Checkpoint Restart Support for Heterogeneous HPC Applications

Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez and Osman Unsal
Barcelona Supercomputing Center
Barcelona, Spain
{konstantinos.parasyris, kai.keller, leonardo.bautista, osman.unsal}@bsc.es

***Abstract*—As we approach the era of exa-scale computing, fault tolerance is of growing importance. The increasing number of cores as well as the increased complexity of modern heterogenous systems result in substantial decrease of the expected mean time between failures. Among the different fault tolerance techniques, checkpoint/restart is vastly adopted in supercomputing systems. Although many supercomputers in the TOP 500 list use GPUs, only a few checkpoint restart mechanism support GPUs.**

In this paper, we extend an application level checkpoint library, called fault tolerance interface (FTI), to support multi-node/multi-GPU checkpoints. In contrast to previous work, our library includes a memory manager, which upon a checkpoint invocation tracks the actual location of the data to be stored and handles the data accordingly. We analyze the overhead of the checkpoint/restart procedure and we present a series of optimization steps to massively decrease the checkpoint and recovery time of our implementation. To further reduce the checkpoint time we present a differential checkpoint approach which writes only the updated data to the checkpoint file. Our approach is evaluated and, in the best case scenario, the execution time of a normal checkpoint is reduced by 15x in contrast with a non-optimized version, in the case of differential checkpoint the overhead can drop to 2.6% when checkpointing every $30s$.

***Index Terms*—Fault Tolerance, Reliability, GPGPU, High Performance Computing Systems**

I. Introduction

The last decades supercomputers have increased in size and in total compute capability. Exascale computing is the next objective, which will bring even more computing power to scientific applications and to industries. However, several challenges arise with exascale computing. The two most important challenges are power consumption and error resiliency. As the number of components increase in large scale systems, the systems become more error prone, and thus more prone to failures [22]. It is expected that the next generation of high performance computing (HPC) machines will experience failures up to several times an hour, driving the need for effective fault resilience for building tomorrow's HPC systems [3].

Another important consideration for the fault resiliency of HPC systems is the increasing heterogeneity of the components within the system. The typical future exascale system will consist of nodes with general-purpose GPU (GPGPU) pipelines for extreme high performance, coupled with high performance multicore CPUs targeting single-thread performance. The GPUs provide high throughput required for exascale levels of computation, whereas the CPU cores handle hard-to-parallelize code sections and provide support for legacy applications [30]. However, GPUs are more error prone than CPUs. In *TSUBAME* 40% of the total number of failures are caused by GPU errors, while the number of CPU related failures is below 5% [25]. When injecting faults [34] to applications executing on CPUs only 2.3% of the injected faults manifest as errors, whereas in GPUs this percentage rises to 16-33%. For all these reasons the Mean Time Between Failures (MTBF) is expected to decrease even more in future systems.

To overcome failures, supercomputers use checkpoint restart (C/R) techniques, by storing the state of the computation in reliable storage. Upon a failure, the most recent state is used to restart the computation. As contemporary HPC applications are able to generate more information, the amount of data to be checkpointed increases. On one hand, the decrease of the MTBF results in higher checkpoint frequency to reduce the amount of recomputation. On the other hand, the increase in data size results in larger checkpoint files, and thus an increase to the overhead of the checkpoint procedure. To make things even worse, typically, in GPGPU HPC and with the introduction of CUDA-aware MPI and GPUDirect the applications memory footprint is distributed across several devices (CPU, GPU) [1]. This distribution of data impacts also the checkpoint procedure, as the data are transferred from the GPU side to the CPU side and then to be stored in stable storage. To summarize, checkpoints reduce heavily the system's efficiency. To maintain high productivity in supercomputers and large data centers, it is important to: i) reduce the programmers effort to implement checkpoints ii) reduce the total overhead of the checkpointing/restart procedure.

The main contributions of this article are:

- We extend a checkpoint library, called Fault Tolerance Interface (FTI) [2], to support checkpoint of data in multi-GPU/multi-node systems. Our method tracks the physical memory location of user defined virtual addresses without extending the library's API. The functionality handles CPU, GPU and Unified Memory Addresses (UVA). The implementation is open source and available in [14].
- We optimize the checkpoint/restart procedure. To be more precise, we perform several optimizations. We use an iterative process in which, each time we identify the bottleneck and we try to optimize it by exploiting the underlying hardware architecture. In the end, our imple-

 DOI 10.1109/CCGrid49817.2020.00-69

mentation is up to 15x and 5x faster in the checkpoint recovery procedure respectively than the unoptimized one.
- We implement differential checkpoint, a method that detects and stores to the checkpoint file only data that changed their value in comparison with the previous checkpoint. The method transparently handles CPU, GPU and UVA.
- Finally, we thoroughly evaluate our approach using different applications on a multi-node multi-GPU system. Our final implementation can perform a normal checkpoint of $1.8TB$ every 30 seconds while adding an average overhead of 13%. When applying differential checkpoint, the amount of data to be checkpointed significantly drops resulting in an overhead of 2.6%.

The rest of the paper is organized as follows. Section II presents the necessary background. In Section III we describe the extensions to support heterogeneous address spaces. Section IV analyzes and optimizes the checkpoint method. In Section V we evaluate our methodology, in Section VI we present the related work and in VII we conclude our work.

II. Background

In this section we provide a brief background on the tool and models we use in this work.

A. FTI implementation

FTI is a library that provides an API to the developer to efficiently perform multi-level checkpointing. It is implemented in C/MPI and it also provides a Fortran interface to the user. The developer uses library function calls to define which data need to be checkpointed as well as at which execution points a checkpoint can be taken. At execution time the library is controlled using a configuration file which defines several parameters. This allows the user to compile the application once and select different parameters prior to executing the application, for example select the file format of the C/R files. In Listing 1 we present a toy example of the use of FTI API.

