VOCL-FT: Introducing Techniques for Efficient Soft Error Coprocessor Recovery

Antonio J. Peña
Argonne National Laboratory
Argonne, IL 60439
apenya@mcs.anl.gov

Wesley Bland
Argonne National Laboratory
Argonne, IL 60439
wbland@anl.gov

Pavan Balaji
Argonne National Laboratory
Argonne, IL 60439
balaji@anl.gov

ABSTRACT

Popular accelerator programming models rely on offloading computation operations and their corresponding data transfers to the coprocessors, leveraging synchronization points where needed. In this paper we identify and explore how such a programming model enables optimization opportunities not utilized in traditional checkpoint/restart systems, and we analyze them as the building blocks for an efficient fault-tolerant system for accelerators. Although we leverage our techniques to protect from detected but uncorrected ECC errors in the device memory in OpenCL-accelerated applications, coprocessor reliability solutions based on different error detectors and similar API semantics can directly adopt the techniques we propose. Adding error detection and protection involves a tradeoff between runtime overhead and recovery time. Although optimal configurations depend on the particular application, the length of the run, the error rate, and the temporary storage speed, our test cases reveal a good balance with significantly reduced runtime overheads.

CCS Concepts

•**Software and its engineering** → *Runtime environments;*

Keywords

Co-processor; ECC; Fault Tolerance; Virtualization; VOCL

1. INTRODUCTION

High-performance computing (HPC) has seen many trends in the past decade. One is the rise of coprocessors to improve computing power while consuming less energy [4, 12]. On the November 2014 TOP500 list [23], 75 machines were using accelerators, whether GPUs or specialized processors, such as the Intel® Xeon Phi™ coprocessor. These accelerators provide an energy-efficient way to perform specific types of computations quickly.

Another recent trend has been the increased concern over fault tolerance at large scales. Machine sizes have increased to hundreds of thousands of cores; and along with the size increase, the reliability of the machines has decreased. Researchers have investigated mitigating various types of failures across many components of the machines. Traditional research has focused on protecting against two types of failures: fail-stop failures and soft errors. With a fail-stop failure, some error causes an execution to halt and requires restarting the job, usually from a checkpoint taken during the execution [7, 22]. Research has also demonstrated ways to prevent an application from needing to be restarted, instead allowing the system to recover on the fly [1, 2]. A soft error usually does not cause a fail-stop error but instead causes a data corruption. Soft errors are harder to detect than fail-stop errors and require more sophisticated mechanisms to find the errors and repair the data. They might be detected by hardware, such as error-correcting codes (ECCs) [15, 29], or they might require the application itself to detect such corruptions [20].

Many soft errors in coprocessor-equipped supercomputers originate solely in the memory of the accelerator devices. For example, analyzing the publicly available error log of the Japanese Tsubame 2.5 supercomputer [26], we find that almost 5% of the error entries involved double-bit ECC errors in GPU memory since late 2010, with an average count of one error every two weeks. The Moonlight supercomputer at LANL was reported to experience an average of 0.8 of these events per day during a 4-month period [6]. Although accelerators sharing the memory system with the main host processors have been developed, deploying memory technologies specialized for the particular access patterns of embarrassingly parallel architectures (such as GDDR) is highly beneficial for the HPC arena.

Current accelerators equipped with ECC hardware in their own memory systems automatically correct single-bit flips, but only report double-bit failures. The uncorrected error events translate to applications merely in error codes returned by the coprocessor application programming interface (API) functions, which in practice means that these are ignored or, in the best case, the execution is aborted. With applications, such as physical simulators, running continuously for several days, an automatic mechanism to ensure accelerator data sanity is clearly beneficial, with no disparagement of a host's main memory integrity guardian. Ideally, the coprocessor runtime could take the responsibility of efficiently detecting and recovering from transient soft errors without the application's intervention.

The time spent on soft error recovery ranges typically from a few minutes to a few hours in case the application leverages

SC '15, November 15 - 20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807640

accelerator-aware checkpointing. However, current solutions are impractical for many use cases. For example, in our experiments we find over 1,000% runtime overhead on the baseline protection of MiniMD runs. Inefficiently protecting from soft errors leads to longer execution times and a less efficient utilization of the system resources.

In this paper we introduce novel techniques for efficiently protecting data residing on coprocessor memories, based on API interception. We use the Virtual OpenCL (VOCL) library [30] as a framework for constructing coprocessor resilience. This library abstracts the concept of physical devices by providing a virtualization layer between the application and the accelerators themselves. The calls to OpenCL are captured by VOCL and forwarded to the physical coprocessor by the library. This approach enables techniques such as remote execution, replication, and execution replay.

We have extended the existing VOCL library as a way to log input and commands to OpenCL to be replayed later in the event of a failure. We call this extended library VOCL-FT. Currently, VOCL-FT focuses on soft error detection by means of ECC query. It can detect uncorrected bit-flips (usually double-bit corruptions) in devices equipped with ECC detection and allow the runtime to react by restoring the corrupted memory and re-executing any potentially affected commands in order to generate a correct answer. Our techniques, however, are independent from the error detection mechanism and can be integrated in other solutions, such as those based in application-level data sanity verifications or coprocessor migration after hard errors, in order to efficiently restore coprocessors' memory.

Traditional checkpoint/restart techniques are expensive because large amounts of data have to be backed up either in main memory or on slow storage systems. In this paper we propose and analyze techniques to address this limitation by identifying and taking advantage of the opportunities brought by the semantics of current accelerator programming models. As a result, incurring low runtime overhead, VOCL-FT can recover efficiently from uncorrected ECC errors by restoring only the meaningful portions of the accelerator memory and replaying the affected epoch commands.

In summary, the contributions of this paper are threefold: (1) a set of semantic and implementation optimizations and (2) a small extension to the OpenCL API to enable further optimizations that (3) along with ECC error checking integration builds an efficient fault detection and recovery runtime system for OpenCL devices. These contributions enable a transparent low-overhead fault recovery mechanism for OpenCL applications, while offering the possibility of lowering the overhead further when leveraging minor code modifications. To the best of our knowledge, this is the first work exploring the semantic optimization opportunities of current accelerator programming models for providing efficient resilience capabilities.

The remainder of this paper is organized as follows. Section 2 introduces background information. Section 3 discusses related work. Section 4 describes the design and implementation of the initial version of VOCL-FT. Section 5 discusses the data protection optimizations we devised. Section 6 provides an in-depth analysis of the performance of our proposed techniques, and Section 7 investigates the performance of VOCL-FT in production environments at scale. Section 8 summarizes the conclusions from our work.

2. BACKGROUND

In this section we provide background information about OpenCL and VOCL.

2.1 OpenCL

The Open Computing Language (OpenCL) [14] is designed to facilitate program execution across a variety of devices including CPUs, GPUs, and FPGAs. It has been adopted by several vendors, including AMD, NVIDIA, and Apple, to integrate with their products. OpenCL has been used primarily as a way of interfacing with GPUs for general-purpose computing, but recently it has been employed by other accelerators, such as the Intel® Xeon Phi™.

OpenCL applications are written by using a series of code kernels and data movement commands submitted from the host to the OpenCL device via *command queues*. Memory in OpenCL contexts is allocated by the `clCreateBuffer` call. This function accepts flags to specify lifetime usage information, including write-only, read-only, and read–write access from kernels. On the other hand, command queues may be ordered or unordered, and commands may be synchronous or asynchronous. In order to determine when asynchronous commands have finalized, and hence be able to reuse buffers or employ the requested data, applications may issue synchronization calls (such as `clFinish` or `clWaitForEvents`) or poll on completion calls (such as `clGetEventInfo`). OpenCL *epochs* contain commands issued between synchronizing calls. In this regard, an epoch can be considered to always be closed by a call to `clFinish` or blocking calls, but closed by the other cited functions only in ordered queues when these target the last command in the queue. To illustrate the use of the OpenCL API, Figure 1 shows in pseudocode the steps required to code a simple OpenCL program that will execute a kernel using a single I/O argument. More detailed information can be found in [14].

