Received: 15 December 2020 | Revised: 6 May 2021 | Accepted: 14 June 2021

DOI: 10.1002/cpe.6474

SPECIAL ISSUE PAPER

WILEY

Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support

Niklas Eiling | **Jonas Baude** | **Stefan Lankes** | **Antonello Monti**

E.ON Energy Research Center, Institute for Automation of Complex Power Systems, RWTH Aachen University, Aachen, Germany

Correspondence
Niklas Eiling, E.ON Energy Research Center, Institute for Automation of Complex Power Systems, RWTH Aachen University, Mathieustraße 10, Aachen 52074, Germany.
Email: niklas.eiling@eonerc.rwth-aachen.de

Funding information
Bundesministerium für Bildung und Forschung, Grant/Award Number: Grant 01IH16010C; H2020 Industrial Leadership, Grant/Award Number: 957246

Abstract

In high-performance computing and cloud computing the introduction of heterogeneous computing resources, such as GPU accelerator have led to a dramatic increase in performance and efficiency. While the benefits of virtualization features in these environments are well researched, GPUs do not offer virtualization support that enables fine-grained control, increased flexibility, and fault tolerance. In this article, we present Cricket: A transparent and low-overhead solution to GPU virtualization that enables future research into other virtualization techniques, due to its open-source nature. Cricket supports remote execution and checkpoint/restart of CUDA applications. Both features enable the distribution of GPU tasks dynamically and flexibly across computing nodes and the multitenant usage of GPU resources, thereby improving flexibility and utilization for high-performance and cloud computing.

KEYWORDS

checkpoint/restart, GPU, remote execution, virtualization

1 | INTRODUCTION

Our work on Cricket is motivated by two trends: the emergence of *heterogeneous computing* systems for high-performance computing (HPC) and the enhancements in virtualization technology that led to the era of *cloud computing*.

Regarding heterogeneous computing, the number of GPU accelerators employed in clusters has already reached a significant amount of 28% in the Top500* list by November 2020. This is because GPUs provide a high-performance and power efficiency for computing tasks while offering a better performance/price ratio compared with CPUs.[1,2] These advantages result from the optimization of GPUs for highly parallel programs that use similarly executing threads to process large amounts of data. As the strengths of GPUs are also beneficial for, for example, multimedia processing or machine learning applications, cloud computing companies also deploy GPU accelerators.[3,4] GPU-accelerated clouds offer more cost-efficient deployments of these services, as GPUs can be used whenever needed on a pay-per-use basis. However, *GPU virtualization* is essential to fulfill typical cloud requirements that enable multitenant cluster usage, that is, isolation, fine-grained control of computing resources, and scalability. Also in HPC clusters, virtualization of GPUs promises more flexible assignment and a higher level of control over computing resources, thereby increasing the availability and utilization of clusters.[5,6]

In this article, we particularly focus on two specific techniques enabled by the virtualization of GPUs: Remote execution and checkpoint/restart. *Remote execution* allows the transparent execution of the GPU related code of an application on a remote system. This enables the execution of a GPU-accelerated applications on two distinct systems, separating the CPU-intensive work from GPU-intensive work. In addition, it allows sharing of

* A list (https://www.top500.org/) ranking 500 of the world's fastest HPC clusters.