To guarantee that the library will not cause any damage to the application communication channels, FTI has a function call, *FTI_Init()* (line 6) that will perform all the necessary actions before the application starts the computation. FTI_Init() will read the configuration file and once the configuration has been checked, FTI will detect in which node each process resides and will write this topology in a file. In case of a recovery, *FTI_Init* also checks whether the checkpoint files are correct. Finally, FTI creates two MPI communicators, one, called *FTI_COMM_WORLD*, which is used by the application processes for their internal communication and another communicator for coordinating the checkpoint procedure internally. Using the function *FTI_Protect* (line 12,13,14) the user can define a continuous memory region which will be stored in the C/R file upon the checkpoint procedure. The function can be called multiple times to protect different memory regions. After defining the memory regions the developer can call *FTI_Snapshot* (line 16) to instruct the library that a checkpoint can be taken. Whether a checkpoint will be actually taken depends on the user-specified checkpoint frequency defined in the configuration file. On recovery *FTI_Snapshot* will perform the actual recovery procedure. Finally, *FTI_Finalize* checks that all the pending checkpoints have finished and frees any FTI internal data structures.

```
1  int main(int argc, char *argv[]){
2    int rank, nbProcs;
3    double *h,*g;
4    int i;
5    MPI_Init(&argc, &argv);
6    FTI_Init(argv[1], MPI_COMM_WORLD);
7    MPI_Comm_size(FTI_COMM_WORLD, &nbProcs);
8    MPI_Comm_rank(FTI_COMM_WORLD, &rank);
9    h = (double*) malloc (sizeof(double)*nElements);
10   g = (double*) malloc (sizeof(double)*nElements);
11   initData(&h,&g);
12   FTI_Protect(0, &i, 1, FTI_INTG);
13   FTI_Protect(1, h, nElements, FTI_DBLE);
14   FTI_Protect(2, g, nElements, FTI_DBLE);
15   for(i = 0; i < N; i++){
16     FTI_Snapshot();
17     performComputations(h,g,i);
18   }
19   FTI_Finalize();
20   MPI_Finalize();
21 }
```

Listing 1: Source code using FTI. FTI API calls and variables are marked as red.

B. Model of Checkpoint Period and Overhead

The optimal checkpoint interval between two consecutive checkpoints can be estimated using Eq. 1, where C_{total} is the total time until a checkpoint is written to the destination storage device (SSD, Main Memory (DRAM) etc.), R is the cost of restoring the checkpoint, D is the downtime and μ is the mean time between failures [7].

$$T_{opt} = \sqrt{2(\mu - D - R)C_{total}} \tag{1}$$

The average time wasted for the checkpoint procedure can be computed using Eq. 2 as presented in [7]. $C_{overhead}$ corresponds to the average time an application is blocked for the checkpoint procedure. Noticeably, in the case of synchronous checkpoints[1] $C_{overhead} = C_{total}$, since the application process is also responsible to store the data to the stable storage. In the case of asynchronous checkpoint $C_{overhead} \neq C_{total}$ as typically a background process finalizes the writing procedure and the application process is released (unblocked) earlier.

$$\text{WASTE} = 1 - \left(1 - \frac{C_{overhead}}{T}\right)\left(1 - \frac{1}{\mu}\left(D + R + \frac{T}{2}\right)\right) \tag{2}$$

In FTI the recovery is divided into two separate phases, in the first phase, called *integrity phase*, which takes place during the *FTI_Init* call, FTI checks whether the checkpoint files are valid, by checking if the files exists and whether the integrity checksums are correct, if everything is correct it moves to the second phase, called *recovery phase*, in which FTI actually recovers the data to the memory. Consequently, in our case we will set $R = T_{integrity} + T_{recovery}$, where $T_{integrity}$ corresponds to the amount of time spend on the *integrity phase* and $T_{recovery}$ corresponds to the amount of time spend on the *recovery phase*.

[1]The application process is blocked until the checkpoint procedure finishes.

```
1  int main(int argc, char *argv[]){
2    int rank, nbProcs;
3    double *h,*g;
4    int i;
5    MPI_Init(&argc, &argv);
6    FTI_Init(argv[1], MPI_COMM_WORLD);
7    MPI_Comm_size(FTI_COMM_WORLD, &nbProcs);
8    MPI_Comm_rank(FTI_COMM_WORLD, &rank);
9    cudaMallocManaged(&h, sizeof(double)*nElements, flags);
10   cudaMalloc(&g, sizeof(double)*nElements);
11   initData(&h,&g);
12   FTI_Protect(0, &i, 1, FTI_INTG);
13   FTI_Protect(1, h, nElements, FTI_DBLE);
14   FTI_Protect(2, g, nElements, FTI_DBLE);
15   for(i = 0; i < N; i++){
16     FTI_Snapshot();
17     performComputations(h,g,i);
18   }
19   FTI_Finalize();
20   MPI_Finalize();
21 }
```

Listing 2: FTI API to support transparent GPU/CPU checkpoints. FTI API calls and variables are marked as red.

III. GPU Support for FTI

There exist several tools that allow GPU checkpointing. These tools, are not open source, do not perform multi-level checkpoint mechanisms and require specific driver support for the GPUs. In this Section we will describe the API implementation details to support GPU checkpointing.

A. FTI API

Most scientific applications do not divide the same workload among the GPU and the CPU in parallel. The computational power of a GPU device is magnitudes larger than the one of a CPU. Assigning a workload to execute simultaneously in both devices raises severe load balancing issues. Hence, applications are executed in different phases, each phase is executed by a specific device. GPUs are used for massively parallel computational intensive phases, whereas CPUs are used for non-parallelizable, communication heavy phases. Therefore, for every MPI-process, the application memory is distributed across the main memory (CPU) and the GPU memory.

The procedure of developing HPC applications for such heterogeneous systems can be tedious, even without taking into account the fault tolerance aspect. The developer needs to optimize the computations across the different device architectures and manage the data transfers to the respective nodes and to the respective devices within a node. Unfortunately though, applying C/R techniques on HPC applications is a necessity and commonly done through application level checkpoint libraries or implemented manually inside the application. To the best of our knowledge there is no open source checkpoint library that automatically handles data stored in the device memory. General C/R libraries instruct the developer to manually copy the data from the device memory to the host memory prior calling the respective API checkpoint function.

Our target is to provide a single API to support checkpoint of different memory regions regardless of their physical location. An example using the GPU/CPU checkpoint API is presented in Listing 2. Noticeably, despite allocating pointers using different devices/address spaces (lines 4,9,10) the FTI API calls remain exactly the same. To be more precise line 4 corresponds to DRAM, line 9 to UVA and line 10 to device memory. In *FTI_Protect* the developer specifies a single address which can be either a host-memory address, a device memory address or a UVM addres. The FTI library will handle accordingly each address type.

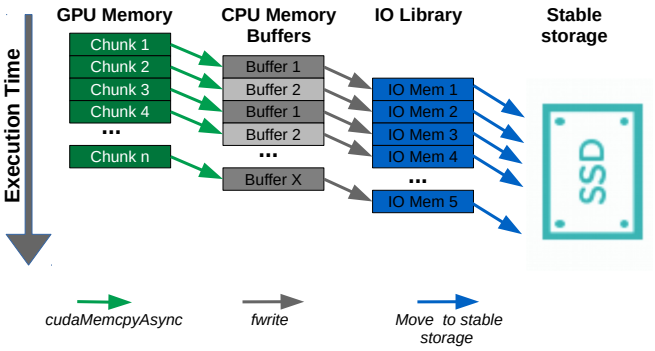


Fig. 1: The device to memory transfer protocol. Ideally, all the data movements are overlapped. The user-application is delayed until all data are copied to the stable storage.