2.2 VOCL

VOCL [30] is a framework providing a transparent virtualization layer between applications and the OpenCL runtime. Apart from executing on the local device, the VOCL client library can intercept and forward OpenCL commands to off-node VOCL proxies driving remote accelerators. This strategy enables applications to increase the number of coprocessor resources to which they have access by harnessing remotely accessible devices. In this paper we do not use the remote feature, but we benefit from VOCL's virtualization infrastructure to implement transparent fault tolerance capabilities in the OpenCL stack. Exploring the integration of our extensions with remote accelerators, for example to perform device migrations, is left for future work.

3. RELATED WORK

To protect application data residing in main memory from eventual errors, the standard mechanism that has been used for decades is checkpointing. The most popular library has been BLCR [7] and is usually employed to checkpoint an entire process state at the operating system (OS) level. Another example is the LAM/MPI Framework [22]. Recently, more lightweight checkpointing options have arisen that promote smaller checkpoints that can be taken with greater frequency because of their low overhead. These include projects such as MPICH-V [3], which combined lightweight

```
// Initialization: obtain/select platform/device IDs,
//   create contexts, command queues, device memory
clGetPlatformIDs(..., &platform_id, ...);
clGetDeviceIDs(..., platform_id, ..., &dev_id, ...);
context = clCreateContext(..., dev_id, ...);
cq = clCreateCommandQueue(context, ...);
dev_mem_obj = clCreateBuffer(context,
                             CL_MEM_READ_WRITE, ...);

// Create and build a program and create a kernel
program = clCreateProgramWithSource(context, ...);
clBuildProgram(program, ..., SOURCE_STR, ...);
kernel = clCreateKernel(program, KERNEL_NAME, ...);

// Set a kernel argument: the OpenCL memory object
clSetKernelArg(kernel, 0, sizeof(cl_mem),
               (void *)&dev_mem_obj);

// Enqueue the write (input), kernel (execution),
// and read (output) operations
clEnqueueWriteBuffer(cq, dev_mem_obj,
                     /* blocking? */ CL_FALSE,
                     ..., SIZE, host_buf, ...);
clEnqueueTask(cq, kernel, ...);
clEnqueueReadBuffer(cq, dev_mem_obj,
                    /* blocking? */ CL_FALSE,
                    ..., SIZE, host_buf, ...);

// Async. command execution; perform other tasks here

clFinish(cq);  // Synchronization

// 'host_buf' has finished receiving the output data

// Finalization: release the created resources
clReleaseKernel(kernel); clReleaseProgram(program);
clReleaseMemObject(memobj); clReleaseCommandQueue(cq);
clReleaseContext(context);
```

Figure 1: OpenCL sample (pseudocode).

checkpointing with message logging. Other new advances include Containment Domains [5], which allow the application to prevent errors in one part of the system from affecting others. Our work is in line with this last concept, presenting a heavily optimized solution for recovery on coprocessors.

VOCL is not the only virtualization layer for accelerator libraries. For instance, rCUDA [17, 19] or SnuCL [11] provide similar functionality. These solutions do not include any additional reliability features but could well be extended to incorporate the techniques that we explore in our work.

Besides VOCL-FT, other attempts have been made to create a resilience layer for GPU computing. DS-CUDA [10] performs redundant computations in multiple GPUs. Similarly, [28] leverages redundant multithreading targeted at systems without ECC hardware. An implementation of silent data corruption (SDC) protection used on GPUs is Hauberk [31], a source-to-source translator that inserts SDC detectors into the application code. When it detects a failure, it automatically restarts the GPU application from the beginning of its execution or from a checkpoint (depending on the configuration). Snapify [21] provides a snapshot service (checkpoint/restart, process migration, and process swapping) specifically for Intel® Xeon Phi™ coprocessors.

CheCUDA [25] and CheCL [24] are libraries that allow an application including calls to CUDA and OpenCL, respectively, to be checkpointed. This process is done differently for each API. For CheCUDA, the authors created a package as a BLCR add-on to manage checkpointing that copies all the data out of the GPU memory at checkpoint time and restores it on restart. For CheCL, a simple transparent checkpoint-restart mechanism is deployed. In contrast

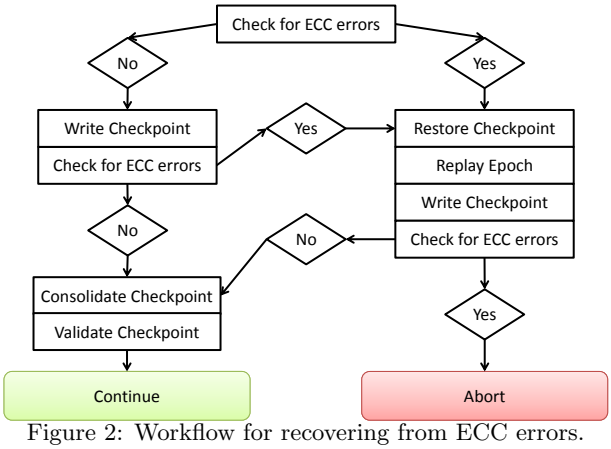


Figure 2: Workflow for recovering from ECC errors.

with VOCL-FT, these solutions do not specifically target errors in the coprocessor, nor are they integrated with any accelerator fault detection system.

All the discussed solutions performing accelerator checkpointing save and restore the entire device memory (as we do in our initial approach and present for comparison). Our work focuses on exploring the optimization opportunities that the semantics of accelerator-offloading programming models offer to attain lightweight checkpointing and fast recovery. Although we showcase them with a double-bit error detector, these could well be directly adopted by several other solutions, such as CheCUDA or CheCL, obtaining the performance benefits that we present in our study.

4. DESIGN AND IMPLEMENTATION

In this section we discuss the overall design of VOCL-FT. We describe the mechanisms for logging OpenCL commands, failure detection, and recovery. While our logging procedure acts upon the interception of a set of OpenCL calls, our detection and recovery mechanism (see Figure 2) is triggered during synchronizing calls. By checking for correctness only at epoch closings we minimize the detection overhead while ensuring the prevention of the propagation of coprocessor data corruptions to the host memory, since only after these points the data retrieved from the device is guaranteed to be consistent in the host memory.

We note that our solution is explicitly designed for separate accelerator memories. Users should deploy OpenCL-aware solutions for host memory protection. Since we adhere strictly to the OpenCL semantics, existing host protection mechanisms should be compatible with our approach.

4.1 Protection Workflow

When VOCL-FT detects an uncorrected ECC error, the execution is restored from the last good checkpoint, and the current epoch is replayed from logs. At the end of the epoch, VOCL-FT again checks for an ECC error; if one is discovered, the execution is aborted because of having found repeated nonrepairable ECC errors. (This behavior could change in a future version of VOCL-FT to migrate away from the problem memory.) If no ECC error is detected initially, then VOCL-FT creates a new checkpoint to protect the current epoch. After checking for ECC errors one last time to ensure that the checkpoint is valid, the checkpoint and logs are consolidated and saved. The remainder of this section expands on the details of this workflow.

4.2 Execution Logging

OpenCL data is stored in checkpoint files on the system's default temporary directory, which is often mounted either as a *tmpfs* partition or in a fast local persistent storage. In any case, this solution benefits from memory-like speeds and large storage space. While the latter may impact performance slightly because of OS I/O buffering management, the former is subject to disk swapping policies and potential competition for physical memory space. Data transfers between the device and the checkpoint files are performed in a highly tuned pipelined fashion by using asynchronous device transfers and pinned host buffers, similar to the approach described in [18] for GPU and network transfers.

We organized the checkpoints by creating one file per memory object. Usually, the number of memory objects in OpenCL applications is relatively low, which prevents the number of checkpoint files from becoming too large. This way, a single checkpoint file can serve all the command queue uses, reducing the number of checkpoints that must be written if individual objects are used in multiple command queues. These files are "double buffered" to prevent checkpoints from becoming corrupted while being written. That is, while the current checkpoint is written, the previous checkpoint is stored in a separate file, which is then switched after the checkpoint has been verified to not contain any new errors (see "Validate Checkpoint" in Figure 2).