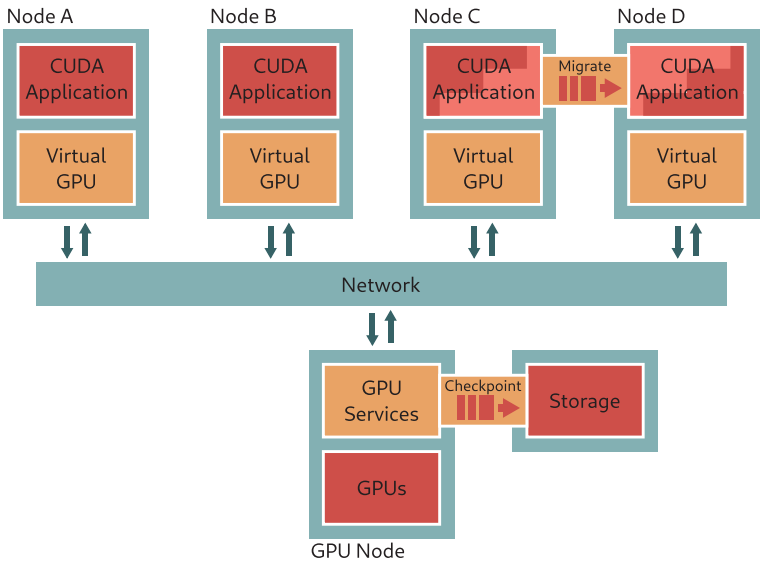


FIGURE 1 Conceptional overview of the Cricket virtualization layer. CUDA applications running an one of the nodes A–D can access GPUs on a dedicated GPU node via remote-procedure calls. Furthermore, applications can be checkpointed or migrated

GPU resources between multiple systems allowing GPUs to be heterogeneously distributed across a cluster. This increases GPU utilization because the amount of GPU and CPU resources assigned to jobs becomes flexible.[7]

Checkpoint/restart is a technique used to increase fault-tolerance by saving the execution state of an application to persistent memory. The written files are called *checkpoints* and can be used to *restore* the application, for example, after a fault has occurred. We distinguish between *application-level*checkpoints that are created by the application itself and *system-level* checkpointing, which does not require any modifications of the application. While implementing application-level checkpointing causes development overhead, it can benefit from the detailed knowledge of the application in order to minimize the data that needs to be saved. By being fully transparent to the application, system-level checkpointing lowers the development burdens but may lead to much larger checkpoint files and checkpoint creation times, as the entire internal application state has to be extracted, including a memory dump.[8] With checkpoint/restart the migration of running applications between devices or systems becomes possible, enabling on-the-fly reorganization of tasks distributed across a cluster, for example, for dynamic load balancing or maintenance.[5]

Both, remote execution and checkpoint/restart for GPUs require virtualization in order to forward or record the interactions of CUDA applications with the CUDA APIs, which keep an internal state that cannot be directly accessed due to the closed-source nature of NVIDIA drivers and APIs.

In this work, we introduce *Cricket*: A virtualization layer implementation that supports remote execution and checkpoint/restart for the CUDA applications. Our checkpoint/restart implementation focuses on saving and restoring the CUDA API related execution state, while using CRIU[†] to checkpoint and restore the part of the execution state that is not related to the GPU execution. We focus on NVIDIA GPUs and CUDA, as these products are most commonly used for computing tasks[‡].

Figure 1 illustrates an application scenario for a cluster that uses GPU virtualization as enabled by Cricket. The virtualization of GPUs involves inserting a virtualization layer between CUDA applications and the GPU device. Conceptually, the CUDA application uses a virtual GPU instead of the real device, thus decoupling the CPU part of the application from the GPU part. This allows complete control of the interactions between CUDA applications and the GPU. Requests to the GPU are made using *remote procedure calls (RPC)* to a GPU service that can be located on a remote GPU node. This enables clusters where GPU resources are concentrated in a few nodes instead of being distributed homogeneously across all nodes. The consequence is an increased utilization of GPU resources, as CPU-intensive jobs do not block GPU resources, when they create a high load on a node equipped with a GPU. The GPU node in Figure 1 is only used for GPU jobs, while the associated CPU code is executed on one of the nodes A to D.

The checkpoint/restart support for CUDA application offered by Cricket, leads to great flexibility in the location where the GPU and CPU code of applications executes. Figure 1 shows the migration of a CUDA application between nodes C and D. Independently of this, the GPU service can checkpoint the application related state of the CUDA APIs. This allows moving applications between nodes for load balancing or maintenance reasons, and makes them tolerant to faults in both nodes.[5]

In contrast to the closed-source NVIDIA drivers and APIs, our open source virtualization layer provides a basis for future research on virtualization techniques that require full control of interactions between CUDA applications and the GPU. An example of a technique that may benefit from the availability of Cricket is custom GPU task scheduling, which allows the dynamic optimization of the resource usage for a cluster-wide goal

[†]Checkpoint/Restart In User-space (https://www.criu.org/)
[‡]The Top500 list from November 2020 contains only a single cluster that uses GPUs from another vendor.