B. FTI GPU/CPU implementation

To support Hybrid GPU/CPU support to FTI we extend the implementation of the *FTI_Protect* API call. In this function we identify the physical location of the data. This is done through the CUDA API support, namely the function *cudaPointerGetAttributes(&attributes,address)*. The function raises an error when called with a host address, whereas it returns normally with a device or an UVM address. In the second case we further check the values of *attributes* field which provide information whether the address is UVM or not. In the end we tag each address as *CPU,GPU,MANAGED*.

When the checkpoint phase takes place, depending on the tag of each address we perform a different action. In the case of CPU or UVM addresses, we invoke the normal FTI C/R procedure. In the case of UVM addresses the CUDA driver will fetch the data from their actual location and move them to the stable storage. Finally, in the case of *GPU* addresses, we overlap the writing of the file with the data movement from the GPU side to the CPU side. The procedure is depcited in Figure 1. This is done through streams and asynchronous memory copies of chunks from GPU memory to host pinned memory. Each protected GPU memory region is divided into chunks. In the beginning of the checkpoint, FTI processes request the first memory chunk from the CUDA driver and wait until it is transferred to the Host memory. Afterwards the FTI processes request the next chunk from the CUDA driver, and writes the current chunk to the stable storage. In reality though, the writing procedure just buffers the data to intermediate memory locations until they are written to the file. Besides writing the file, the FTI process also computes the integrity checksum of the current buffer. When both actions complete the FTI processes continue with the next chunk, the procedure terminates when all chunks are processed. In the case of recovery the implementation is the reverse. Namely, the reading operation is overlapped with moving the data to the GPU device in the case of *GPU* addresses.

IV. FTI Analysis and Optimizations

In this section we analyze and optimize the FTI GPU checkpoint scheme. The experiments are performed on a cluster composed of 16 nodes, where each node is equipped with 2 x IBM Power9 8335-GTG @ 3.00GHz (20 cores, 160 threads), 512GB DRAM, 2 x Micron 5100 Series 1.9TB SATA SSD, 2 x Samsung PM1725a 3.2TB NVMe SSD, 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2, Single Port Mellanox EDR and GPFS via one fiber link of 10 GBit.

We use two small micro-benchmarks for profiling and analysis purposes. The micro-benchmarks check the strong/weak scaling of our approach using different mixtures of device/host memory allocations. The first micro-benchmark allocates two memory buffers, the first buffer, called *hBuff*, is allocated on the host memory, whereas the second one, called *dBuff*, is allocated on the device memory. The size of each memory buffer is user defined. The application protects these two buffers. The second micro-benchmark is identical, but the device memory is allocated using a unified memory address space. We set the total checkpoint size to be equal to 48 Gb per node. We execute 3 different memory allocation schemes. In the first scheme, for each process we allocate 10% memory on the host and the remaining 90% to the device memory. The second scheme allocates 50-50% on the respective memories and the final scheme allocates 90-10%. Finally to analyze the checkpoint time we perform 5 checkpoints, whereas to analyze the recovery time we measure the time spent to recover from each one of these checkpoints (5 recoveries).

A. Analysis and Optimization of Checkpoint time

Figure 2 depicts the execution time spent (C_{total}) to perform a checkpoint when executing on 4 and 16 nodes for different memory allocation schemes using GPU memory or UVM memory. The Y-axis represents the execution time spent to perform a single checkpoint. The X-axis represents the allocation type of the micro-benchmark as well as the number of nodes used for this experiment. As expected, when increasing the number of nodes the checkpoint time does not increase, since we store the checkpoint on the Local storage NVMe device, and therefore the nodes do not share any resources during the checkpoint. Moreover we do not observe any significant differences when comparing UVM with non UVM executions or any differences depending on the memory distribution ratios. However the checkpoint time is quite high, since we need on average $C_{total} = 39s$ to checkpoint 48Gb of data for each node resulting to a per node bandwidth equal to 1.2Gb/s.

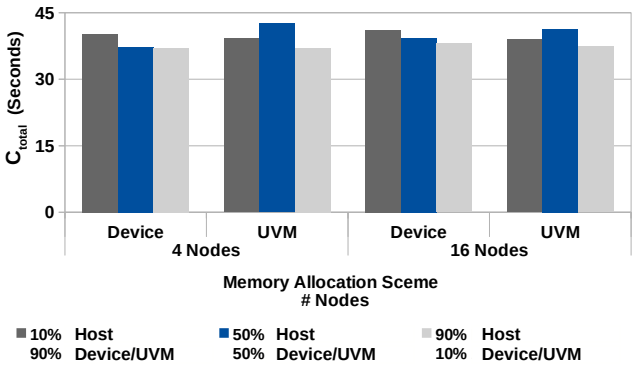


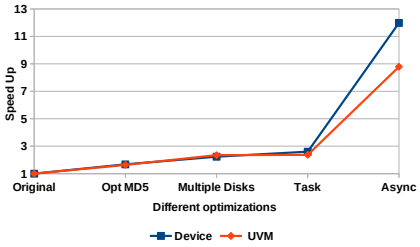
Fig. 2: Execution time spent to checkpoint 48Gb per node (weak scaling) for different memory allocation schemes using UVM memory or device memory.

To optimize the checkpoint procedure we perform an iterative approach, namely in each iteration we break-down the execution time spent on different actions performed during the checkpoint, depending on the analysis we try to eliminate the respective bottleneck.

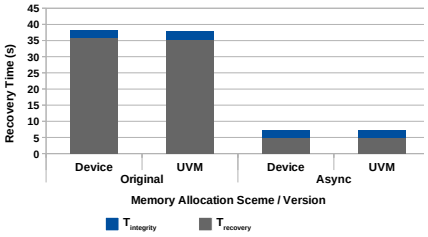
1) Optimization of Integrity checksum: In the initial version, a large portion of the checkpoint time is spent to compute the integrity checksum of the file, which will be used during recovery in the *recovery phase* to determine the correctness of the checkpoint file. So, we optimized the integrity checksum algorithm. FTI uses the MD5 algorithm [28] as an integrity checksum. The first step is to implement a GPU CUDA version of the MD5 algorithm presented in [16]. The input data is splitted into smaller chunks, cuda threads compute, in parrallel, the MD5 hash of each chunk, after computing the hash values the MD5 algorithm is applied again on the computed hashes. The procedure continues iteratively until a single hash is produced. The algorithm is also used for data stored the CPU memory as well for UVA memory addresses. In these cases we use a streaming implementation of the aforementioned algorithm, which uses streams to copy data to the GPU and compute the respective MD5 checksum. Applying the streaming algorithm to the UVA data chunks accommodates for cases in which the UVA memory does not fit entirely in the device memory. After this optimization $C_{total} = 30s$ achieving a bandwidth of $1.6Gb/s$ per node.