Capturing the data only at synchronization points is not sufficient. For example, the user's execution might use the same host buffer as both device input and output in the same epoch. Because the original input memory is overwritten by the data transfer from the device to the host, it cannot be reused later when trying to replay the epoch in the event of a detected failure. For this reason, we must capture data as it is transferred from the host to the device.

In addition to capturing all the data involved in the executions, we must capture a log of the issued commands. This allows us to automatically replay the commands in the event of an error. The command log is retained in memory because of its relatively small size.

4.3 Failure Detection

Currently VOCL-FT addresses soft data corruption by leveraging existing ECC memory protection. ECC memory automatically detects and corrects single-bit errors (a single bit of data is changed in a word) and can detect, but not correct, double-bit errors (two bits of data are changed in a word). The NVIDIA Management Library (NVML) [16] allows VOCL-FT to query NVIDIA hardware to determine the number of uncorrected ECC errors that have been detected. However, this information is currently provided about the entire device memory, hence preventing our middleware from targeted actions. If in the future information is provided about the place where the uncorrected ECC error happened, a more efficient mechanism could be deployed by performing recovery actions affecting specific memory regions.

We note that the time spent in the third-party error detection call affects the general execution time. We experienced an average of 208 ms employed by the NVML ECC check call, obtaining low runtime overheads.

In order to analyze error recoveries, we simulate ECC error events. These are injected by hardcoding the return of a positive number of ECC errors by the error query function call to happen at the point of interest of each particular case.

4.4 Failure Recovery

Once a failure is detected, VOCL-FT begins its protocol for repairing data and replaying executions for recovery. VOCL-FT restores each device memory object from the checkpoint files. After the memory has been repaired, the command queue is re-executed for the current epoch. All these actions occur without user intervention.

5. OPTIMIZATIONS

In this section we present several optimizations to the baseline approach described in Section 4,[1] designed to reduce runtime overhead and improve recovery time.

5.1 Per-Epoch Buffer Modifications

Our first optimization involves examining when the device and host buffers are modified. In our naive implementation, we checkpoint every buffer at the end of each epoch. Since the entire device memory may not be touched in every epoch, however, such large and time-consuming checkpoints are often not required. These optimizations can be implemented completely transparently to the user.

5.1.1 No Modifications

For those device objects that have not been potentially modified during the epoch, checkpointing can be avoided because the previously stored data is still valid. This is the case for applications that do not touch the entire set of device objects in every epoch. We refer to this optimization as `MOD` (*Modified*).

5.1.2 Coprocessor Modifications

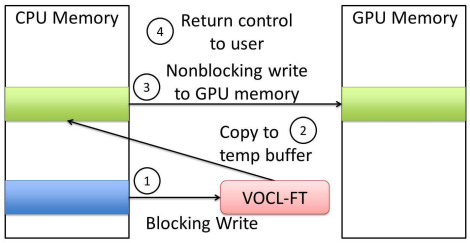
If the `CL_MEM_READ_ONLY` flag is specified during object creation, we can assume that kernels are not allowed to modify the buffer contents and will reuse the previously checkpointed data as long as no host-side modifications (e.g., `clEnqueueWrite`) are present within the epoch. We refer to this optimization as `RO` (*Read Only*).

5.1.3 Host Modifications

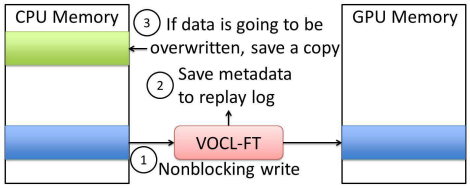
We can extract further information from OpenCL calls that allows us to determine the exact part of the buffer that is to be modified. Here, our optimizations depend on whether the data is being written via a blocking or a nonblocking call. If the host uses a blocking write, the specification of OpenCL states that when the call returns, the user can reuse the buffer; but it does not specify that the complete buffer has been transferred to the coprocessor memory. Therefore, we immediately make a copy of the user-specified host memory contents to an internal host buffer and convert the blocking call to a nonblocking call. In this way, we have a copy of the original host buffer contents that we can use to reissue the write command if necessary. This process is shown in Figure 3a. The implementation of swapping blocking writes for nonblocking writes is common in OpenCL libraries, including the increased memory overhead.

In the case of nonblocking writes, the contents of the user-provided host buffer must be saved when the user makes the call, as discussed in Section 4.2. Unlike a blocking write call, a nonblocking call is not expected to be implemented with an additional buffer, so we use a temporary file. If at the end of an epoch an OpenCL object has been modified only by a

[1] We will refer to the baseline as `NO` (*No Optimizations*).



(a) Blocking.



(b) Nonblocking.

Figure 3: OpenCL write call being captured by VOCL-FT.

host call, after the checkpoint from device memory stage has been validated, the checkpoint file will be updated from the logged data (either in host memory or in a file) instead of the device. We call this a *consolidation* stage (see Figure 2). This stage requires traversing the whole set of commands of the epoch looking for write operations. Since (1) the number of operations per epoch is usually relatively small, (2) the modified regions are potentially smaller than the entire object, and (3) both host–to–file and file–to–file data copies tend to be faster than pipelined device–to–file operations, we leverage the consolidation stage where applicable rather than performing a regular checkpoint from the device. We include this behavior as part of `MOD`.

On the other hand, most applications do not use a single buffer as both input and output. We benefit from this fact with another optimization: `HB` (*Host Buffer*). Instead of immediately logging to a file the user data in a nonblocking write, we log only the information about the user buffer (i.e., its memory location and size). If a subsequent read call modifying any part of the same host buffer is found within the epoch, at that point we perform the delayed copy of the host buffer before queuing the read command. This process is shown in Figure 3b. Because of the relative infrequency of this situation, our optimization avoids most file writes in practice; and the host buffer tracking overhead is negligible in the majority of use cases because of the limited number of commands applications tend to leverage per epoch.

5.2 Extended API

Our second set of optimizations requires additional knowledge not provided by the user when interacting with the original OpenCL API. For this reason, we extend it with a function with which the user can inform VOCL-FT of the per-epoch usage pattern for each OpenCL buffer. We refer to this optimization as `EAPI` (*Extended API*).

5.2.1 Read-Only Buffers

We extend the original approach described in Section 5.1.2 by allowing the application to specify read-only protection per epoch, rather than for the entire memory object lifetime. This extension provides further flexibility and room for finer-grained overhead reductions.

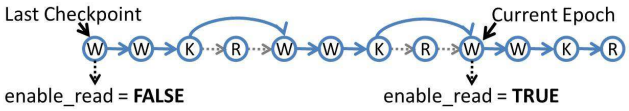


Figure 4: Replay procedure with CS optimization enabled.

5.2.2 Scratch Buffers

Scratch buffers are those whose contents are not meaningful across epochs. This is a common type of buffer for epochs performing tasks such as matrix multiplication. Usually in this scenario the input buffers are provided at the beginning of every epoch, and the output buffers are written at the end of each epoch. Once a buffer is marked as a scratch buffer, VOCL-FT continues to log data written from it, in order to allow kernel replay in the event of a failure (preserving the `HB` optimizations). However, it will not create an on-file checkpoint at the end of the epoch, nor will the buffer's contents be restored in the event of a detected failure. We refer to this optimization as `SB` (*Scratch Buffer*).

5.3 Reducing Synchronization Points

The third set of proposed optimizations focuses on reducing the checkpointing frequency, which poses most of the runtime overhead of coprocessor data protection.