(such as energy efficiency or performance) and limit or track the usage of GPU resources. This is something that is currently not possible with the official CUDA tools.

While GPU virtualization is desirable, it is also challenging due to the tight integration between user-level code and the device driver that manages the interaction between CPU and GPU. As accelerator devices, GPUs are able to execute programs similarly to CPUs, but require to be controlled by CPU code. Thus, a GPU application consists of a CPU part and a GPU part. The CPU part consists of a process that interacts with the GPU driver to provide the GPU part with input data, to launch the GPU code, and to collect the computation results. Furthermore, GPUs have on-board memory that is separate from the main memory accessible by the CPU. To distinguish the two memory types, we stick to the CUDA terminology of calling the GPU memory *device memory* and the CPU memory *host memory*. While there are several frameworks that developers might use to create GPU applications,[9] CUDA is most commonly used for the implementation of computing applications. Because of the market dominance of CUDA, we focus on virtualization for applications that are implemented using this technology. It is only available for NVIDIA GPUs, however for these GPUs, it provides more advanced features and is better able to fully utilize the performance potential than, for example, the vendor-neutral OpenCL.[10] CUDA consists of several software layers with multiple APIs, that provide different abstraction levels for the interaction with the GPU, most notable the CUDA runtime library and the lower-level driver library. It includes libraries offering high-level primitives for linear algebra calculation, matrix solvers and neural network computation, but also the CUDA runtime library for the development of GPU code in C++ and the CUDA driver library for lower-level development in C.[11] NVIDIA keeps the implementation of these libraries proprietary. This significantly hinders the research on novel GPU virtualization techniques for which the interaction of applications with the GPU devices has to be manipulated.

This article presents a virtualization layer that enables the realization of he previously discussed scenario (see Figure 1). It is able to fully control the usage of GPU resources of CUDA applications, thus allowing redirection, manipulation and recording of device interactions, while CUDA applications stay unaware of the virtualization.

This article is an extended version of our previous work,[12] where we presented our virtualization layer implementation capable of remote execution of CUDA applications. We extend this work by implementing checkpoint/restart. We add a discussion of previous work on checkpoint/restart and present the new implementation in chapter 4. Furthermore, we extend the depth of the evaluation of Cricket and add an evaluation of the checkpoint/restart capabilities.

The rest of the article is structured as follows: Section 2 discusses related work, Section 3 describes the design of Cricket, and Section 4 covers checkpoint/restart. Section 5 presents an evaluation of our GPU virtualization layer and Section 6 concludes this work.

2 | RELATED WORK

There has previously been interest in virtualization layers and checkpoint/restart for GPU applications. However, for many virtualization solutions no source code is available and others support only outdated CUDA versions.

2.1 | GPU virtualization and API remoting

The closed-source nature of GPU drivers makes virtualization approaches that require modified drivers impossible. API remoting enables GPU virtualization at the documented library level between the application and the driver. In addition to decoupling the application and the driver, this approach can facilitate remote execution by forwarding API calls to a remote machine. However, this approach can cause higher overhead than other approaches.[4] The following paragraphs discuss different related GPU virtualization approaches.

rCUDA allows CUDA applications to use GPUs installed in a remote system.[13] This is achieved by replacing the CUDA APIs with alternatives that forward CUDA API calls of local applications to a remote machine either via a TCP connection or via Infiniband verbs. rCUDA supports the driver API and the runtime API as well as several higher-level CUDA APIs, such as cuDNN, cuSOLVER, and cuBLAS. The runtime API is reimplemented using the driver API, making the implementation of new API functions work-intensive as there is not always a clear driver API counterpart to runtime API functions. The tool achieves memory bandwidths that are comparable with native CUDA executions, when a sufficiently fast interconnect is used.[14] The most recent release does only support CUDA 9.0 and may therefore not be used with GPUs from the Turing or Ampere generations that were released in 2018 and 2020, respectively. rCUDA is not open-source and the authors publish implementation details only to a limited extent, making detailed evaluation of the approach and code reuse impossible. The authors target users who want to execute existing applications remotely and therefore do not require source code access. However, not being open-source makes rCUDA impossible to use for research into advanced virtualization strategies such as those previously discussed.