2) Multiple IO-Devices: In our cluster each node consists of 2 NVMe SSDs, therefore in each node half of the MPI-Processes can store their data on the one SSD and the other half on the other one. We implement this feature on FTI, in which, the user can define multiple local devices in the configuration file. The implementation assumes that the local devices have the same characteristics, and therefore it evenly distributes the processes to the IO devices. We profile once more our implementation and $C_{total} = 23s$ with a bandwidth of $2.0Gb/s$ per node.

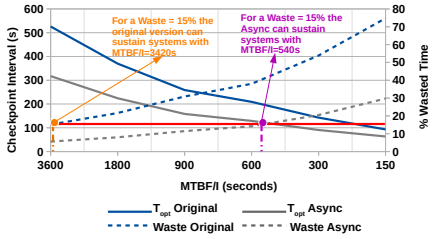
3) Task based checkpoint: During the checkpoint each process performs two tasks, the first one computes the integrity checksum and the second one writes the data. However, these tasks are completely independent. Both tasks, writing of the file and the MD5 computation, read only the data and do not change any value, therefore both of them can execute in parallel. We exploit this observation and we assign a task called *MD5 Task* to apply the MD5 computation on the data, this task uses the same context as the application, and a second task using a new shared memory context, called *Write Task* to write and synchronize the data with the stable storage. The tasks are executed in parallel. The total overhead of this scheme is equal to the maximum execution time of this



(a) Speed up the checkpoint procedure in respect to the original implement version



(b) Execution time to perform the different tasks during recovery for the initial and asynchronous implementation



(c) The T_{opt} on the left Y-axis for the initial and the final async version and the % $Waste$ of the two versions.

Fig. 3: Summary of profiling results from the normal checkpoint restart procedure.

procedure. Noticeably, both tasks transfer data simultaneously through the *NVLNK*. However, *MD5* task only stresses the NVLINK channel towards the device, as it copies data from the CPU to the GPU, whereas *Write Task* in the worst case scenario copies data from the device through another channel. Therefore these tasks are completely independent and do not share any hardware resources. Using this optimization the checkpoint bandwidth is on average equal to $2.8Gb/s$ per node ($C_{total} = 17s$). From our profiling we see that the execution time of the *MD5 Task* is smaller than the execution time of the *Write Task*, hence the bottleneck is writing and synchronizing with the stable storage.

4) Asynchronous checkpoint: During the profiling runs of the task-based checkpoint we observe that a significant amount of time (on average 78% of the checkpoint time) is wasted waiting for the data to be flushed on the NVMe SSD. During that time the integrity of the checkpoint file is already computed. Therefore, the *Write Task*, just before synchronizing any pending IO operations, it releases the application context (*MD5 Tasks*) to continue each execution. On the one hand, in an execution scenario in which the system has enough spare memory to store the checkpoint data into the main memory, the proposed procedure is similar to in memory checkpoint. When an application performs an IO write library call, for example in POSIX the *fwrite* function call, the memory residing in the application layer is copied internally in the Posix library buffers. Therefore, the data is not yet flushed to the stable storage device, but the application context is no more needed. Consequently we can release the application to continue the execution. On the other hand, in a system in which there is not enough available free memory, in contrast to in memory checkpoint which will not succeed to perform a checkpoint, our methodology will succeed. As the IO library driver will handle such cases accordingly by flushing some of the buffers to the stable storage to release the memory. This optimization does not decrease C_{total} of the execution but significantly decreases the overhead of the application $C_{overhead}$. In Figure 3a we present the speed-up of the checkpoint $C_{overhead}$ for the different optimizations in comparison with the initial implementation. When data is allocated using UVA memory the checkpoint procedure is a little slower, since the MD5 implementation for UVA data is slower than the one using GPU data as it uses streams to prefetch data to the GPU.

B. Analysis and Optimization of Recovery

To optimize the recovery procedure of an application we followed the same methodology as for the checkpoint procedure. We also applied the respective opimization for the recovery part. For brevity we present in Figure 3b only the execution time spent for the $T_{integrity}$ as well as for $T_{recover}$ of the original implementation and the final one, namely the asyncronous one. The majority of the speed up was due to the optimized version of the MD5 checksum algorithm. In total we present a 5.23x speed up of the total recovery time.

C. Optimal checkpoint Frequency and Wasted Time

In Figure 3c we extrapolate the optimal checkpoint frequency and the % wasted time of current/future systems. On the x-axis we use MTBF/I ranging from 60 minutes to 150 seconds, which covers MTBF/I of current/future exascale systems [9]. On the left Y-axis we present the checkpoint frequency and the right axis presents the % $Waste$. Althogh FTI supports resiliency for any type of failures, we set $D = 0$, consequently we extrapolate for systems that experience soft-failures. These failures typical occur due to transient parity errors on the L1 cache line. Although most components of the memory subsystem can correct single bit parity errors and detect double bit errors, the L1 cache can only detect single bit errors and is unable to correct them [12], [22]. Setting $D = 0$ takes only into account the overheads introduced by the C/R technique. When comparing the versions for the expected MTBF/I exascale system ($MTBF/I \approx 1hour$) [21] the optimized version can checkpoint every $T_{opt} = 5$ minutes and $WASTE = 5\%$ whereas for the initial version $T_{opt} = 8.7$ minutes with $WASTE = 15\%$. For the same amount of $WASTE = 15\%$, the asynchronous version can sustain execution in systems with 6.3 times smaller MTBF/I.

D. Differential Checkpoint

Up to this point we always considered storing all the user requested data to the checkpoint file. This is not always necessary. In essence during the checkpoint procedure one needs to store only the data that have changed their value from the previous checkpoint. This technique is called differential checkpoint [18], and can be implemented in two ways. The

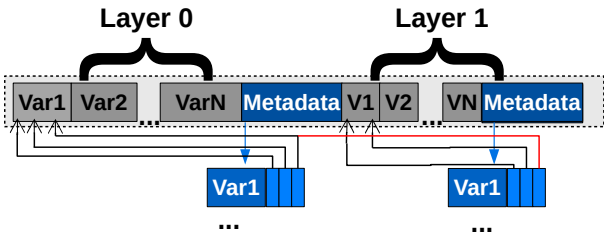


Fig. 4: File format of the differential checkpoint.

first one, uses the page dirty bits[2] of the Operating System (OS) to detect which pages has changed from the previous checkpoint. This technique is efficient, but not sufficient in the case of GPU applications, as the Operating system has no information about the contents of the GPU memory. Therefore, it is impossible to detect whether the data is dirty or clean. The second method, splits the data into chunks and uses hash values to encode the information of each chunk. During the checkpoint it compares the hashes of the data with the hashes of the checkpoint file. Should the hashes differ, the checkpoint library will store the data to the checkpoint file.

