only the host private state and the shared state need to be included in the checkpoint files. The fact that neither private nor local memory data of the devices need to be checkpointed improves the performance of the proposal because smaller checkpoint file sizes are obtained, a key factor for reducing the checkpoint overhead.

Also, the solution must guarantee the consistency of the checkpointed data. During the execution of a heterogeneous application, computations performed in the host and in the devices can overlap, as the host can launch several kernels for their execution in the devices and continue with its own computation. Thus, a consistency protocol is needed to ensure a successful restart upon failure. The protocol must include synchronizations so that the kernels that may modify the data included in the checkpoint files are finished before the checkpointing. In addition, those checkpointed shared variables modified by the kernels must be transferred back to the host memory.

Finally, further design decisions aim to improve the portability and adaptability of the proposal in order to obtain a solution completely independent of the machine, that can be employed to restart the applications using different hosts and/or devices. The benefits of the migration to a different device architecture may include performance, however, its main advantage is the fact that it enables the execution to be completed. In heterogeneous supercomputers, this feature is useful in those situations in which the original resources are no longer available or, for instance, when the waiting time to

access them is prohibitive.

As checkpointing back-end, the CPPC tool was chosen because of the portable application-level approach it provides. The checkpoint files generated by CPPC allow the restart on different host, while their size is reduced by checkpointing only the user-variables necessary for the recovery of the application, thus, reducing the checkpointing overhead.

When placing the checkpoints in the host code, if a non-vendor specific heterogeneous framework is used, the application can be potentially recovered using a different device architecture. Moreover, if the distribution of data and computation performed by the application is not tied to the number of devices available at runtime, applications could be restarted adapting to a different number of devices. However, decoupling the distribution of data and computation from the number of devices available at runtime can be a hard task in some applications. The high programmability of HPL simplifies the implementation of programs in which the application data and computation is not tied to the available devices at runtime. This, together with the fact that HPL is not tied to any specific vendor, operating system or hardware platform, facilitates the implementation of a fault tolerance solution that enables applications to be restarted using a different device architecture and/or a different number of devices.

4.2. Implementation details

While no modifications are performed in the HPL library to implement this proposal, the CPPC tool is extended to cope with the particularities of HPL applications. Given a HPL program, the CPPC compiler automatically instruments its code to add fault tolerance support by performing three major

actions: inserting checkpoints, registering the necessary host private variables or shared variables, generating the appropriate consistency protocol routines, and identifying the non-portable state recovery blocks.

First, the compiler identifies the most computationally expensive loops in the host code by performing an heuristic computational load analysis [3] and a `CPPC_Do_checkpoint()` call is inserted at the beginning of the body of these loops, in between kernel invocations. The most computationally expensive loops are those that perform the core of the computation, and thus take the longest time to execute, allowing the user to specify an adequate checkpointing frequency.

Once the checkpoints are located, the CPPC compiler automatically registers the necessary data for the successful recovery of the application during restart. As established by the design of the proposal, by locating checkpoints in the host code, only the host private state and the shared state between host and devices need to be included in the checkpoint files. The CPPC compiler analyses the host code to identify which host private variables are alive when checkpointing, inserting the appropriate calls to `CPPC_Register()`. The compiler is extended to cope with HPL `Array` objects, identifying which of them correspond to shared variables alive when checkpointing, and using the HPL `data()` method to obtain a raw pointer to the data to be registered.

Regarding data consistency, both the CPPC compiler and the CPPC library have been extended. Now, the CPPC compiler automatically generates a consistency protocol routine (`CONSISTENCY_<checkpoint_number>()`) related to each checkpoint call introduced in the application code. This routine performs the synchronizations and data transfers required to ensure the

consistency of the data included in the checkpoint files. The consistency protocol routine works as a callback function: it is passed as an argument to the new CPPC library `CPPC_Consistency_protocol_ref()` routine, so that, when a checkpoint call triggers a checkpoint generation, the appropriate callback consistency protocol routine is invoked. Using the information from the live variable analysis performed during the registration process, the CPPC compiler identifies which of the registered shared variables may be modified within the kernels. These potentially inconsistent shared variables necessarily correspond with shared `Arrays` in global memory. The consistency protocol routine is implemented by invoking the HPL `data()` method on those shared `Arrays`, which performs the necessary synchronizations and data transfers. Moreover, both synchronizations and data transfers are only performed when the host copy of the variable is inconsistent, otherwise, both operations are avoided, thus, minimizing the consistency protocol overhead.

In order to preserve the portability features of both the checkpointing and the heterogeneous framework back-ends, the CPPC compiler is extended to avoid the inclusion in the checkpoint files of the non-portable state specific to heterogeneous applications. Instead, such non-portable state is recovered by the re-execution of those blocks of code responsible for its creation in the original execution. The CPPC compiler identifies as non-portable state the setup of the available devices at runtime, as well as the kernels definitions and compilations. As a result, the proposal is completely independent of the machine, and applications can be restarted using different hosts and/or devices.

Figure 4 shows, in boldface, the instrumentation generated by the com-

```
void kernel_1(Array<int, 1, Global> a1,/*INPUT*/
   Array<int, 1, Global> a2,/*INPUT*/
   Array<float, 1, Global> tmp,/*OUTPUT*/
   Array<float, 1, Constant> b,/*INPUT*/
   Array<int, 0, Global> i);/*INPUT*/

void kernel_2(Array<int, 1, Global> a2,/*OUTPUT*/
   Array<float, 1, Global> tmp,/*INPUT*/
   Array<int, 1, Local> c,
   Array<float, 1, Global> a3);/*INPUT&OUTPUT*/

void CONSISTENCY_1(){
   a2.data();
   a3.data();
}

Array<int, 1, Global> a1(N), a2(N);
Array<float, 1, Global> a3(N), tmp(N);
Array<float, 1, Constant> b(M);
Array<int, 1, Local> c(M);

int main()
{
   CPPC_Init_configuration();
   CPPC_Init_state();
   if (CPPC_Jump_next()) goto REGISTER_BLOCK_1;
   [...]
```

```
   REGISTER_BLOCK_1:
   CPPC_Register(&i, [...]);
   CPPC_Register(a1.data(), [...]);
   CPPC_Register(a2.data(), [...]);
   CPPC_Register(a3.data(), [...]);
   CPPC_Register(b.data(), [...]);
   if (CPPC_Jump_next()) goto RECOVERY_BLOCK_1

   [...]

   RECOVERY_BLOCK_1:
   /* kernel_1 and kernel_2 are associated to their
   OpenCL C kernels using the HPL API (not shown) */

   CPPC_Consistency_protocol_ref(&CONSISTENCY_1);
   if (CPPC_Jump_next()) goto CKPT_BLOCK_1;

   [...]
   for(i = 0; i < nIters; ++i){
      CKPT_BLOCK_1:
      eval(kernel_1)( a1, a2, tmp, b, i);
      eval(kernel_2)( a2, tmp, c, a3);
   }
   [...]

   CPPC_Shutdown();
}
```