5.3.1 Host Dirty

Synchronization is needed for correctness only in order to ensure that the data read from a device is ready to be used on the host or that a host buffer is ready to be reused after a write to device operation. Frequently, developers introduce arbitrary synchronization points by means of unnecessarily synchronous data transfer operations or explicit synchronizations for timing purposes. Our functionality, however, requires only ensuring that the data reaching the host memory from the device is not potentially corrupted. In this optimization (`HD` or *Host Dirty*) we limit triggering the checkpointing mechanism only on those epoch closings that have contained device read operations, hence guaranteeing the host data integrity. With this mechanism we avoid performing the checkpointing procedure on common code patterns, such as synchronous writes or many synchronizations introduced for timing purposes. `HD` effectively extends VOCL-FT epochs by merging OpenCL epochs.

5.3.2 Checkpoint Skip

The `CS` (*Checkpoint Skip*) optimization avoids performing the file checkpointing stage in every synchronization point. To ensure the integrity of the host data originated on the device, we still perform the ECC check in all synchronization points. If an error is found, the recovery process is triggered. The replay procedure skips read operations occurring before the current synchronization point: the data read and used by the host in those prior epochs was ensured to be sane because of the appropriate ECC checks. This mechanism is compliant with OpenCL's epoch semantics in which the host buffers are assured to no longer be used by the runtime after the synchronization point. This recovery procedure is depicted in Figure 4. The `CS` optimization extends the VOCL-FT epochs farther beyond device to host transfers.

Users may elect different checkpoint frequencies depending on particular runtime overhead and recovery time requirements: lower frequencies lead to lower runtime over-

head but higher recovery times. Our proposed default for entry-level users is to let the framework determine the frequency automatically according to the disk speed, in order to prevent eventually degrading the checkpointing performance. Adjusting the I/O rate at runtime is easily accomplished by measuring the time since the last checkpoint and deciding whether to perform another in the current synchronization point according to the checkpoint sizes. More advanced users may set the checkpointing frequency to a given number of synchronization points or at time intervals. These options can be configured by an environment variable.

6. PERFORMANCE ANALYSIS

We present an in-depth analysis of the performance of the fault-tolerant runtime methodologies we propose. The experiments in this section were run in a single node in order to avoid the influence of internode communication performance artifacts. Production evaluations at scale are presented in Section 7. After introducing our testbed, we present our analysis based on microbenchmarks and two OpenCL codes: MiniMD, a molecular dynamics miniapplication, and Hydro, a full hydrodynamics code. In our evaluation, presenting results based on the average of three executions, we apply optimizations on top of each other. Therefore, when referring to (or plotting) a new optimization, those previously discussed for that particular experiment are also leveraged. For convenience, in this section we refer to the `CS` optimization by its keyword followed by the number of application-level iterations constituting a VOCL-FT epoch. We use the term *native* to refer to experiments on top of the bare NVIDIA's OpenCL implementation, which we use as our baseline.

6.1 Testbed

We performed our performance evaluation in a SuperMicro SuperServer 7047GR-TPRF platform equipped with two Intel E5-2687W v2 octocore CPUs running at 3.40 GHz, 64 GB of DDR3-1866 RAM, an NVIDIA Tesla K40m, and storage based on Intel S3700 SSD disks (where the system's temporary directory is mounted). The server runs the Red Hat Enterprise Linux Server release 6.5 operating system with NVIDIA driver version 340.29.

6.2 Microbenchmark Evaluation

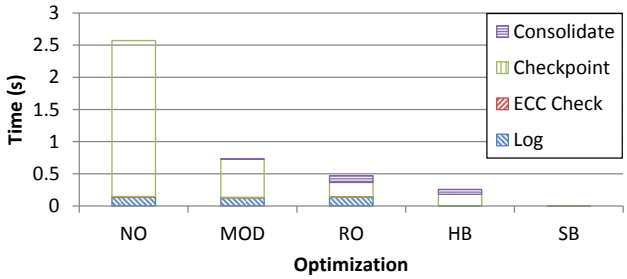
Since the temporary directory is mounted on disk, we first evaluate the storage system and discuss its performance implications. Next, we present a preliminary evaluation of our strategies by means of a synthetic benchmark.

6.2.1 Disk

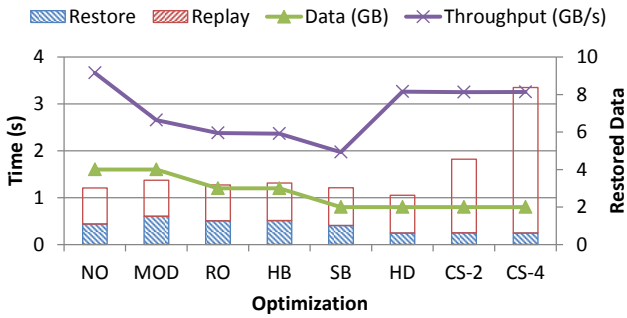
Modern operating systems feature cached I/O operations that are performed asynchronously in the background after buffering the user data into memory. Although meant to accelerate I/O transactions, these introduce timing variability when disk operations are involved, because of the internal I/O buffer management. While our disk features a physical write transfer rate of roughly 300 MB/s, we experience up to 3.5 GB/s for the smallest payloads. When the internal buffer size is exhausted, write speeds decrease, approaching the physical write rate. This effect is also visible in multiple small write operations: the write call exhausting the internal buffering space performs much slower than others, introducing timing variability. Similarly, read speeds from buffered I/O are much higher than those being read directly from disk. Hardware disk buffers introduce another source of disk variability. In Section 6.4 we explored further how this feature impacts our solution.

```
voclSetMemFlags(a_dev, READONLY | SCRATCH)
voclSetMemFlags(b_dev, READONLY | SCRATCH)
voclSetMemFlags(c_dev, SCRATCH);
writeBuffer(dummy_dev, blocking, dummy, 2GB)
for (i=0 to iterations):
  enqueueWrite(a_dev, nonblocking, offset, SIZE/4, a)
  enqueueWrite(b_dev, nonblocking, offset, SIZE/4, b)
  enqueueKernel(sgemm, a_dev, b_dev, c_dev, SIZE/4)
  enqueueRead(c_dev, nonblocking, offset, SIZE/4, c)
  finish()  // Epoch closing
  offset = offset == MAX_OFFSET ? 0 : offset + SIZE/4
```

Figure 5: Microbenchmark pseudocode.



(a) Error-free overhead.



(b) Fault recovery overhead.

Figure 6: Microbenchmark evaluation.

6.2.2 VOCL-FT

We used a synthetic benchmark based on repetitive matrix products in order to showcase the potential benefits of the different optimizations we propose. It features a 2 GB dummy buffer that is initialized immediately after creation but never used further. Matrices of 12,032 floating-point square elements are used, constituting 1.6 GB of memory. A quarter of the matrices is used on each iteration that makes up an epoch. The pseudocode for this benchmark is shown in Figure 5.

The runtime overhead breakdown for 100 fault-free iterations is shown in Figure 6a. In all cases, the time employed in ECC checks (208 ms) is negligible with respect to the other tasks because of having sufficiently long epochs. While the logging time remains constant during the first three cases, the checkpoint time is greatly reduced. `MOD` removes the need for checkpointing the dummy buffer because it is not written or used by a kernel during the epochs. `RO` does the same for `a_dev` and `b_dev`, introducing a consolidation stage to save the written data to the checkpoint file, saving only the transferred data (one quarter of the memory object) in-

Table 1: MiniMD runtime overhead of first optimizations.

	Log	ECC Check	Checkpoint	Consolidate
NO	2.9%	74.4%	1,679.9%	0.0%
MOD			567.3%	1.6%

Table 2: MiniMD synchronizations per 100 timesteps.

NO	HD	MA	CS-25	CS-50	CS-100	CS-200
330	25	16	4	2	1	1/2

stead of the whole object. HB removes most of the remaining runtime overhead because the host buffers used to write data to the device are not employed on read operations. Thus we avoid saving the user data during the logging stage. SB removes the need for tracking c_dev because its contents are meaningless across epochs. On the other hand, HD and CS are meant to reduce the synchronization points and do not lower the runtime overhead further in this simple example.