We opted for the second method, as we target heterogeneous applications that split their memory footprint across several memory locations. As presented in the previous sections we have implemented an efficient GPU version of the MD5 checksum algorithm. Therefore, there is no need to implement anything extra in terms of the hashing. However, we need to device a file format that stores the differences across consecutive checkpoints. Figure 4 presents the file format, the file is divided into layers. The first layer stores all the protected data. After all data is stored, there is a metadata region which contains information for each variable, such as the size of the variable, and where each chunk of the data is stored. The next layers append to the file only the data chunks that have changed their value and in the end of the new layer there is the metadata section which stores the updated location of the latest data in the checkpoint file. In the example, presented in Figure 4, *Var1* consists of three chunks, the first two have changed their values, wheres the third one (red line) has the same value. Therefore for the last chunk we point to the former location of the data. Since this procedure can infinitely increase the size of the checkpoint file, the user can bound the number of layers to a maximum layer size. When this size is reached we just create a new file and restart the procedure.

To evaluate the checkpoint efficiency of the differential checkpoint, we extend the previous micro-benchmarks to change a percentage of the contents of the protected variables before requesting a new checkpoint. In total we perform 20 checkpoints in which we change 20%, 50% and 80% of the checkpoint data. Finally we set the maximum layer size to 5, 10 and 15. Therefore, in the first case we perform a full checkpoint (1^st layer) and 4 differential ones, in the second case we perform 1 full checkpoint and 9 differential ones, and in the last 1 full checkpoint and 14 differential ones. Then the checkpoint procedure resets, by performing a full checkpoint.

In Figure 5a we present the time spent on the different actions when performing C/R using differential checkpoint. Setting the maximum layer size to small numbers, for example 5, reduces the potential gain of differential checkpoint. Since every 5 checkpoints one is a full checkpoint. However, on larger layer size values for example 15 we observe that C_{total} linearly scales down to the amount of data to be stored. On the other hand, we observe in all cases the $C_{overhead}$ of DCP to be higher than the $C_{overhead}$ of the normal checkpoint procedure regardless of the amount of data to be stored. The time spent on performing the comparisons of the checksums as well as the time spent on performing more IO-calls than the normal checkpoint procedure. In DCP each *write* call writes only $16Kb$ of data, therefore we store less data in total, but we make more IO-library calls.

In recovery we observe an increase on $T_{integrity}$ and $T_{recovery}$. During recovery we read randomly inside the file which is known to be slower than reading consecutive positions. Moreover, when recovering in differential checkpoint during the *recovery* phase, we compute once more the checksums of the data, so that in the next checkpoint we can perform a differential one. We could compute once all the checkpoint hashes during recovery this would require significant code changes in the initial FTI library, which are intrusive to the code structure. To favor the extensibility of the library we opted not to perform this optimization to provide a more intuitive code structure in the library. Finally, the recovery time does not increase as we recover from different layers.

Figure 5b depicts the optimal checkpoint frequency to checkpoint/recover $48Gb/Node$ of data for future systems with decreasing MTBF/I. The maximum number of layers is set equal to 15. When using dCP, even when 80% of the data needs to be written, the checkpoint frequency is decreased by 5%, regardless of the MTBF/I. When the changed data drops to 20% the checkpoint frequency is reduced by 30%. Figure 5c depicts the % *WASTE*. Interestingly, for dCP to present the same amount of wasted time the application requires to store 20% or less of the data, otherwise the dCP checkpoint versions presents higher $Waste$ values. However, the increase in $Waste$ reaches up to 5% when comparing the 80% of change with the *no DCP* version for an extreme case of $MTBF/I = 150$ seconds. This observation is a result of the increased latencies during the recovery procedure.

V. Evaluation

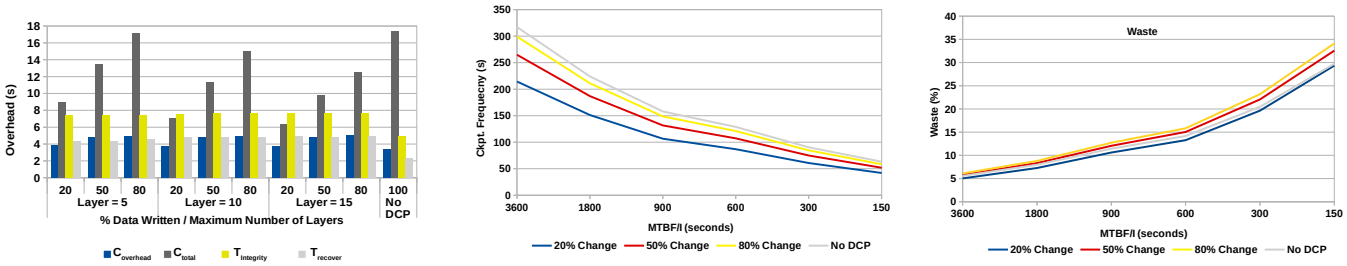
In Section IV we have analyzed and optimized our implementation using synthetic benchmarks. In this section we will use applications to evaluate different checkpoint scenarios of our multi-node/multi-GPU checkpoint scheme.

A. HPC applications

For our evaluation we used the following applications:

Heat2D is a 2D heat distribution simulation using a 1D domain decomposition. It simulates the transition from a non-equilibrium heat distribution to the equilibrium state. In each

[2]A memory page is considered as dirty when the contents of the page has been modified



(a) Time spent to perform different actions for different write percentages and different maximum number of layers.

(b) Optimal checkpoint frequency for executions with different % of written data.

(c) % Wasted time for different MTBF/I when C/R with the optimal T_{opt} executions with different % of written data.

Fig. 5: Summary of profiling results from the differential checkpoint restart procedure.

time step, the cells of the temperature grid are updated via a 4-point stencil operation that stores the average of the 4 neighbor cells temperatures into the center cell. We use a GPU-kernel that performs the grid update, each GPU-thread is responsible for one cell, to reduce the read operations on the GPU global memory, each thread loads the values to the shared memory and afterwards it performs the stencil operation.