Figure 4: The CPPC compiler locates the checkpoint at the beginning of the main loop, generates the appropriate registration calls for the live variables, and determines that the consistency protocol must be applied only to the `Arrays` `a2` and `a3`, as no other registered `Array` may be modified by the kernels.

piler for the HPL application used as example in Section 2.2 (Figure 2). HPL simplifies the application of the consistency protocol, as the library automatically tracks at runtime the most recently updated copy of a variable. Moreover, HPL also ensures that the correct copy of a shared variable is saved. For example, if OpenCL applications were targeted, there could exist situations in which a compile-time analysis could not be able to detect which copy of a shared variable is the most recent one. In that situation, to ensure correctness, the compiler would need to register both the copy of the variable in the host memory and the copy in the device memory (assuming for simplicity that it is only used in one device), doubling the state included in the checkpoint files and, thus, introducing more overhead when checkpointing.

4.3. Restart portability and adaptability

As commented previously, the implementation of the proposal pays special attention to the preservation of the portability features of both frameworks, allowing applications to be restarted using different hosts and/or devices. Additionally, by exploiting the high programmability of HPL via the instrumentation introduced by CPPC, applications can be restarted using a different number of devices. Two types of applications can be distinguished: pseudo-malleable applications and malleable applications, being the late ones able to fully adapt to a different number of devices at restart time. The CPPC compiler inserts the same instrumentation for both types of applications and it distinguishes one from another by activating a flag in the instrumentation code.

Pseudo-malleable applications are those in which the distribution of data and/or computation must be preserved when restarting, otherwise, the restart will not be successful. The CPPC compiler determines that an application is pseudo-malleable when any of the checkpointed variables has a dependency with the number of devices available at runtime. For instance, in the example code shown in Figure 5a, the registered variable `d` has such a dependency. These applications can be restarted using a larger or smaller number of physical devices but the same number of virtual devices. When fewer physical devices are used to recover the application, some of them will have to perform extra computations. When using a larger number of physical devices, some of them will not perform any computation. As shown in Figure 5b, the CPPC compiler inserts the instrumentation to perform the following actions:

- Saving and recovering the number of devices originally used by the

```
[...]
int num_devices = /*Available number of devices*/

[...]
Array<int, 1, Global> d(num_devices*N);
[...]

for(j = 0; j < T; j++){
  for(i = 0; i < num_devices; i++){
    device = HPL::Device(DEV_TYPE,i);
    eval(kernel).device(device)(
                d(Tuple(i*N, (i+1)*N-1)));
  }

  [...]
}

[...]
```

(a) Original.

```
[...]
int pseudomalleable=1;
int num_devices = /*Available number of devices*/
int real_devices=num_devices;
int orig_devices=num_devices;

/*Register/Recover original number of devices*/
CPPC_Register(&orig_devices,[...]);
if(CPPC_Jump_next() && (pseudomalleable==1)){
  /*Force same number of devices during restart*/
  num_devices=orig_devices;
}
[...]
Array<int, 1, Global> d(num_devices*N);
CPPC_Register(d.data(),[...]);
[...]
for(j = 0; j < T; j++){
  CPPC_Do_checkpoint(1);
  for(i = 0; i < num_devices; i++){
    device = HPL::Device(DEV_TYPE,i%real_devices);
    eval(kernel).device(device)(
                d(Tuple(i*N, (i+1)*N-1)));
  }
  [...]
}
[...]
```

(b) Instrumented by CPPC.

Figure 5: Automatic instrumentation of pseudo-malleable applications.

application (with the variable `orig_devices`).

- Setting the number of devices used to the original number when the `pseudomalleable` flag is activated.

- Modifying all the references to particular devices to ensure they correspond with a physical device by using the `real_devices` variable.

At runtime, HPL transparently manages the data allocations and transfers into the devices memory, releasing the CPPC instrumentation of this duty.

On the other hand, malleable applications are those which can be adapted to a different number of devices during the restart, obtaining an optimal data and computation distribution while ensuring the correctness of the results. The CPPC compiler identifies malleable applications when none of the registered variables presents dependencies with the number of devices available

```
[...]
int num_devices = /*Available number of devices*/

[...]
Array<int, 1, Global> d(N);
Array<int, 1, Global> * v_d[MAX_GPU_COUNT];
for(i = 0; i < num_devices; ++i){
  //Tuple builds HPL subarrays references
  v_d[i] = &d(Tuple(ini_p,end_p));
}
[...]

for(j = 0; j < T; j++){
  eval(kernel_1).device(v_devices)(v_d);
  [...]
}
[...]
```

(a) Original.

```
[...]
int pseudomalleable=0;
int num_devices = /*Available number of devices*/
int real_devices=num_devices;
int orig_devices=num_devices;

/*Register/Recover original number of devices*/
CPPC_Register(&orig_devices,[...]);
if(CPPC_Jump_next() && (pseudomalleable==1)){
  /*Force same number of devices during restart*/
  num_devices=orig_devices;
}
[...]
Array<int, 1, Global> d(N);
Array<int, 1, Global> * v_d[MAX_GPU_COUNT];
CPPC_Register(d.data(),[...]);
//BLOCK OF CODE REEXECUTED UPON RESTART
for(i = 0; i < num_devices; ++i){
  //Tuple builds HPL subarrays references
  v_d[i%real_devices] = &d(Tuple(ini_p,end_p));
}//END BLOCK OF CODE REEXECUTED UPON RESTART
[...]
for(j = 0; j < T; j++){
  CPPC_Do_checkpoint(1);
  eval(kernel_1).device(v_devices)(v_d);
  [...]
}
```

(b) Instrumented by CPPC.