DS-CUDA is a GPU virtualization approach that targets a scenario where a cloud provides the GPU resources for local CUDA applications.[15] Similarly to rCuda, DS-CUDA uses a client-server architecture, where API calls in a CUDA application are forwarded to a server that interacts with the GPU devices. For the communication, DS-CUDA can use RPC or Infiniband verbs. The authors increase the reliability of GPU calculations by allowing redundant calculations, where API calls are performed on multiple GPUs and repeated if they have different results. DS-CUDA is licensed

under a GPLv3 license and supports version 6 of the CUDA toolkit. This means GPUs of a newer generation than Pascal (released in 2016) are not supported by DS-CUDA. Furthermore, the tool is not actively developed anymore.

vCUDA uses runtime API interception and redirection to provide GPU access to virtual machines.[16] Similarly to the previous tools, vCUDA redirects API calls of CUDA applications in the virtual machine to a server process running on the host which in turn forward them to the CUDA driver. The communication is implemented using RPC via shared memory, thus making access to GPUs on remote systems impossible. vCUDA tracks the memory objects passed between guest and host and translates pointer values to internal representations to allow a checkpoint/restart implementation. To improve performance, vCUDA uses lazy updates in which multiple CUDA API calls without side effects are combined to reduce the amount of communication between guest and host. The authors reported good compatibility with different applications. However, vCUDA only supports CUDA version 1.1 (released in 2007), which predates support for any data center grade GPUs. In addition, the source code of vCUDA is not available anymore.

CUDA multiprocess service (MPS) enables multiple GPU jobs to be executed concurrently, thus increasing GPU utilization compared with the case where only a single application may occupy a GPU at any given time.[17] MPS achieves this by replacing the CUDA APIs with a client-server structure, where client processes send GPU tasks to a server who manages the concurrent access to the GPUs. This mechanism is implemented at the driver API level, thus allowing the use of any higher level API (e.g., the runtime API, cuSolver, cuDNN). The fact that the amount of page-locked memory is limited by the available shared memory suggest MPS uses shared memory to transfer page-locked memory allocation between processes. MPS uses named pipes and domain sockets for the communication between server and client. Limitations of MPS include incomplete support for all CUDA features, a limited amount of client-server connections and only a simple job scheduler. Furthermore, MPS is not open-source thus making customization and reuse impossible.

Docker 19.03 introduces support for using NVIDIA GPUs inside Docker containers. This is achieved using a software layer that provides a driver version agnostic interface for the CUDA APIs. With this approach, the containerized application still directly interacts with the NVIDIA driver running on the host. Because of strong separation of the CPU and GPU parts of a CUDA application, Docker offers no proper isolation between containers and the host beyond limiting the use of specific GPUs.

The recent NVIDIA Ampere generation (released in 2020) introduces a feature called multiinstance GPU based on single-root I/O virtualization.[18] It enables separating a single GPU into up to seven isolated instances. However, this feature only allows increased isolation between CUDA applications and not a higher degree of control of GPU resources. Consequently, this features cannot be a basis for virtualization techniques, such as remote execution or checkpoint/restore support.

2.2 | Checkpoint/restart

For CPU applications there are already stable and well-established C/R tools available. A popular system-level C/R tool, specifically designed for the need of clusters is Berkeley Lab's Checkpoint/Restart (BLCR).[19] It requires the Linux kernel to load a kernel module and applications to link against a library. Thus, while not requiring code modifications, BLCR is not entirely transparent to application developers. The lack of support from the Linux kernel means that BLCR has to be adapted to most new kernel version. With the release of Linux kernel 3.11 in 2013, support for checkpoint/restart was added to the mainstream Linux kernel, thus making the approach of BLCR obsolete. Consequently, there has been no new release of BLCR since 2013[§].

CRIU [¶]builds upon the kernel support that allows the implementation of checkpoint/restart for CPU applications entirely in the user space. It requires no application modifications, injected libraries or custom launchers and is thus completely transparent to applications.