Figure 6b shows our recovery evaluation for a single failure event. Restore times differ for the same data size because we experience different read speeds due to the I/O buffering effects. For instance, the NO restore throughput[2] is higher than that of MOD because in the former the dummy_dev object is written to disk in every checkpoint and hence all data to be recovered is more recently used than in the latter. HB slightly increases the replay stage (5%) because writes to device are performed from pageable user buffers instead of our pipelined device and disk transfers using pinned host buffers, which attain higher transfer rates. Because their epochs are twice as long, the CS-2 and CS-4 optimizations almost double their replay time with respect to the previous cases. These replay times pose close to 100% of an epoch's time increase because, in spite of avoiding intermediate read operations (see Figure 4), the epoch time in this code is dominated by the kernel execution.

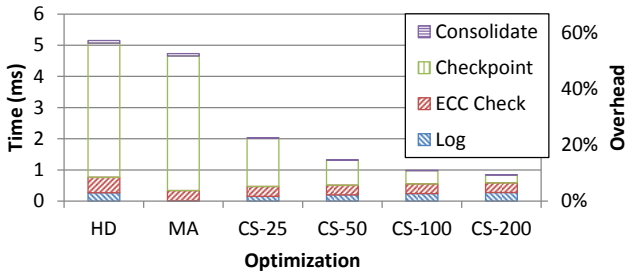
6.3 MiniMD

MiniMD [8] is a codesign miniapplication targeted at evaluating the performance of molecular dynamics simulations. In our experiments we used the OpenCL implementation of MiniMD version 1.2. We performed experimental evaluations with the biggest size supported by this version: 237,276 atoms, employing roughly 180 MB of GPU memory. The relative standard deviation (RSD) for our three experiment repetitions is under 5% for all nonnegligible measured times, validating the significance of our experimentation.

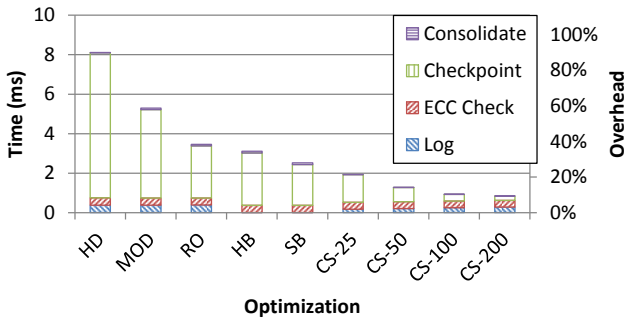
Our fault-free evaluation results are shown in Table 1 and Figure 7, reporting per-timestep numbers averaged from 1,000 timestep executions. In Table 1 we observe that the overhead with respect to native executions when leveraging the first set of optimizations is considerable, caused mainly by the checkpointing stage. This situation stems from the high number of synchronization points introduced by the code (see Table 2). RO cannot obtain any benefit because of the absence of the corresponding OpenCL flags in the user code on device memory allocation calls. Similarly, no memory objects offer optimization opportunities for MOD and HB.

Figure 7a shows our results for a more aggressive set of optimizations. HD reduces the overhead to under 60% thanks to skipping the checkpointing stage in those epochs finishing without having touched the host memory—introduced

[2]We define *Restore Throughput* as the effective transfer rate obtained restoring the contents of the relevant OpenCL memory objects to their status at the last check point.



(a) Aggressive optimizations.



(b) Extended API.

Figure 7: Error-free MiniMD execution times.

mainly with timing purposes. As shown in Table 2, this optimization reduces the synchronization points from 330 to 25 every 100 timesteps.

For the next optimization in Figure 7a, MA (*Manual*), we performed a manual reduction of the synchronization points of this code by simply replacing consecutive sets of synchronous data transfer operations with their asynchronous counterpart followed by a final synchronization. While the HD optimization is not able to automatically skip checkpointing in synchronous reads, we can benefit from application-level semantics to perform this manual optimization. This avoids 9 more checkpoints every 100 timesteps, reducing the overhead another 6% while not noticeably impacting the native runtime. This optimization also removes most of the logging overhead that was caused by saving user data from synchronous data transfer operations.

The CS optimizations presented in Figure 7a reduce the costly checkpoint operations further. We explored from a check every 25 (CS-25) to 200 (CS-200) timesteps, attaining execution overheads from 24% to under 10%, respectively. As we can observe in the plot, the checkpoint time per iteration decreases proportionally to the number of checkpoints avoided. From the different frequencies we explored, the first that meets our 300 MB/s disk write boundary is CS-50 (see Figure 8), attaining less than 15% overhead. We examine the I/O effects in our Hydro evaluations in Section 6.4. Since we are able to use larger datasets with that use case, these effects become more obvious and easily explored.

Figure 7b shows our MiniMD fault-free evaluation when using the extended API. We apply our successive set of optimizations in a different order to showcase better the benefit of the different optimizations. Our starting point, the MA optimization, is not shown in the figure because its large overhead would make the graph difficult to read. The HD optimization attains 91% runtime overhead. When applying the MOD optimization we reduce the overhead to less than 60%. The RO optimization brings another 20% overhead re-

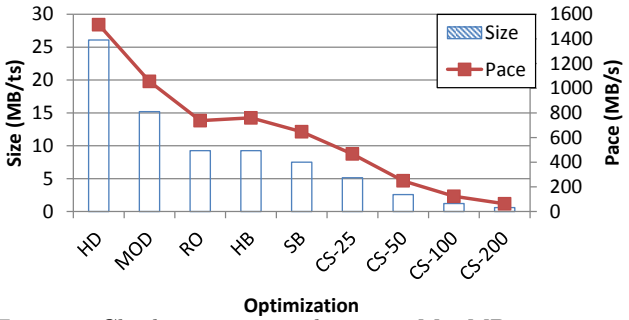
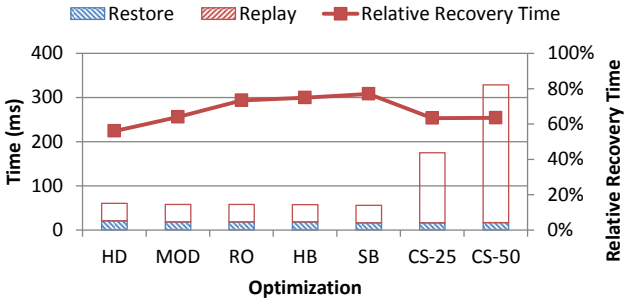
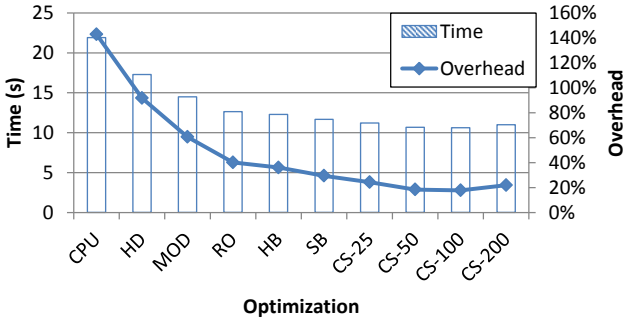


Figure 8: Checkpoint sizes and rates on MiniMD executions.



(a) Recovery time breakdown.

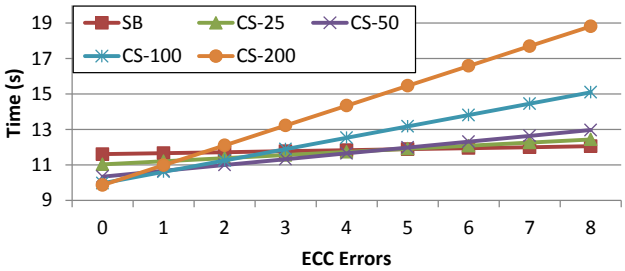


(b) Execution time in 1,000 timesteps.
Figure 9: MiniMD recovery times in a single faulty epoch.