Figure 6: Automatic instrumentation of malleable applications.

at runtime. The high programmability of HPL simplifies the implementation of malleable applications. A tipical pattern is shown in Figure 6a, in which an array of references `v_d` is built from a single unified image of the data, the array `d`. These references represent a particular distribution of data among the devices, which is actually performed by the `eval` routine. As shown in Figure 6b, the CPPC compiler registers the single unified image of each array, which can then be split among an arbitrary different number of devices when the application is restarted. The compiler includes the instrumentation used for pseudomalleable applications, but now the `pseudomalleable` flag is deactivated.

Table 2: Testbed benchmarks description and original runtimes.

		BENCHMARCK DESCRIPTION	N.GPU	RUNTIME (SECONDS)
SINGLEDEVICE	**FT**	Fourier Transform Class B	1	43.31
	Floyd	Floyd-Warshall algorithm on 2^{14} nodes	1	260.64
	Spmv	Sparse matrix-vector product 2^{15}rows, 1e4 iters	1	153.14
MULTIDEVICE	**FTMD**	Fourier Transform Class B	1	51.27
			2	34.58
	Summa	Matrix multiplication NxN, N=2^{13}	1	45.50
			2	26.20
	MGMD	Multi-Grid Class B	1	26.45
			2	20.06
	Shwa1ls Short	Simulation of a contaminant 1 week, 400x400 cell mesh	1	271.48
			2	238.03
	Shwa1ls Large	Simulation of a contaminant 6 week, 800x800 cell mesh	1	10203.89
			2	6515.67

5. Experimental results

The application testbed used, summarized in Table 2, is comprised of seven applications already implemented in HPL by our research group: three single-device applications (FT, Floyd, Spmv) and four multi-device applications (FTMD, Summa, MGMD, and Shwa1ls). Spmv is a benchmark of the SHOC Benchmarks suite [9]. Floyd is from the AMD APP SDK. FT, FTMD, and MGMD are benchmarks of the SNU NPB suite [10]. Summa implements

the algorithm for matrix multiplication described in [11]. Finally, Shwa1ls is a real application that performs a shallow water simulation parallelized for multiple GPUs in [12]. Most of the experimental results shown in this section were carried out with Shwa1ls-Short configuration, since it presents a reasonable execution time to carry out exhaustive experiments. However, some of the experiments have also been conducted with the Shwa1ls-Large configuration, so as to show the impact of the checkpointing operation in a long-running application. Table 3 further characterizes the testbed heterogeneous applications. Regarding the kernels, it shows the number of kernel functions, the total number of invocations to the kernels, and the execution time of the longest kernel in the application. Additionally, the table presents how many buffers in global and local memory are used (showing both the number and their total size), and also the overall ratio between the application data inputs and outputs and the intermediate results. As can be seen in the table, the testbed applications cover a wide range of scenarios.

Table 4 details the hardware used for the experiments. The first system is used for the experiments in Sections 5.1 and 5.2, while all the systems are used in Section 5.3 to show the portability and adaptability of the solution. The tests were performed writing and reading the checkpoint files from the local storage of the node (SATA magnetic disks). The CPPC version used was 0.8.1, working in tandem with HDF5 v1.8.11. The GNU compiler v4.7.2 was used with optimization level O3 in systems #1 and #2, while Apple LLVM version 7.3.0 (clang-703.0.31) is used in system#3. Each result is the average of 15 executions. The original runtimes of the testbed benchmarks on system#1, shown in Table 2, are on average only 0.4% slower than their

Table 3: Testbed benchmarks characterization.

		N.GPU	KERNELS			GLOBAL BUFFERS		LOCAL BUFFERS		APP. DATA RATIO*
			#FUNC.	#CALLS	LONGEST KERNEL TIME (S)	#	SIZE	#	SIZE	
SINGLEDEVICE	**FT**	1	8	111	0.63757	10	2307 MB	1	512 B	0.64
	Floyd	1	1	16384	0.01959	3	2048 MB	0	0 B	0.60
	Spmv	1	1	10000	0.01369	6	410 MB	1	512 B	1.00
MULTIDEVICE	**FTMD**	1	17	156	0.49146	22	2560 MB	1	736 B	0.80
		2	17	312	0.24630	22	3072 MB	1	736 B	0.80
	Summa	1	1	8	4.97366	14	1536 MB	2	256 B	0.92
		2	1	64	0.61972	50	1536 MB	8	512 B	0.92
	MGMD	1	43	5623	6.50812	25	456 MB	2	16 B	0.57
		2	43	17182	3.25574	25	466 MB	4	32 B	0.57
	Shwa1ls Short	1	3	3356465	0.00024	17	17 MB	1	256 B	0.68
		2	3	6712930	0.00025	23	17 MB	2	256 B	0.81
	Shwa1ls Large	1	3	40327770	0.00083	17	68 MB	1	256 B	0.68
		2	3	80655540	0.00072	23	68 MB	2	256 B	0.81

* Application data ratio calculated as $\frac{InputData+OutputData}{InputData+OutputData+IntermediateData}$

native OpenCL equivalents.

5.1. Instrumentation and checkpoint overhead

In a failure-free scenario, two main sources of overhead can be distinguished: the instrumentation of the code and the checkpoint operation over-

Table 4: Hardware platform details

		SYSTEM#1	SYSTEM#2	SYSTEM#3
HOST	OPERATING SYSTEM	CentOS 6.7		MacOS X 10.11
	PROCESSOR	2x Intel E5-2660		Intel I7-3770
	FREQUENCY GHz	2.20		3.4
	#CORES	8 (16 HT)		4 (8 HT)
	MEM. CAPACITY GB	64		16
	MEM. BANDWIDTH GB/s	51.2		25.6
DEVICE	PROCESSOR	Nvidia K20m	Xeon PHI 5110P	Nvidia GeForce GTX 675MX
	FREQUENCY GHz	0.705	1.053	0.6
	#CORES	2496	60 (240 HT)	960
	MEM. CAPACITY GB	5	8	1
	MEM. BANDWIDTH GB/s	208	320	115.2
	DRIVER	NVIDIA 325.15	Intel OpenCL 4.5.0.8	NVIDIA 310.42

head. The instrumentation overhead corresponds to the CPPC instrumented applications without generating any checkpoint files. The checkpoint overhead is measured in the execution of the CPPC instrumented versions generating one checkpoint file, and it includes the instrumentation, the consistency protocol and the checkpoint file generation overheads. Note that the checkpointing frequency is a user-defined parameter, that can be modified to generate more or fewer checkpoint files depending on the user requirements. In all the experiments the dumping frequency is set to take a single checkpoint when 75% of the computation has been completed, as this accounts for an adequate checkpointing frequency given the testbed applications runtimes. In order to assess the performance also in long-running applications

where more checkpoints would have to be done during a normal execution, we also show results for the Shwa1ls-Large application, where 10 checkpoint files (one every 17 minutes when using one GPU and one every 11 minutes when using 2 GPUs) are performed.