While BLCR and CRIU enable checkpoint/restart for CPU applications, they are unable to operate on CUDA applications, as they are unaware of the GPU and driver internal execution state. This means, clusters that use GPU accelerators are not able to benefit from checkpoint/restart. While checkpoint/restart for CPU applications is a well-researched topic, there are only a handful of checkpoint/restart tools that support GPU applications.

CheCUDA[20] is a prototype implementation of a plugin to BLCR that enables checkpoint/restart for CUDA applications. It installs a hook in the CUDA driver API function in order to record status changes to the API internal state. When a checkpoint is requested, all CUDA resources are deleted, and recreated after the checkpoint has been created using BLCR. CheCUDA wraps pointers to CUDA resources with an internal representation that allows updating the underlying memory pointers when they change after restoring an application. In their evaluation, the authors show significant execution time overhead introduced by their approach. They used CUDA toolkit version 2.2 (from 2009) for their evaluation.

HKC[21] is a system-level checkpoint/restart tool that allows the transparent checkpointing CUDA applications and even supports the creation of checkpoints during a kernel execution. When creating a checkpoint of a kernel, HKC only considers the state of threads that have either not yet started or have already finished execution, and not that of threads currently in execution. HKC still requires complete reexecution for threads that

[§]http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/
[¶]https://criu.org/

have not finished at the time of checkpointing, but saving the results of already finished warps reduces the time required to regain the execution progress. While in-kernel checkpoint/restart may be beneficial to long-running kernels, the incomplete consideration of threads that are in execution makes the value of the increased checkpoint and restart complexity doubtful. The developers of HKC report considerable runtime overhead for applications being checkpointed or restored. HKC uses CUDA toolkit version 3.2 (from 2010) for their evaluation.

Kang et al.[22] present a technique for migration and partial migration of GPU applications, where a kernel execution can be partially executed on a remote GPU to avoid delays due to memory shortage. They do not create checkpoints that can be restored at a later point in time, but focus on the migration of running applications. Similarly to Cricket, the authors use RPC-based virtualization to modify device interactions of GPU applications, however they support only OpenCL. Kang et al. do not publish the source code of their migration solution.

Parasysris et al.[23] extend the checkpoint/restart library FTI by enabling it to create application-level checkpoints for CUDA applications. The extended FTI creates checksums for GPU memory to support differential checkpoints, thereby decreasing the amount of memory transfers necessary. Because FTI requires source code modifications to support checkpoint/restart for a CUDA application, for example, by registering memory regions to be saved, the approach is not transparent. The extended FTI is available under an open source license and the authors show compatibility with NVIDIA GPUs from the Volta generation (released in 2017).

None of the previously discussed solutions represents an open-source virtualization solution for GPUs that supports the latest GPU generation. In this article, we present Cricket, a novel GPU virtualization tool that enables remote execution and checkpoint/restart between kernel executions, while supporting the latest GPUs and being released under an open-source license.[24] Because our approach uses CRIU to checkpoint and restart the part of a CUDA application that is unrelated to CUDA APIs, we also require the previously mentioned Linux Kernel features for checkpoint/restart support.

3 | GPU VIRTUALIZATION LAYER FOR CUDA APPLICATIONS

As previously discussed, a virtualization layer for GPUs has applications beyond remote execution and checkpoint/restart. Cricket aims at being a basis for future implementations of virtualization techniques other than those presented in this article. Therefore, a key requirement is that the code has to be published under an open-source license to allow researchers to reuse and build on top of the existing code. Furthermore, many scenarios require transparency or binary compatibility, that is, original CUDA application code cannot be required to be modified, as their source code might not be available. A transparent solution also means that applications remain unaware of the virtual nature of the execution environment. The improved flexibility and control introduced by virtualization is not allowed to come at the cost of large performance overheads. Another requirement is support for recent GPU architectures and CUDA toolkit versions.

GPU virtualization requires the insertion of a virtualization layer in the GPU software stack. The task of this virtualization layer is to separate the CUDA application from the real device and manage the interactions between both. The virtual device used by applications may differ from the real device, for example, they may be located on different machines or the computing resources of the virtual device may be limited. We achieve the separation by splitting the GPU and CPU parts of CUDA applications into separate processes. Instead of directly accessing GPU resources, CUDA applications use RPC to send requests to an RPC server that is responsible for the management of the interactions with the GPUs. Every CUDA application executed using Cricket launches exactly one RPC server with which it communicates.