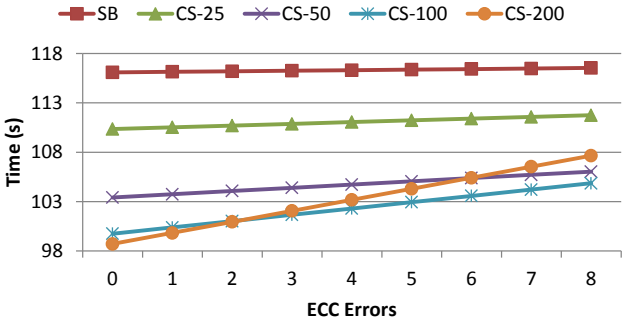
duction, enabled thanks to properly reporting the per-epoch read-only properties of the different memory objects. The `HB` optimization removes most of the logging overhead, now available because of the longer epochs, and reduces the runtime overhead to roughly 35%. The `SB` optimization, enabled by the introduced extended API calls, brings another 7% overhead reduction in the checkpoint stage by avoiding unnecessarily saving the contents of scratch buffers. The set of `CS` optimizations behave as in the former plot, leading from 22.4% to 9.5% overheads.

Recovery times on MiniMD for a single faulty epoch are reported in Figure 9. Figure 9a shows the breakdown of the recovery times for the different optimizations in the order explored in Figure 7b. Replay times in this case are over twice the restore times. Up to the `SB` optimization, the replaying stage takes constantly around 40 ms. The relative recovery time[3] increases as we apply more optimizations because these decrease the VOCL-FT overhead and make executions more efficient. The `CS` optimizations have lower relative recovery times because of skipping read operations.

[3]We define *Relative Recovery Time* as the relative execution time of the recovery stage with respect to nonfaulty epochs.



(a) 1,000 timesteps.



(b) 10,000 timesteps.
Figure 10: MiniMD recovery for multiple faulty epochs.

Figure 9b presents execution times for 1,000 timesteps and a single faulty epoch with the different optimizations. The overhead with respect to native error-free executions is also shown in the figure. CPU times employing all the cores by means of the OpenMP version of the code are included for comparison. In the `CS` cases, worst-case scenarios are considered; that is, uncorrected ECC errors are found in the check that closes a VOCL-FT epoch. The restore throughput reaches over 90% efficiency with respect to the available GPU bandwidth thanks to the fast cached disk read times. On the other hand, we can see that the execution times decrease up to the `CS-100` optimization. The runtime overhead reduction in `CS-200` with respect to the `CS-100` case is not sufficiently high to overcome the longer replay time for the number of executed timesteps. As depicted in Figures 10a and 10b, there is a tradeoff not only between the error rate and the check period but also the runtime, as longer executions make the use of longer epochs beneficial if the error rate is sufficiently low by attaining low runtime overhead.

6.4 Hydro

The Hydro benchmark is a complete hydrodynamics code implementing a 2D Eulerian scheme by means of a Godunov method. We employ the OpenCL variant [13] published in https://github.com/HydroBench/Hydro.git. We use the most recent version available at the master branch of the repository at the time of benchmarking: commit ef3a63b5 dating from October 21, 2014. The RSD for our three experiment repetitions is under 6% for all nonnegligible measured times, which confirms the significance of our results.

We first evaluate error-free executions when leveraging different optimizations on a range of problem sizes, reaching up to the maximum 4 GB of device memory exposed by NVIDIA's OpenCL implementation.

Table 3 presents per-timestep overheads with respect to native executions when the first optimizations are applied. In the `NO` case we obtain an unbearable slowdown starting at

Table 3: Evaluation of first optimizations in Hydro.

Dimension (Elements)		1,000	2,000	3,000
Overhead (%)	NO	29,467	54,452	80,026
	MOD	7,659	10,747	14,235
Synchronizations		603	1,167	1,750

Table 4: Hydro runtime overhead on a 1,000 × 1,000 mesh.

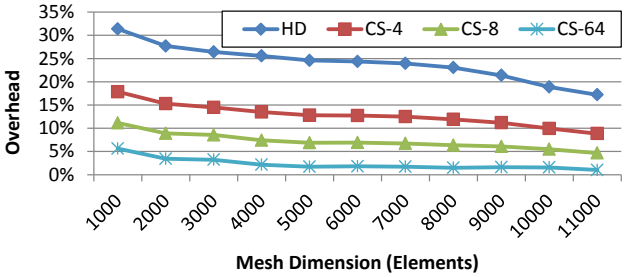
	Log	ECC Check	Checkpoint	Consolidate
NO	1.2%	4,646.1%	25,049.2%	0.2%
MOD			6,519.9%	0.2%

almost 30,000%, caused mostly by the checkpointing stage, as shown in Table 4 for the 1,000 by 1,000 mesh. In this case, MOD brings a large overhead reduction by avoiding saving to disk those objects not modified by the executed kernel on the corresponding epoch. This application cannot benefit from the remainder of the set of first optimizations except from EAPI, with which we can set two small temporary memory objects as scratch buffers; however, this does not pose a noticeable performance impact. These considerable slowdowns result from the large number of synchronizations per timestep that this code performs (see Table 3). Most of these, however, are systematically performed after many OpenCL calls, and their semantics are not actually required for a correct execution. HD automatically detects this situation and provides a large performance improvement.

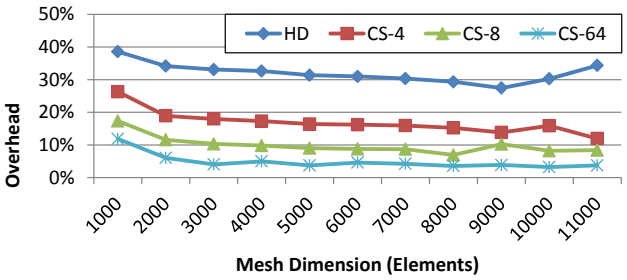
Figure 11a shows the results when applying a more aggressive set of optimizations. In this experiment we avoid the I/O variability by committing the buffer cache to disk before checkpointing (by means of the sync POSIX call) and subtracting this call's time. In this way we obtain a clean picture of the *expected* overhead without the inherent I/O caching variability. The HD optimization attains a reduction in the number of synchronization points to one every two timesteps only: a device buffer read as part of a reduction process in the *compute delta T* operation. This optimization greatly reduces the overhead—up to 17% in our experiments. The CS optimizations we show as an example (4, 8, and 64 timesteps) attain further overhead reductions of up to 1% in the CS-64 case with the largest data size, by avoiding the checkpoint stage in every synchronization point. The trend in this plot indicates that the checkpointing overhead increases more slowly than the rest of the tasks, with the problem size not being a bottleneck any further. Figures 12a and 12b show per-timestep overhead breakdowns, revealing that these are dominated by the checkpointing stage.

Figure 11b shows the *measured* overhead, including regular I/O operations, on executions covering five synchronization points in the different optimizations. As expected, the obtained overhead is larger than that shown in the former plot because of the inherent operating system I/O management, as discussed in Section 6.2.1. The HD case shows an increasing trend in our two largest problem sizes, while the remaining optimizations reveal more stable trends. This behavior is related to the rate at which the VOCL-FT middleware performs disk write operations on the different optimizations, as shown in Figure 13. While all optimizations reveal a decreasing disk write pace trend as the problem size increases, because of the larger increase in computation than checkpointing time with the problem size, only the CS-8 and CS-64 cases show consistently sustainable transfer rates under our disk actual write throughput of roughly 300 MB/s.

Figure 14a shows per-epoch recovery times with respect to

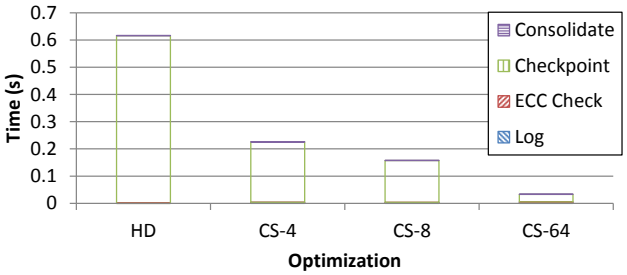


(a) Without I/O effects.