Table 5 analyses the instrumentation and checkpoint overheads. First, the original runtimes (in seconds) are presented. Then, both the instrumentation overhead absolute value (the difference with the original runtimes, in seconds) and relative value (that difference normalized with respect to the original times, in percentage) are shown. The instrumentation overhead is negligible, always below a few seconds. In the application Summa running on two GPUs, the instrumentation overhead is negative, explained by the optimizations applied by the GNU compiler.

Regarding the checkpointing overhead, Table 5 also presents, for each experiment, the number of checkpoints taken, and their frequency, i.e. every how many iterations of the most computationally expensive loop a checkpoint file is generated, as well as the absolute and relative values of the checkpoint overhead. Note that the checkpointing overhead includes both the cost of the instrumentation and the cost of taking as many checkpoints as specified in the table. In addition to this, the times of the actions performed when a single checkpoint is taken, as well as the checkpoint file size, are included in the table under the title "Checkpoint operation analysis". Figure 7 presents a summary of this information: the runtimes when generating one checkpoint file are normalized with respect to the original runtimes of each application. In addition, both the consistency protocol and the checkpoint file generation times are highlighted.

Table 5: Instrumentation and checkpoint overhead analysis for the testbed benchmarks.

		N.GPU	ORIGINAL RUNTIME (s)	INSTRUM. OVERHEAD *1		CHECKPOINTs TAKEN *2		CHECKPOINT OVERHEAD *1		CHECKPOINT OPERATION ANALYSIS		
				Δ(s)	[%]	#	N	Δ(s)	[%]	CONSISTENCY PROTOCOL (s)	CHECKPOINT GENERAT. (s)	FILE SIZE (MB)
SINGLEDEVICE	**FT**	1	43.31	0.65	[1.5]	1	15	2.56	[5.9]	0.140	1.835	769.54
	Floyd	1	260.64	0.31	[0.12]	1	12288	6.03	[2.32]	4.323	4.878	2048.02
	Spmv	1	153.14	0.18	[0.12]	1	7500	1.95	[1.27]	3.320	0.980	410.15
MULTIDEVICE	**FTMD**	1	51.27	0.98	[1.91]	1	15	2.74	[5.34]	0.138	1.823	768.04
		2	34.58	0.58	[1.67]	1	15	1.79	[5.17]	0.145	1.827	768.04
	Summa	1	45.50	0.26	[0.57]	1	3	4.08	[8.97]	5.101	3.665	1536.07
		2	26.20	-0.09	[-0.35]	1	12	4.21	[16.06]	0.768	3.660	1536.1
	MGMD	1	26.45	0.55	[2.08]	1	15	1.77	[6.71]	0.513	1.072	300.78
		2	20.06	0.74	[3.7]	1	15	1.8	[8.98]	0.130	1.085	303.53
	Shwa1ls Short	1	271.48	0.78	[0.29]	1	503470	0.78	[0.29]	0.003	0.022	5.96
		2	238.03	0.98	[0.41]	1	503470	1.04	[0.44]	0.003	0.022	5.96
	Shwa1ls Large	1	10203.89	1.18	[0.01]	10	733236	2.61	[0.03]	0.010	0.077	21.42
		2	6567.82	1.39	[0.02]	10	733236	2.33	[0.04]	0.013	0.082	21.41

$*_1$) Δ(s) absolute overhead in seconds. [%] relative overhead with respect to the original runtimes.

$*_2$) # Total number of checkpoints taken. N checkpointing frequency, iterations between checkpoints.

The total overhead introduced in the applications when checkpointing is small. Its absolute value ranges from a minimum of 0.78 seconds for Shwa1ls

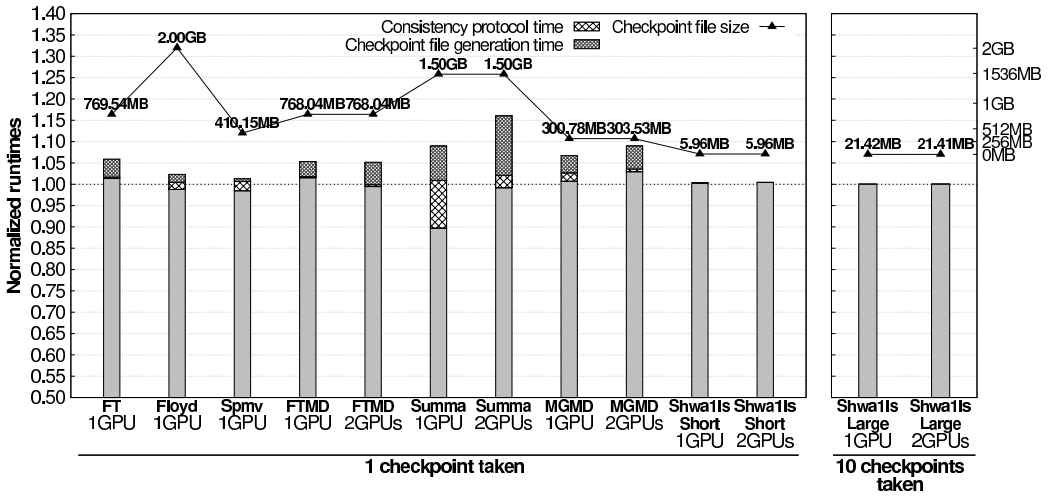

Figure 7: Normalized checkpoint overhead for the testbed benchmarks.

running on one GPU and generating a 5.96 MB checkpoint file, to a maximum value of 6.03 seconds for Floyd when saving 2 GB of data. The checkpoint file generation overhead includes the state management operations, the copy in memory of the data, and the creation of a thread to dump the data to disk. The actual dumping to disk is performed by the multithreaded dumping of CPPC in background, overlapping the checkpoint file writing to disk with the computation of the application. As can be observed, the checkpoint file generation overhead heavily depends on the size of the checkpoint files. Besides, the impact of this overhead obviously depends on the original application runtime.