Jacobi solver is a real world example that iteratively solves the Poisson equation on a rectangle with Dirichlet boundary conditions. The algorithm uses second-order central differences to approximate the Laplacian operator on the discrete grid. The *Jacobi* kernel applies a 4-point stencil operation to store the average value of the 4 neighbouring cells. To use multiple GPUs in multiple nodes the applications applies a 2D domain decomposition with $n \times k$ domains.

Hydro is a multi-node multi GPU benchmark [6] which implements a 2D Eulerian scheme using a Godunov method [13]. The space domain is a rectangular two-dimensional splitting with a regular cartesian mesh, the code solves compressible Euler equations of hydrodynamics, based on a finite volume numerical method using a second order Godunov scheme. A Riemann solver [29] computes numerical flux at the interface of two neighbouring computational cells.

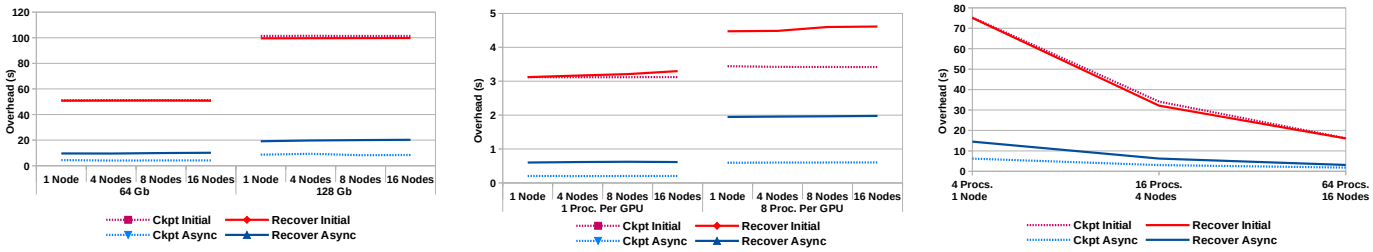
B. Experimental Results

In the evaluation we present only the overhead of the application process, as the asynchronous tasks present minimum extra overhead.

Since, none of the GPU application level checkpoint mechanisms presented in Section VI is publicly available, we test the 2 different methods implemented in this work. The *initial* corresponds to the implementation without any optimization presented and the one that includes all the optimizations is called *Async*. For each experiment we present the amount of time in seconds the application process spends to checkpoint or recover from a checkpoint file

1) ***Checkpoint UVM Address Space****:* We use *Head2D* to test the behavior of our multi-gpu/multi-node checkpoint methodology when the application is using UVM memory allocations. We checkpoint *Heat2D* for two different problem sizes, namely in the first problem we checkpoint 16Gb per process whereas in the second we checkpoint 32Gb per-process. The first problem size barely fits in the GPU main memory at once, the second problem size does not fit at once in the GPU main memory. In both cases, the CUDA-driver is responsible to transfer and manage the UVM allocated data. We test different number of nodes, in each node we execute 4 processes, one per GPU device, therefore the GPU devices are not shared among the processes. Finally, the problem size is weakly scaled as the number of nodes increases. When we use 16 nodes the total size of the problem size and thus the total size of the checkpointed data is equal to 1Tb and 2Tb respectively. Figure 6a depicts the results of our experiments for the different methods. The *x-axis* corresponds to the different problem sizes and the different node configurations, whereas the *y-axis* corresponds to the $C_{overhead}$ and to $R_{integrity} + R_{recovery}$ for the checkpoint and the recovery procedure respectively . As expected, the checkpoint overhead does not increase as we increase the number of nodes for the two different problem sizes, since each node stores results on the local NVMes. The overhead decreases as we apply our optimized methods. Namely when we compare the initial version with the async version we obtain resepctively a $12.05X$ and $5.13X$ reduction in the $C_{overhead}$, and to the $R_{integrity}+R_{recovery}$. The same amount of reduction is observed in both problem sizes, consequently our implementation strongly scales.

2) ***Shared GPU among Processes****:* In *Jacobi* we test the behavior of our implementation, when multiple processes share the same GPU device. We execute two different sets of experiments, in the first set there is a 1:1 ratio between the number of processes to the number of GPUs. In other words, each GPU is used only by a specific CPU. On the second set we use a 8:1 ratio, 8 processes share the same GPU. Once more, we evaluate the weak scaling of our method, each application rank solves a local domain size of 8192×8192 elements which corresponds to per process checkpoint size of 1Gb. We execute both ratio configuration on different number of nodes. The results are depicted in Figure 6b. The *Y-axis* presents the overhead in seconds, whereas the *X axis* represents, the different node/GPU per process configurations. Recall, that each node consists of 4 GPUs, therefore on the 1:1 ratio we execute 4 user processes per node, whereas on the 8:1 configuration we execute 32 processes per node. When we compare the different versions with the initial one, for the



(a) Execution time spent to C/R *Heat2D*. (b) Execution time spent to C/R Jacobi. (c) Execution time spent to C/R Hydro.

Fig. 6: Summary of the evaluation using the different applications.

1:1 CPU to GPU ratio, the speed up of the async version executes $15.23x$ and $5.21x$ times faster for the checkpoint and recovery procedure respectively. In the case of the 8:1 ratio the speed up is dropped to $5.3x$ and $2.68x$ for the two versions respectively. The slow down in comparison with the 1:1 version is due to the latency of the I/O layer and not due to the latency of the transfers from the GPU device to the CPU device, since multiple processes share the same device. The processes among the same layer share the I/O layer, increasing the amount of data stored in this layer should also increase the execution time of the procedure. Finally, since almost all the checkpoint data are stored in the GPU side, the MD5 GPU algorithm is executed without any intermediate copy steps during the checkpoint, and therefore, the computation is completely offloaded asynchronously to the GPU device. This case corresponds to the maximum observed speed up in the case of checkpoint overhead.

*3) **Strong Scaling of multi-GPU/multi-node checkpoint**:* In the case of *Hydro* we test the strong scaling of our method. In *Hydro* the data are almost evenly distributed among the GPU memory and the CPU. Data in the GPU device do not use UVM address space, consequently they are not accessible directly from the Host. Figure 6c depicts the results of our experiments. When comparing the *initial* version with the *async* one, the checkpoint overhead is reduced by a factor of 11x, regardless of the node configuration. However, in contrast to *Jacobi* we do not observe the same speedup. *Hydro* checkpoints 50 different memory regions, each one with a relative small size, however the larger gains of the MD5 checksum algorithm are obtained when performing the GPU-MD5 algorithm to big continuous memory chunks, and not to many smaller ones. In terms of recovery the ocontinuous ptimized version is $5.16X$ times faster than the original one.