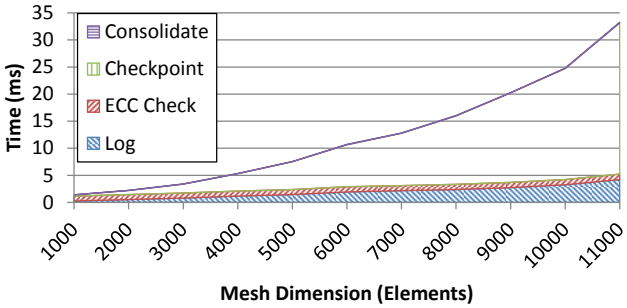


(b) Regular executions.

Figure 11: Error-free Hydro runs—aggressive optimizations.



(a) Aggressive optimizations on an 11,000 × 11,000 mesh.



(b) Overhead breakdown with the CS-64 optimization.

Figure 12: Error-free Hydro execution times—breakdown.

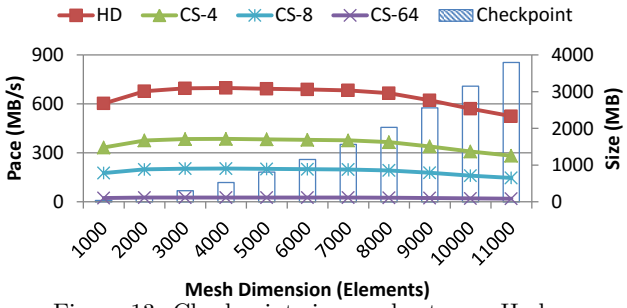
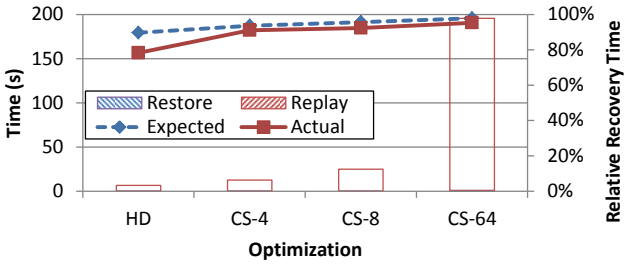
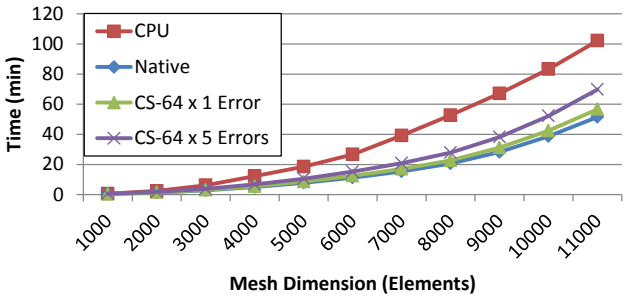


Figure 13: Checkpoint sizes and rates on Hydro.



(a) Hydro recovery times on an 11,000 × 11,000 mesh.



(b) Hydro error recovery versus CPU-only executions.
Figure 14: Hydro recovery times.

error-free executions using VOCL-FT in the 11,000 square elements mesh case. The `CS` cases represent worst-case scenarios as discussed for MiniMD. This implies a different number of timesteps depending on the epoch size: two timesteps in the `HD` case, and four (`CS-4`), eight (`CS-8`), and sixty-four (`CS-64`) timesteps in the `CS` cases. The restore operation poses roughly 0.4 s of this process, attaining over 7 GB/s. This high recovery rate is enabled by the operating system I/O cache. Restores using cached data are expected to be a common case, hence not posing a significant portion of the recovery time. On the other hand, replay times grow proportionally to the VOCL-FT epoch length. Command re-executions, dominating the recovery time, are faster than error-free executions because of avoiding host computations and read operations. As shown in the figure, the *expected* relative recovery time, following the mechanism as in Figure 11a, varies between 90% for the `HD` case and 98% for `CS-64`. The relative recovery time becomes higher with the size of the epoch because the epoch checkpointing time is amortized among a larger number of timesteps. The *actual* relative recovery times do not present such a neat trend and are comparatively smaller, between 78% and 95%, because the I/O overhead poses a larger portion of time in the "real" epoch than in the "expected" counterpart, being lower with the decrease of the checkpointing frequency.

Results for error-free executions of 1,000 timesteps of the CPU-only OpenMP version of the code using all the CPU cores of the computer are shown in Figure 14b as a comparison of our fault-safe accelerated proposal with the execution on a traditional nonaccelerated environment. We compare these with VOCL-FT executions leveraging the longest checkpointing period we have considered, `CS-64`, leading to the lowest runtime overhead but longest recovery time. Our results reveal that GPU-accelerated executions suffering from an uncorrected ECC error are executed 45% faster than they would in a CPU-only run on an error-free environment in the biggest mesh we could execute and only slightly slower than a native execution on a healthy device. In the unlikely case, assuming a healthy device at current error rates [6], that during the close-to-an-hour execution the GPU would experience 5 uncorrected ECC errors, the executions would still be correct on the device, attaining over 30% speedup with respect to their nonaccelerated counterpart.

In production runs executing thousands of timesteps [13], where the possibility of experiencing a soft error event on the accelerator is a reality [6], having a runtime ensuring the correctness of executions at the expense of a small overhead is clearly appealing. In the Hydro case, GPU-accelerated runs attain roughly 2x speedup with respect to their CPU-only counterpart, which is a common minimum factor for applications featuring a good match with accelerator architectures. Therefore, we expect our techniques to attain similar or better recovery results in most coprocessor-accelerated applications with respect to CPU-only executions.

6.5 Discussion

Both of our test cases experience a relatively high synchronization frequency, what introduces nonnegligible runtime overhead in order to ensure data integrity: at the very least, error checks must be performed at every synchronization point. This overhead is higher as the amount of data used by non-scratch buffers increases. Hence, MiniMD's overhead prior to `HD` and `CS` is lower than that of Hydro: MiniMD's maximum dataset is relatively reduced. While `HD` skips many arbitrary synchronization points, it is necessary to apply `CS` to reduce the data backup overhead to reasonable levels. The remaining optimizations help reducing the data backup overhead by determining what portions of the device memory require to be backed up according to the specific application use of the OpenCL API.

In those use cases in which scratch buffers are dominant, however, those optimizations designed to reduce synchronization points (`HD` and `CS`) would not lead to further significant performance improvements: the data backup overhead would be negligible. That would be the case of those applications using coprocessors mainly as massively-parallel solvers with little or no persistent data among kernel runs.

7. EVALUATION AT SCALE

We next showcase how the use of our techniques affect production executions at scale. We first introduce our evaluation platform and then analyze the results.

7.1 Experimental Environment

We use the HokieSpeed supercomputer [27] from Virginia Tech. Its compute nodes, interconnected by an InfiniBand QDR fabric, feature two hexa-core Intel Xeon E5645 CPUs (24 GB of RAM) and two NVIDIA Tesla M2050 GPUs (5.25 GB of combined memory). The temporary directory in this system is mounted as a *tmpfs* partition. We leverage Intel MPI [9] version 4.1 as the MPI implementation.

7.2 Experimental Evaluation

In our production evaluation at scale we employ the Hydro application, which presents good scalability properties (note that the OpenCL version of MiniMD cannot be used at scale because of the input size limitation). We perform both strong- and weak-scaling evaluations using up to 128 GPUs in 64 nodes, the largest allocation permitted in HokieSpeed. In our 1,000-timestep executions using all the GPU memory, both the host data and the temporary files on the *tmpfs*

filesystem for the two GPUs can be held by the main host memory of the nodes without disk swapping interferences.