Regarding the consistency protocol times, they are inherently dependent on the application. These times are the addition of the time spent in the synchronizations and the data transfers performed by the protocol. For the applications Floyd, Spmv and Summa, the absolute checkpoint overhead is lower than the addition of the consistency protocol and checkpoint file gen-

eration times. This situation can also be observed in Figure 7, where the consistency protocol is represented below the value 1 for those applications. This occurs because some operations in the original application take slightly less time when a checkpoint file is generated, due to the synchronizations performed by the consistency protocol. In Floyd, experimental results show that these synchronizations reduce the time spent by OpenCL, used as back-end by HPL, in some inner operations. In Summa and Spmv these synchronizations reduce the time spent in further synchronizations that exist in the original application code. This situation can happen quite frequently, since heterogeneous applications present synchronizations at some point of their execution.

5.2. Restart overhead

The restart overhead plays a fundamental role in the global execution time when failures arise. The restart process includes all the operations required to reach the point where the checkpoint file was generated. It can be broken down into two parts: reading the checkpoint file and positioning in the application. In heterogeneous applications, the positioning overhead can be split in the host and the devices positioning. The host positioning is determined by the operations that must be re-executed in the host during the restart and by the state that has to be moved to the proper memory location. The devices positioning is the set-up of the devices, including the kernels compilation and the transfers of the recovery data to their memory. Figure 8 shows the restart runtimes when the applications are recovered from a failure using the checkpoint files generated when the application has completed the 75% of its computation (right bars). These times include all

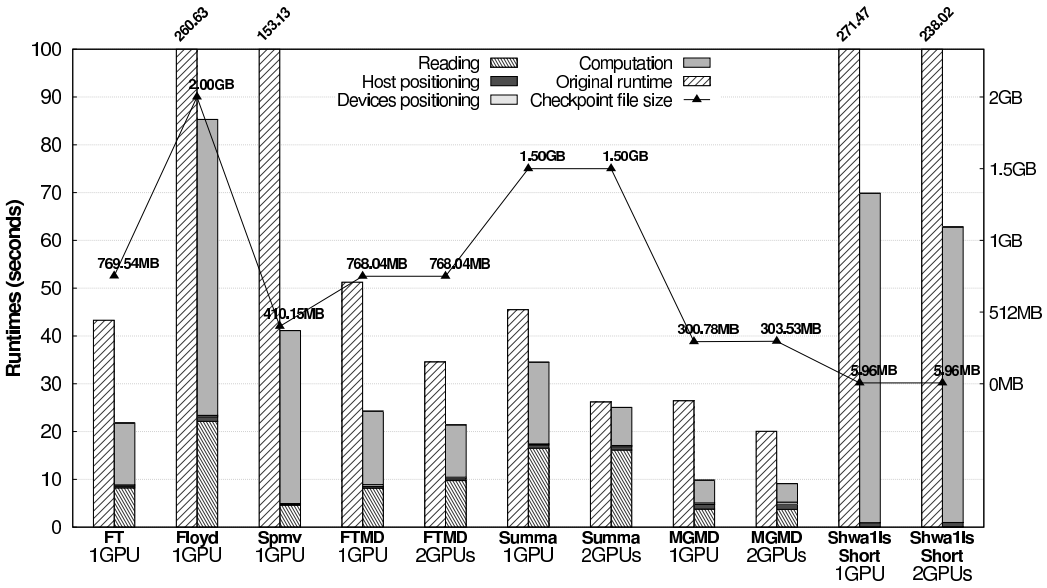

Figure 8: Runtimes for the testbed benchmarks after a failure. When a failure occurs, applications can be restarted from a checkpoint file instead of starting the execution from the beginning, which would consume more time.

the costs: the restart overhead (reading and positioning) and the application computation from the restart point to the end (the 25% of the total execution in these experiments). Upon a failure, when non-fault tolerance mechanisms are used, the applications have to be re-executed from the beginning. Thus, for comparison purposes the original runtimes are also represented in the figure (left bars).

Note that the restart overhead is always below 25 seconds, the positioning overhead being at most 3 seconds. Thus, the restart overhead is mainly determined by the reading phase, which is related to the checkpoint file sizes. Only in Shwa1ls the positioning times represent a larger percentage of the restart overhead, due to the small checkpoint file size of this application.

In some applications the reading phase has a high impact due to the short runtimes, making the restart runtime close to the original runtime. However, restarting the application from a previous checkpoint is always better than starting it from the beginning of the computation, and the benefits of including fault tolerance mechanisms in long-running HPC heterogeneous applications will be unquestionable.

5.3. Portability and adaptability benefits

The restart experiments presented in the previous subsection recovered the application using the same host and devices available during the original execution. This subsection presents the results when restarting using a different host or device architecture and/or a different number of devices. All the systems described in Table 4 will now be used.

Figure 9 shows the restart runtimes when recovering the applications using the same host and different devices: the same GPUs, Xeon Phi accelerators, and the CPU. The checkpoint file sizes are also shown. The devices positioning times vary with the device architecture, as the kernels compilation times are larger when using the Intel OpenCL driver in the Xeon Phi and CPU experiments. The computation times on the different devices are consistent with the original runtimes in the same device. For instance, in Spmv the original runtime is larger when using a GPU than when using a Xeon Phi accelerator, thus, the same tendency can be observed in the restart runtimes.