C. Differential checkpoint evaluation

To evaluate differential checkpoint we compare it against executions which do not perform checkpoint and against which perform normal checkpoints. We execute the three applications for 20 minutes on 16 Nodes with 16 processes per node, the checkpoint size depends on the application itself. We perform a checkpoint every $30s$ of pure application execution, therefore the checkpoint frequency is equal to $T_{overhead} + 30s$. As a Figure Of Merit (FOM) we use the metric iterations per second. The larger the value the more efficient the checkpoint procedure is. In Table I we present our evaluation.

Application	Version	Total Ckpt Size	% Diff	Iter/Sec	Num Ckpts.
Heatdis	No Ckpt	0	0	10.37	0
	Normal	768Gb	100	7.36	31
	Differential	768Gb	29.8	7.85	33
Jacobi	No Ckpt	0	0	5.94	0
	Normal	768Gb	100	4.09	27
	Differential	768Gb	10.4	4.70	31
Hydro	No Ckpt	0	0	43.7	0
	Normal	1.8Tb	100	38.08	34
	Differential	1.8Tb	3.4	42.55	38

TABLE I: Evaluation of our differential checkpoint approach for the tree applications with realistic checkpoint sets.

Interestingly, on all applications we observe a significant reduction to the amount of data to be stored, namely the data to be stored range from 3.4% to 29.8% for the three applications. The reduction of the data to be stored leads into two interesting points, firstly differential checkpoint introduces lower checkpoint overhead, since we observe larger values of Iter/Sec, namely the overhead ranges from 2% to 24% whereas the normal checkpoint overhead ranges between 13% to 31%. Secondly, in these experiments the differential checkpoint leads into performing more frequent checkpoints, to be more precise in Heatdis differential checkpoint performs 2 more checkpoints than the normal procedure while in *Jacobi* and *Hydro* differential checkpoint performs 4 extra checkpoints. In other words, differential checkpoint is able to perform checkpoints with higher frequency and lower overhead. In terms of the recovery time differential checkpoint is 1.7 times slower than then normal recovery time. The premise of this performance reduction is discussed in section IV-D. In the future we will investigate how we can hide such latencies.

VI. Related Work

HPC systems typically employ C/R for fault tolerance [8], in which, at frequent intervals, the application state is stored in a non-volatile storage device. The frequency depends on the MTBF of the execution environment. Checkpoints can be either application transparent or application initiated.

Transparent checkpoint methods [11], [15] do not require source code modifications, instead the entire memory state is stored, therefore, the size of the checkpoint file can proportionally increase. Application coordinated checkpoint methods [2] require the application to define when to checkpoint and

which data to checkpoint. A number of optimizations have been proposed to reduce the amount of data to be stored. Differential approaches [10], [11], [18] update only the data that has changed in comparison to the previous checkpoint and are beneficial only when applications do not change substantially. In contrast to these works, our work focuses on the programmability aspect of application initiated multi-level C/R and optimizes the normal checkpoint procedure. Moreover, we present an implementation of a differential checkpoint for GPU applications and take into account the overheads of the checkpoint procedure as well as the overheads of the recovery procedure.

Fault tolerance for GPU computing is a frequently discussed research topic, with HiAL-Ckpt [33] being one of the first attempts to checkpoint GPU applications. HiAL-Ckpt is a C/R mechanism based on the Brook+ programming language and allows the programmer to do checkpoints at the application level. CheCUDA [32] blends a CUDA aware checkpoint mechanism with Berkeley Lab C/R (BLCR) mechanism [15]. CheCUDA copies the GPU context from the GPU memory to the host memory, so that the GPU can restart from the saved status. NVCR [23] uses a similar protocol as CheCUDA, but they present a transparent CR library, which allows C/R without recompiling the application. Both CheCUDA and NVCR save all CUDA resources during checkpoint and restart, therefore the latency required by data transfers between the GPU and CPU significantly slows down the GPU performance. In [19] the authors combine C/R mechanisms, virtualisation and CUDA streams to optimize the checkpoint overhead. HeteroCheckpoint [17] presents a CPU-GPU checkpointing mechanism using non-volatile memory (NVM). CUDA streams, pre-copy and checksum methods are used to enable parallel data movement which reduce the checkpoint data movement cost. The authors in [35] propose an application-level C/R scheme to save and restore GPU computation states. The work addresses the complex memory hierarchy of GPU devices and utilizes secondary storage for scalability and long-term fault tolerance. None of these works are open source and none of them consider the distribution of data among different memory devices nor the overhead of the integrity checksum of the checkpoint procedure. We provide publicly available in Github [14] a single API to checkpoint data among different memories, and reduce the overhead by exploiting several compute capabilities of our system.

Other C/R methods focus on the OpenCL programming model as it provides portability across different device families. Similar to CheCUDA, CheCL [31] is the mechanism based on OpenCL and BLCR. CheCL synchronizes the host and the command queues by waiting for all commands to complete and add an extra step of inter-process communication which incurres extra communication latency. A portable checkpoint solution for OpenCL applications is presented in [20]. The authors combine a host-side application level checkpoint tool and an openCL library. VOCL-FT [24] uses ECC error checking, logging of OpenCL inputs and checkpointing for correcting those ECC errors in the device memory. It is used to detect soft errors, which not always results into a fail-stop error but can also cause data corruption.

There are other works which focus on heterogeneous checkpoints in Many Integrated Core (MIC) systems. MIC-Check [26] outlines and analyzes the intrinsic and extrinsic issues that limit the I/O performance of MIC when checkpoint parallel MPI applications. Snapify [27] introduces a set of extensions that provide C/R features for MIC offload applications. This work relies on BLCR for checkpoint and consumes a lot of saving space and leads to extra transferring via PCIe. The authors in [4] perform in-memory C/R to strengthen the fault tolerance of a CPU-MIC system. The provided functionality is improved in [5] in which the checkpoint procedure is executed completely asynchronously.

In contrast to the related work, our implementation allows checkpoint of memory regions which are distributed in the host memory as well as in the device memory. It supports highly optimized efficient classic as well as differential checkpoints. We provide a single API to support all these memory locations, without needing any hints from the programmer. To the best of our knowledge we are the first open-source library that supports multi-node/GPU and multi-level checkpoints.

VII. Conclusions

In this paper we extended a checkpoint library called FTI, to support checkpoint of data stored in GPU and CPU memory locations. Our approach provides an identical API for memory addresses using normal (Host), device or UVM addresses. This feature reduces the programmers effort as it only needs to define which memory regions need to be checkpointed.