We compare error-free native (`Native`) with VOCL-FT (`V-FTx0E`) runs leveraging the full set of optimizations. The VOCL-FT runtime is configured to checkpoint every 3 minutes. We select intentionally a relatively frequent interval for a production run to demonstrate the low overhead enabled by our techniques. VOCL-FT executions including recovery from 1 (`V-FTx1E`) and 5 (`V-FTx5E`) disjoint uncorrected ECC events are also discussed as examples of faulty executions. The errors occur on a single GPU at the end of the second VOCL-FT epoch, which for short runs in the strong scaling experiments is the last synchronization point instead of a 3-minute epoch. We also performed CPU-only runs employing the hybrid OpenMP+MPI Hydro version leveraging the 12 cores per node of the system. For clarity, we skipped them in the plots, since their execution time is an order of magnitude higher than that of the other experiments. The RSDs for the three repetitions of our experiments in this section are under 1%.

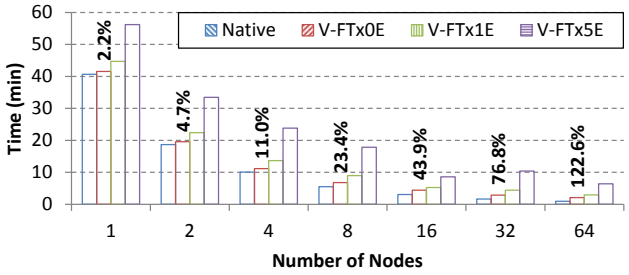
For our strong-scaling experiments, Hydro computes a mesh of 11,000 square elements, which roughly fills the combined GPUs memory of a HokieSpeed's compute node. Our results are shown in Figure 15a. We observe a reduced 2.2% overhead on single-node error-free VOCL-FT executions with respect to the native OpenCL library. This overhead increases with the number of nodes involved because of the lower compute loads. Derived from our checkpointing interval, each recovery procedure lasts 136–176 s up to 8 nodes (lower as the GPU load decreases), while the remainder of the experiments recover from shorter VOCL-FT epochs. Executions incurring one recovery event start at execution times 10% longer than `Native` for the single-node case, naturally increasing with the reduction of the load per node. In the unlikely case of experiencing 5 recovery events, VOCL-FT driven executions still finish with correct results an order of magnitude faster than their CPU-only counterpart does.

The weak-scaling experiments, involving meshes of 11,000 to 88,000 square elements, are presented in Figure 15b. The error-free overheads caused by the VOCL-FT runtime are roughly between 2% and 6%. The trend to a slight overhead increase with the number of processes is due to more communications being performed, leading in this code to further GPU data movements and synchronization points. When a recovery event is triggered, we experience from 10% to 13% increased execution time with respect to error-free native executions, which corresponds with the configured checkpoint period. In the unlikely 5-error recovery sample case, correctness is provided at the expense of a 38–45% runtime increase, which is still well below that of the CPU-only (error-free) version.

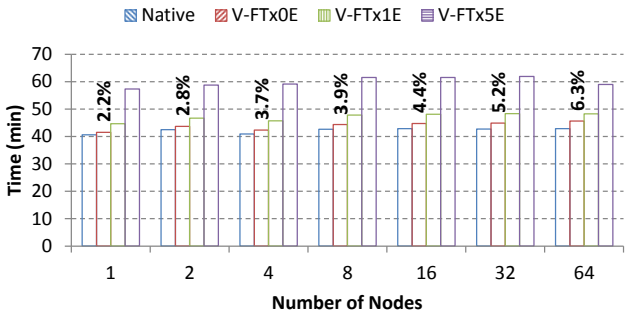
Our production-level experiments reveal similar performance properties to those of single-node runs (Section 6), indicating that our proposal is feasible for production executions at scale. The experiments in this section confirm that our solution does not significantly affect the original scalability properties of the application being protected.

8. CONCLUSION

In this paper we have explored efficient data protection mechanisms for coprocessor memories based on API interception. We have devised an extensive set of optimization



(a) Strong scaling.



(b) Weak scaling.

Figure 15: Hydro evaluation at scale. Bar labels indicate overhead with respect to `Native`.

opportunities derived from the OpenCL semantics that enable a low-cost runtime system. We have demonstrated, both in single-node environments and in production at scale, that our methodologies can provide error-free runtime overheads below 5% while attaining low recovery times for reasonably long runs and common device error rates.

9. ACKNOWLEDGMENTS

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation at Argonne National Laboratory. This work was also supported by NSF grant CNS-0960081 and the HokieSpeed supercomputer at Virginia Tech. The authors also thank Leonardo Bautista, from Argonne National Laboratory, for his useful comments on this paper.

10. REFERENCES

[1] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[2] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[3] A. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, and M. Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *ACM/IEEE Supercomputing Conference*, Nov. 2003.

[4] A. Castelló, J. Duato, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, V. Roca, and F. Silla. On the use of remote GPUs and low-power processors for the acceleration of scientific applications. In *International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY)*, 2014.

[5] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient and flexible resilience scheme for exascale systems. *Scientific Programming*, 21(3-4):197–212, 2013.

[6] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda. GPGPUs: how to combine high computational power with high reliability. In *Design, Automation & Test in Europe (DATE)*, 2014.

[7] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[8] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical report, Sandia National Laboratories, 2009.

[9] Intel Corporation. Intel MPI library. http://software.intel.com/en-us/intel-mpi-library, 2015.

[10] A. Kawai, K. Yasuoka, K. Yoshikawa, and T. Narumi. Distributed-Shared CUDA: Virtualization of large-scale GPU systems for programmability and reliability. In *International Conference on Future Computational Technologies and Applications*, 2012.

[11] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *26th International Conference on Supercomputing*, pages 341–352. ACM, 2012.

[12] P. Lama, Y. Li, A. M. Aji, P. Balaji, J. Dinan, S. Xiao, Y. Zhang, W. Feng, R. Thakur, and X. Zhou. pVOCL: Power-aware dynamic placement and migration in virtualized GPU environments. In *33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 145–154. IEEE, 2013.

[13] P. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J. Dupays. Porting and optimizing HYDRO to new platforms and programming paradigms–lessons learnt. Technical report, PRACE, Dec. 2012.

[14] A. Munshi, editor. *The OpenCL Specification Version 2.0*. Khronos OpenCL Working Group, 2014.

[15] R. Naseer and J. Draper. Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs. In *34th European Solid-State Circuits Conference (ESSCIRC)*, pages 222–225, Sept. 2008.

[16] NVIDIA Corporation. NVIDIA Management Library (NVML). http://developer.nvidia.com/nvidia-management-library-nvml, 2015.

[17] A. J. Peña. *Virtualization of accelerators in high performance clusters*. PhD thesis, Universitat Jaume I, Castellon, Spain, Jan. 2013.

[18] A. J. Peña and S. R. Alam. Evaluation of inter- and intra-node data transfer efficiencies between GPU devices and their impact on scalable applications. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[19] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Orti, and J. Duato. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computing*, 40(10):574–588, 2014.

[20] M. Rebaudengo, M. Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 210–218, Nov 1999.

[21] A. Rezaei, G. Coviello, C. Li, S. Chakradhar, and F. Mueller. Snapify: capturing snapshots of offload applications on Xeon Phi manycore processors. In *International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2014.

[22] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[23] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. TOP500 supercomputing sites. http://www.top500.org/lists/2014/11, Nov. 2014.

[24] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent checkpointing and process migration of OpenCL applications. In *International Parallel & Distributed Processing Symposium (IPDPS)*, pages 864–876. IEEE, 2011.

[25] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A checkpoint/restart tool for CUDA applications. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 408–413. IEEE, 2009.

[26] TSUBAME Computing Services. Failure history of TSUBAME2.5. http://mon.g.gsic.titech.ac.jp/trouble-list, 2015.

[27] Virginia Tech. HokieSpeed (Seneca CPU–GPU). http://www.arc.vt.edu/resources/hpc/hokiespeed.php, 2015.

[28] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *International Symposium on Computer Architecture (ISCA)*, pages 73–84, 2014.

[29] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S. Lu. Reducing cache power with low-cost, multi-bit error-correcting codes. *SIGARCH Comput. Archit. News*, 38(3):83–93, 2010.

[30] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing (InPar)*. IEEE, 2012.

[31] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 287–300, 2011.