Figure 10 presents the restart runtimes when using a number of GPUs that is different from the one used in the execution where the checkpoint files were generated. In our testbed benchmarks, Summa and MGMD are

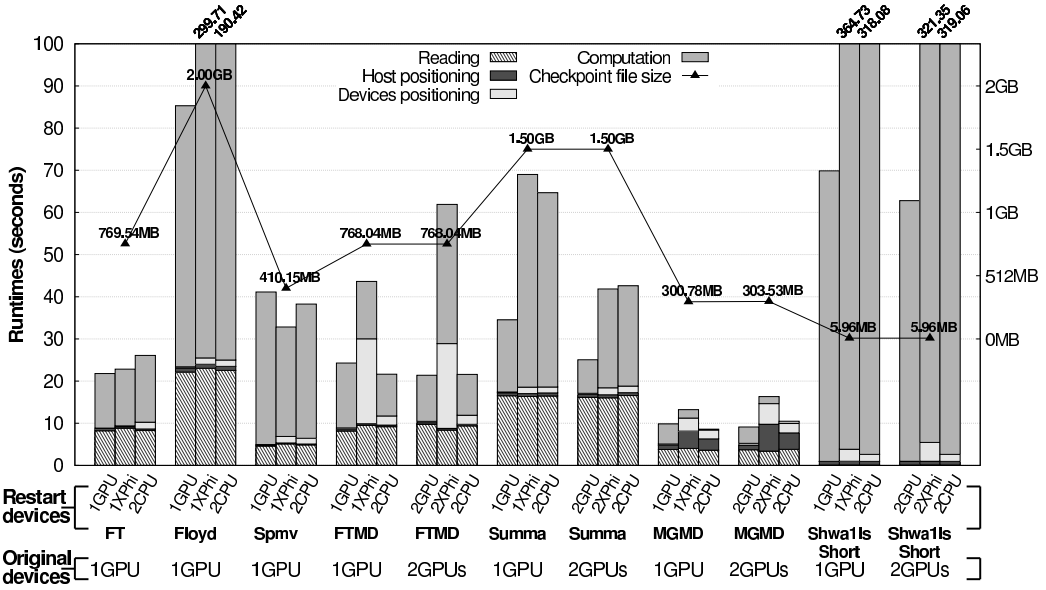

Figure 9: Restart runtimes for the testbed benchmarks on different device architectures.

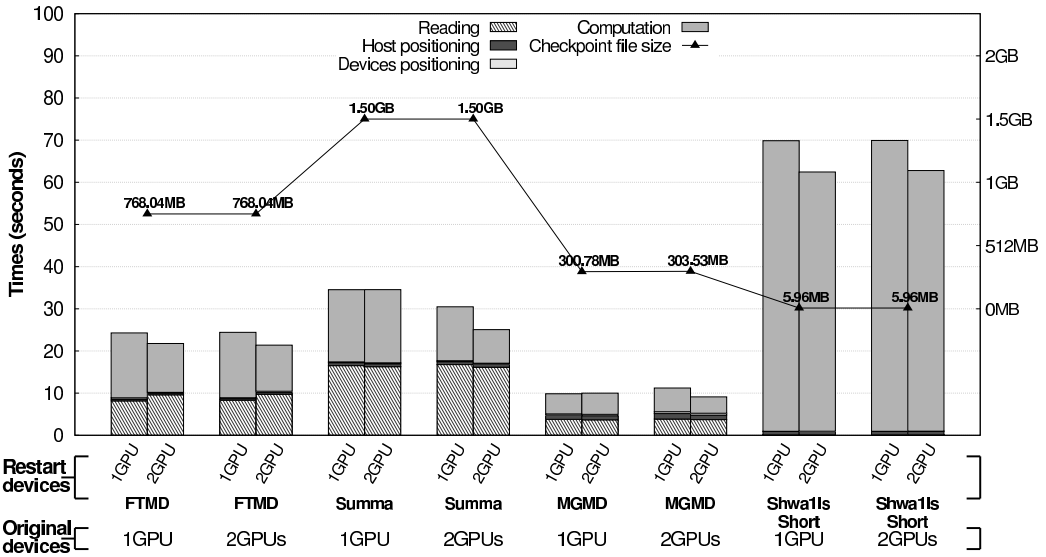

Figure 10: Restart runtimes for the testbed benchmarks using a different number of GPUs.

pseudo-malleable applications, while FTMD and Shwa1ls are fully malleable applications. As can be observed, it is possible to restart all the applications

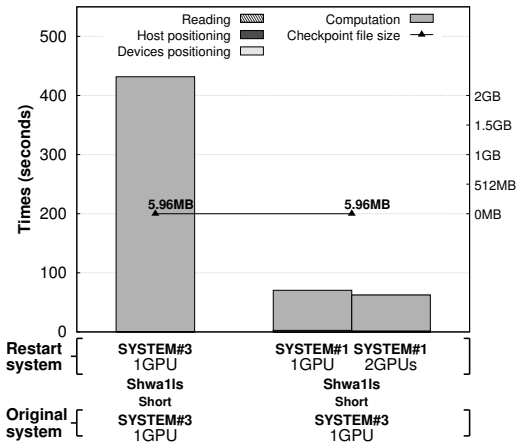

Figure 11: Restart runtimes for Shwa1ls using a different host, a different number and architecture of devices, and a different operating system.

using a larger or a smaller number of devices. Besides, the restart runtimes of the malleable applications (FTMD and Shwa1ls) are not conditioned by the number of devices used for the checkpoint file generation, and, instead, these times are only influenced by the number of devices used during the restart execution, as an optimal distribution among them is performed.

Finally, Figure 11 shows the restart times when the application Shwa1ls is recovered in system#3 and system#1 from the checkpoint files generated in system#3. In this scenario, the application is restarted using a different host with a different operating system, a different device architecture and a different number of devices, demonstrating the portability and adaptability benefits of the proposal. The combination of CPPC and HPL allows applications to continue the execution after a failure occurs in one or several of the running devices, and/or in the host CPUs. Besides, in heterogeneous cluster systems, the proposed solution allows any application to start its execution

using the available devices, even without using any accelerators at all, and to later continue its execution using a more appropriate set of devices, or vice-versa, depending on the availability of resources in the cluster.

6. Related work

Fault tolerance for parallel applications is a very active research topic with a large number of approaches published in the last two decades. Regarding fault tolerance in heterogeneous applications, approaches exist based on Algorithm-Based Fault Tolerance (ABFT), in which extra information in the application is used to check the results for errors. Such a solution is presented in A-ABFT [13], which describes an ABTF proposal for matrix operations on GPUs. However, these solutions are highly specific for each particular application and algorithm.

More generic solutions are based on checkpointing. On the one hand, solutions exist that have focused on detecting soft errors, which usually do not result in a fail-stop error but cause a data corruption. A solution of this kind is VOCL-FT [14], which combines ECC error checking, logging of OpenCL inputs and commands, and checkpointing for correcting those ECC errors in the device memory that cannot be corrected by the device. On the other hand, the following paragraphs comment on other proposals that, like the one presented in this paper, are focused on fail-stop failures.