During checkpoint we used streams and we overlapped the device to host data transfer, with the writing of the data to reliable storage. We analyzed our approach using micro-benchmarks, and while the data transfers are completely overlapped, a significant amount of time is spent on computing the checkpoint checksum code. Consequently, we implemented a parallel CUDA version of the MD5 checksum algorithm to reduce the execution time spent on computing the integrity code. Finally, the application is released the moment we have performed all the necessary actions with the user specified data. On the background, we finalize the checkpoint procedure while the application continuous the normal execution. All this optimization steps reduce the checkpoint and recovery time by up to 15.23X and 5.21X respectively.

To further reduce the checkpoint overhead we implement an efficient differential checkpoint methodology within FTI. The method capitalizes on the efficient MD5 checksum GPU implementation to identify which memory chunks have changed their value in comparison with the value stored in the previous checkpoint file, and writes only the changed data. By doing so, we are able to effectively reduce the amount of data stored during the checkpoint procedure. The differential checkpoint is able to perform more frequent checkpoints with a reduced overhead in comparison with the normal checkpoint procedure. However, the cost of recovering from differential checkpoints is increased, as we need to read small data chunks from

random file locations, which is inefficient in comparison with reading from a file big continuous data chunks.

VIII. Acknowledgements

The research leading to these results has received funding from the European Union's Horizon 2020 Programme under the LEGaTO Project (www.legato-project.eu), grant agreement #780681.

References

[1] A. M. Aji, L. S. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. R. Bisset, J. Dinan, W. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur. Mpi-acc: Accelerator-aware mpi for scientific applications. *IEEE Transactions on Parallel and Distributed Systems*, 27, 2016.

[2] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[3] Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications*, 2009.

[4] Cheng Chen, Yunfei Du, Zhen Xu, and Canqun Yang. Ft-offload: A scalable fault-tolerance programing model on mic cluster. In *Algorithms and Architectures for Parallel Processing*, 2015.

[5] Cheng Chen, Yunfei Du, Ke Zuo, Jianbin Fang, and Canqun Yang. Toward fault-tolerant hybrid programming over large-scale heterogeneous clusters via checkpointing/restart optimization. *The Journal of Supercomputing*, 2017.

[6] Guillaume Colin de Verdire. Hydro benchmark. Technical report.

[7] Jack Dongarra, Thomas Herault, and Yves Robert. Fault Tolerance Techniques for High-Performance Computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer, 2015.

[8] E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B Johnson. A survey of rollback-recovery protocols in message-passing system. 2002.

[9] Kurt Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[10] Kurt B. Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. libhashckpt: Hash-based incremental checkpointing using gpu's. In *Recent Advances in the Message Passing Interface*, 2011.

[11] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[12] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz. Extending stability beyond cpu millennium: a micron-scale atomistic simulation of kelvin-helmholtz instability. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[13] S. K. Godunov. A difference scheme for numerical solution of discontinuous solution of hydrodynamic equations. *Math. Sbornik*, 47:271306, 1959.

[14] Leonardo Bautista Gomez. FTI. https://github.com/leobago/fti.git, 2014.

[15] Paul Hargrove and Jason Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. Technical report, Computational Research Division Ernest Orlando Lawrence Berkeley National.

[16] G. Hu, J. Ma, and B. Huang. High throughput implementation of md5 algorithm on gpu. In *Proceedings of the 4th International Conference on Ubiquitous Information Technologies Applications*, 2009.

[17] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan. Heterocheckpoint: Efficient checkpointing for accelerator-based systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[18] Kai Keller and Leonardo Bautista-Gomez. Application-level differential checkpointing for hpc applications with dynamic datasets. In *CCGrid 19: Proceedings of 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2019.

[19] Supada Laosooksathit, Nichamon , Naksinehaboon , Chokchai Leangsuksan, Apurba , Dhungana , Clayton Chandler, K Chanchio, and Amir Farbin. Lightweight checkpoint mechanism and modeling in gpgpu environment. 01 2010.

[20] Nuria Losada, Basilio B. Fraguela, Patricia Gonzlez, and Mara J. Martn. A portable and adaptable fault tolerance solution for heterogeneous applications. *Journal of Parallel and Distributed Computing*, 2017.

[21] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, Al Geist, Rud Haring, Jeffrey Hittinger, Adolfy Hoisie, Dean Micron Klein, Peter Kogge, Richard Lethin, Vivek Sarkar, Robert Schreiber, John Shalf, Thomas Sterling, Rick Stevens, Jon Bashor, Ron Brightwell, Paul Coteus, Erik Debenedictus, Jon Hiller, K. H. Kim, Harper Langston, Richard Micron Murphy, Clayton Webster, Stefan Wild, Gary Grider, Rob Ross, Sven Leyffer, and James Laros III. Doe advanced scientific computing advisory subcommittee (ascac) report: Top ten exascale research challenges. 2014.

[22] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[23] A. Nukada, H. Takizawa, and S. Matsuoka. Nvcr: A transparent checkpoint-restart library for nvidia cuda. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011.

[24] Antonio J. Peña, Wesley Bland, and Pavan Balaji. Vocl-ft: Introducing techniques for efficient soft error coprocessor recovery. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[25] B. Pourghassemi and A. Chandramowlishwaran. cudacr: An in-kernel application-level checkpoint/restart scheme for cuda-enabled gpus. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[26] Raghunath Rajachandrasekar, Sreeram Potluri, Akshay Venkatesh, Khaled Hamidouche, Md. Wasi-ur Rahman, and Dhabaleswar K. (DK) Panda. Mic-check: A distributed check pointing framework for the intel many integrated cores architecture. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[27] Arash Rezaei, Giuseppe Coviello, Cheng-Hong Li, Srimat Chakradhar, and Frank Mueller. Snapify: Capturing snapshots of offload applications on xeon phi manycore processors. *HPDC 2014 - Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, 2014.

[28] R. Rivest. "the md5 algorithm". Technical report, April 1992.

[29] P.L Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 1981.

[30] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 2015.

[31] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. Checl: Transparent checkpointing and process migration of opencl applications. In *2011 IEEE International Parallel Distributed Processing Symposium*, 2011.

[32] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. Checuda: A checkpoint/restart tool for cuda applications. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009.

[33] X. Xu, Y. Lin, T. Tang, and Y. Lin. Hial-ckpt: A hierarchical application-level checkpointing for cpu-gpu hybrid systems. In *2010 5th International Conference on Computer Science Education*, 2010.

[34] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *2011 IEEE International Parallel Distributed Processing Symposium*, 2011.

[35] Yulu Zhang, Xinyuan Guo, Hai Jiang, and Kuan-Ching Li. A checkpoint/restart scheme for cuda applications with complex memory hierarchy. In *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2013.