The rest of this section introduces details about our implementation and the rationale behind design choices that had to be made.

3.1 | The CUDA software stack

CUDA offers multiple ways of interfacing applications with the GPU driver: High-level primitives, the runtime API and the lower level driver API. Figure 2(A) shows the different components involved in a compiled CUDA application built using the runtime API. At which level the virtualization layer is inserted requires careful consideration. The ideal point for this would be between driver API and NVIDIA driver, as this way all software layers above the driver would be unaware of the virtualization, thus achieving full transparency to any GPU application. However, due to the closed-source nature of the NVIDIA driver and the API implementations, there is not enough information available to implement this approach.

Going one layer up the software stack, the next possible point of separation is at the CUDA driver API. With this approach CUDA applications written with both driver and runtime can benefit from the virtualization layer. Furthermore, the virtualization layer does not have to be modified for every new runtime API version, as the driver API does not change as frequently. However, the runtime API uses parts of the driver API that are not documented by NVIDIA. For example, the undocumented driver API function cuGetExportTable exports a set of function pointers that implement hidden functionalities, which are used extensively by the runtime API. Properly isolating the runtime API from the driver API therefore necessitates intercepting also these hidden functions. However, as the undocumented nature of these functions prohibits proper intercepting. Consequently, a virtualization layer at the driver API level does not allow the use of the original runtime API on top of it.

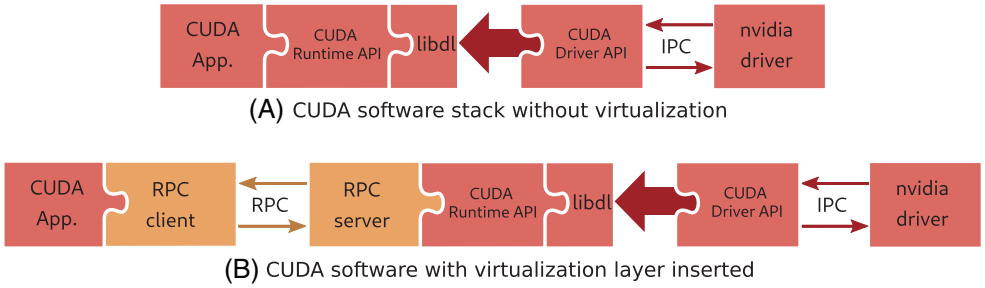


FIGURE 2 Comparison between original CUDA (A) and Cricket (B) software stacks. By default, the CUDA Runtime API is linked statically. It loads the Driver API dynamically during startup using `dlopen` and `dlsym`. Replacing the Runtime API with RPC code allows separating the CUDA application from the internal state of both Runtime and Driver APIs

Because of these complications, we opted to implement a virtualization layer that may be inserted at the levels of either the runtime API or the driver API. CUDA applications may use one of the virtualization layer positions depending on whether they use the runtime API or the driver API. While involving more work, this has the benefit of completely isolating the user code of the CUDA application from any internal state of the CUDA APIs, as interactions between the APIs are hidden behind the virtualization layer. For applications that use the runtime API, one API call often necessitates multiple driver API calls. Therefore, virtualization at the position of the runtime API requires less communication, making it beneficial for performance.

A prerequisite for the insertion of the virtualization layer is that the replaced CUDA API library is linked dynamically to the CUDA application. This is because with dynamically linked libraries the library code is loaded during the startup of the application, making the replacement of the library code possible. With statically linked libraries the library code is inserted into the application binary at compile time. The replacement of statically linked code requires techniques such as instrumentation, which introduce significant performance degradation.[25,26]

With dynamic linking, we can intercept the calls to library functions and replace them with our own code by replacing the linked object with a different one that exports the same symbols. The replacement involves either providing a shared object with the same name as the original library in the library search path, or using the LD_PRELOAD environment variable to load a library before all others, thus overwriting the symbols in the original library. Using this technique, we achieve the insertion of our virtualization layer by loading a replacement library that overwrites the function symbols of the original CUDA API libraries.