Bautista et al. [15] propose a user-level diskless checkpointing using Reed-Solomon encoding for CPU-GPU applications. The checkpointing frequency is determined by the data transfers in the host code. When the CUDA kernels and the data transfers are finished, an application-level strategy checkpoints

the host memory. The main drawback of diskless checkpointing is its large memory requirements. As such, this scheme is only adequate for applications with a relatively small memory footprint at checkpoint. Besides, some GPU applications postpone the data transfers until the end of the execution, which will be translated in an unsuitable checkpointing frequency.

CheCUDA [16] and CheCL [17] are checkpointing tools for CUDA and OpenCL applications, respectively. Both are implemented as add-on packages of BLCR [18], a system-level checkpoint/restart implementation for Linux clusters. Checkpointing is triggered by POSIX signals: after receiving the signal, in the next synchronization between the host and the devices, the user data from the device memory is transferred to the host memory, and the host memory is checkpointed using BLCR. Also, both use wrapper functions to log the CUDA or OpenCL calls, in order to enable their re-execution during the restart process. CheCUDA requires no CUDA context to exist when checkpointing, as otherwise BLCR fails to restart. For this reason the context is destroyed before every checkpoint and recreated afterwards, as well as during the restart process, using the log of CUDA calls. In CheCL a different strategy is used. The OpenCL application is executed by at least two processes: an application process and an API proxy. The API proxy is an OpenCL process and the devices are mapped to its memory space, allowing the application process to be safely checkpointable.

NVCR [19] uses a protocol similar to CheCUDA, however, it supports CUDA applications developed using the CUDA driver API and CUDA runtime API , without the need to recompile the application code. NVCR uses wrapper functions to log the CUDA calls. It is also based in BLCR, thus, as

in CheCUDA, all CUDA resources have to be released before every checkpoint and recreated afterwards, as well as during the restart process, using a replay strategy to re-execute the CUDA calls from the log. However, the replay during the restart process relays on the reallocation of the memory chunks at the same address as before checkpointing, which is not guaranteed by NVIDIA and could not work correctly in certain environments.

Laosooksathit et al. [20] model and perform simulations to estimate the performance of checkpoints relying on virtualization and CUDA streams that are applied at synchronization points under the control of a model, but they offer no actual implementations.

HeteroCheckpoint [21] presents a CPU-GPU checkpointing mechanism using non-volatile memory (NVM). The application is instrumented by the programmer, explicitly indicating where and when a checkpoint is taken and which CUDA variables need to be checkpointed. CUDA streams are used to enable parallel data movement and the programmer can specify which CUDA variables are not modified in the kernels executed before a checkpoint, allowing variables to be pre-copied before a checkpoint starts. Also, redundant data between two checkpoints, as in an incremental checkpointing, do not cause unnecessary data transfers. In this proposal, when checkpointing, the host and the device are synchronized and data is transferred from the device memory to the NVM via the host memory.

Snapify [22] is a specific solution for Xeon Phi accelerators. It is a transparent, coordinated approach to take consistent snapshots of the host and coprocessor processes of Xeon Phi offload applications. It is based on BLCR and it applies a device-side checkpointing taking into account the data pri-

vate to an offload process, and dealing with the distributed states of the processes that conform the offload application. When the host receives a checkpoint signal, it pauses the offload application, drains all the communication channels, captures a snapshot of the offload processes and the host, and resumes the execution. Snapify can be used for checkpoint and restart, process migration, and process swapping.

Table 6 summarizes the main features of the fail-stop checkpoint-based solutions commented above, specifying: the supported devices, the checkpointing granularity (application level vs. system-level) and frequency, and whether the restart process is portable (can take place in a different machine, using both different host and devices) and adaptable (can take place using a different number of accelerator devices). Most of the proposals are focused on CUDA applications, which restricts them to GPUs from a specific vendor. Five of them use a system-level approach. System-level checkpointing simplifies the implementation of a transparent solution, in which no effort from users is needed to obtain fault tolerance support, however, it results in larger checkpoint files and, thus, in larger overhead introduced in the application. Moreover, system-level checkpointing binds the restart process to the same host, thus, the restart will not be possible on different host architectures and/or operating systems, as the checkpoint files may contain non-portable host state. Furthermore, in order to allow the successful checkpointing of the application, the system-level strategy forces the use of an API proxy in CheCL, or the destruction and reconstruction of the devices context every time a checkpoint is taken in CheCUDA and NVCR, which have a negative influence in the checkpointing overhead.

Table 6: Related work overview.

	SUPPORTED DEVICES			CHECKPOINTING		RESTART	
	NVIDIA	XEONPHI	GENERIC	GRANULARITY	FREQUENCY	PORTABLE	ADAPTABLE
Bautista et al. [15]	✓			Application level	Timer decides when data transfer originates ckpt		
CheCUDA [16]	✓			System level (BLCR [18])	Signal triggers ckpt in next host-devs synchro.		
CheCL [17]			✓	System level (BLCR [18])	Signal triggers ckpt in next host-devs synchro.		
NVCR [19]	✓			System level (BLCR [18])	Signal triggers checkpoint		
Laosooksathit et al. [20]	✓			System level (VCCP [23])	In kernels at synchros. chosen by a model		
HeteroCkpt [21]	✓			Application level	Indicated by the user in the host code		
Snapify [22]		✓		System level (BLCR [18])	Signal triggers checkpoint		
Proposal: CPPC+HPL			✓	Application level	User-defined freq. at points chosen by the tool	✓	✓

The solution presented in this work targets HPL applications, based on OpenCL. Thus, this proposal is not tied to a specific device architecture or vendor. By combining an OpenCL back-end and a host-side checkpointing strategy, the approach provides several advantages to the checkpoint files: their size is reduced and they are decoupled from the particular character-

istics and number of devices used during their generation. Moreover, the application-level portable checkpointing further reduces the checkpoint files size and also decouples them from the host machine. Therefore, applications can be recovered not only using a different device architecture and/or a different number of devices, but also using a host with a different architecture and/or operating system. To the best of our knowledge, no other work provides such portability and adaptability benefits to heterogeneous applications.

7. Concluding remarks

This work presents a fault tolerance solution for heterogeneous applications. The proposal targets HPL applications thus, it is not tied to any particular accelerator vendor. Fault tolerance support is obtained by using a host-side application-level checkpointing. The host-side approach avoids the inclusion in the checkpoint files of the device/s private state, while the application-level strategy avoids the inclusion of not relevant host data, thus minimizing the checkpoint files size. This approach provides portability and efficiency, while ensuring an adequate checkpointing frequency, as most of HPC heterogeneous applications execute a large number of short kernels.

The proposal is implemented by combining CPPC, a portable and transparent checkpointing infrastructure, and HPL, a C++ library for programming heterogeneous systems on top of OpenCL. A consistency protocol, based on synchronizations and data transfers, ensures the correctness of the saved data. The protocol overhead is minimized by the HPL lazy copying policy for the data transfers. The host-side application-level strategy and the com-

bination of CPPC and HPL maximizes portability and adaptability, allowing failed executions to be resumed using a different number of heterogeneous computing resources and/or different resource architecture. The ability of applications to adapt to the available resources will be particularly useful for heterogeneous cluster systems.

The experimental evaluation shows the low overhead of the proposed solution, which is mainly determined by the saved state size. The restart experiments using hosts with different operating systems, different device architectures and different numbers of devices demonstrate the portability and adaptability of the proposal.

Future work includes the study of the proposed solution on heterogeneous MPI applications, which can also benefit from the use of a local recovery strategy. Using MPI implementations that provide resilience features, a failure in one of the devices can be managed locally by one MPI process, instead of forcing the roll-back of all the MPI processes executing the application.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Projects TIN2013-42148-P, TIN2016-75845-P and the predoctoral Grant of Nuria Losada ref. BES-2014-068066), by EU under the COST Program Action IC1305, Network for Sustainable Ultrascale Computing (NESUS), and by the Galician Government (Xunta de Galicia) and FEDER funds of the EU under the Consolidation Program of Competitive Research (ref. GRC2013/055). Authors would like to thank Moisés Viñas for his help and support in the use of HPL.

References

[1] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, D. Glasco, GPUs and the future of parallel computing, IEEE Micro 31 (5) (2011) 7–17.

[2] C. Di Martino, W. Kramer, Z. Kalbarczyk, R. Iyer, Measuring and Understanding Extreme-Scale Application Resilience: A Field Study of 5,000,000 HPC Application Runs, in: IEEE/IFIP International Conference on Dependable Systems and Networks, 2015, pp. 25–36.

[3] G. Rodríguez, M. J. Martín, P. González, J. Touriño, A Heuristic Approach for the Automatic Insertion of Checkpoints in Message-Passing Codes, Journal of Universal Computer Science 15 (14) (2009) 2894–2911.

[4] G. Rodríguez, M. J. Martín, P. González, J. Touriño, R. Doallo, CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications, Concurrency and Computation: Practice and Experience 22 (6) (2010) 749–766.

[5] M. Viñas, Z. Bozkus, B. B. Fraguela, Exploiting heterogeneous parallelism with the Heterogeneous Programming Library, Journal of Parallel and Distributed Computing 73 (12) (2013) 1627–1638.

[6] Khronos OpenCL Working Group, The OpenCL specification. Version 2.0.

[7] M. Viñas, B. B. Fraguela, Z. Bozkus, D. Andrade, Improving OpenCL programmability with the Heterogeneous Programming Library, in: International Conference on Computational Science, Vol. 51, 2015, pp. 110–119.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: IEEE International Symposium on Workload Characterization, 2009, pp. 44–54.

[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, J. S. Vetter, The Scalable Heterogeneous Computing (SHOC) Benchmark Suite, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 63–74.

[10] S. Seo, G. Jo, J. Lee, Performance characterization of the NAS Parallel Benchmarks in OpenCL, in: IEEE International Symposium on Workload Characterization, 2011, pp. 137–148.

[11] R. A. Van De Geijn, J. Watts, SUMMA: Scalable universal matrix multiplication algorithm, Concurrency-Practice and Experience 9 (4) (1997) 255–274.

[12] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J. A. García, M. J. Castro, R. Doallo, A multi-GPU shallow-water simulation with transport of contaminants, Concurrency and Computation: Practice and Experience 25 (8) (2013) 1153–1169.

[13] C. Braun, S. Halder, H. J. Wunderlich, A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units, in: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2014, pp. 443–454.

[14] A. J. Peña, W. Bland, P. Balaji, VOCL-FT: Introducing Techniques for Efficient Soft Error Coprocessor Recovery, in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 71:1–71:12.

[15] L. Bautista-Gomez, A. Nukada, N. Maruyama, F. Cappello, S. Matsuoka, Low-overhead diskless checkpoint for hybrid computing systems, in: International Conference on High Performance Computing (HiPC), 2010, pp. 1–10.

[16] H. Takizawa, K. Sato, K. Komatsu, H. Kobayashi, CheCUDA: A Checkpoint/Restart Tool for CUDA Applications, in: International Conference on Parallel and Distributed Computing, Applications and Technologies, 2009, pp. 408–413.

[17] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, H. Kobayashi, CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications, in: IEEE International Parallel Distributed Processing Symposium, 2011, pp. 864–876.

[18] P. H. Hargrove, J. C. Duell, Berkeley lab checkpoint/restart (BLCR) for Linux clusters, Journal of Physics: Conference Series 46 (1) (2006) 494.

[19] A. Nukada, H. Takizawa, S. Matsuoka, NVCR: A transparent checkpoint-restart library for NVIDIA CUDA, in: IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 104–113.

[20] S. Laosooksathit, N. Naksinehaboon, C. Leangsuksan, A. Dhungana, C. Chandler, K. Chanchio, A. Farbin, Lightweight Checkpoint Mechanism and Modeling in GPGPU Environment, in: Workshop on System-level Virtualization for High Performance Computing, Vol. 12, 2010, pp. 13–20.

[21] S. Kannan, N. Farooqui, A. Gavrilovska, K. Schwan, HeteroCheckpoint: Efficient Checkpointing for Accelerator-Based Systems, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 738–743.

[22] Rezaei, A. and Coviello, G. and Li, C.-H. and Chakradhar, S. and Mueller, F., Snapify: Capturing Snapshots of Offload Applications on Xeon Phi Manycore Processors, in: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, 2014, pp. 1–12.

[23] H. Ong, N. Saragol, K. Chanchio, C. Leangsuksun, VCCP: A transparent, coordinated checkpointing system for virtualization-based cluster computing, in: IEEE International Conference on Cluster Computing and Workshops, 2009, pp. 1–10.