3.2 | Isolation with RPC

After library calls have been intercepted, they have to be forwarded to the real GPU. For this, an RPC server process waits for incoming GPU resource request from CUDA applications, executes them, and passes the results back to the original application. For each CUDA application, we start a separate RPC server process, thereby allowing a separation of server-side resources between different applications. Unlike other approaches, such as rCUDA, we execute the original API function even for the runtime API and do not reimplement the runtime API using the driver API. This makes the implementation of APIs in future CUDA releases less time-consuming. We use the transport independent remote procedure calls (TI-RPC) implementation of the RPC protocol specification version 2[27] as a basis for this communication. To support future CUDA versions, we need to only implement RPC wrappers for any new API functions and benefit from the code generation of TI-RPC, which creates corresponding RPC code. We register the RPC servers with the rpcbind service allowing RPC clients to find the correct RPC server even when multiple CUDA applications are running.

The replacement library that inserts the virtualization layer, uses a library constructor to set up the connection to the RPC server process. Our virtualization layer supports connections via either a domain socket or a TCP socket. A TCP connection enables the use of a GPU that is installed in a different system than where the CUDA application runs. The RPC server process is also realized by launching the CUDA application binary and loading a dynamic library at startup. The library constructor for the RPC server only waits for incoming RPC requests and never launches the original main function. By using a dynamic library instead of building a standalone server application, the server process has access to the GPU code, the code for launching kernels, and the CUDA initialization functions, which the CUDA compiler inserts into the application binary. Because the management of these resources is not documented, using the original binary enables us to initialize CUDA and launch kernels as if writing a normal CUDA application.

Some CUDA API functions return pointers to internal resources, for example, pointers to device memory and internal data structures. These pointers are only intended for passing to the CUDA APIs and user programs should not access the underlying data directly. Instead of collecting and copying the internal data structures from the RPC server process to the CUDA application, we pass only the raw pointer values, ignoring the fact that

they reference address spaces in a different process. This way, for most API functions the virtualization layer needs to transfer only pointer values for parameters and return values rather than also the referenced data. By contrast, the cudaMemcpy class of API functions is often used to transfer large amounts of application data between host and device memories. Using our RPC approach, we have to first copy this data from the CUDA application to the RPC server, which copies it to the GPU memory. When the CUDA application is launched on the same system as the RPC server, we can avoid this additional copy operation by using shared memory. Using Infiniband IBverbs, our virtualization layer is able to use RDMA in case the CUDA application and the RPC server execute on different systems. However, these optimizations require setting up the shared memory or RDMA memory segments during the allocation of the host memory from which a transfer originates. Therefore, increasing the transfer performance using shared memory or IBverbs only works for applications that allocate host memory using the cudaHostAlloc function, which is originally intended to request pinned memory from which CUDA can perform faster copy operations to device memory.

Figure 2(B) summarizes how requests to a virtual GPU occur. A CUDA API call in the CUDA application is redirected to our replacement library. The library implements all CUDA API functions with procedures that execute a RPC to the server process. There, the request is executed using the original API function of either the runtime or driver APIs. The server collects the results and sends them back to the CUDA application, where they are returned to the original program.

4 | CHECKPOINT/RESTART

On top of our virtualization layer Cricket, we implemented a checkpoint/restart functionality for CUDA applications.

We derived design goals for our checkpoint/restart implementation from those formulated by Hargrove and Duell[28] for CPU checkpoint/restart tools for HPC environments. Our checkpoint/restart implementation allows preemptive checkpoints, that is, the creation of checkpoints at any code position and without the cooperation of applications. This allows the immediate reaction to fault precursors or an unevenly distributed load across the nodes of a cluster. Only when a CUDA API is currently executing does Cricket have to wait for this to finish so a consistent state when restoring can be ensured. Furthermore, our implementation is transparent to application developers, requiring no source code modification to enable checkpoint/restart. The implementation is also performance transparent, that is it introduces no significant runtime overhead when no checkpoints are created. Especially for HPC applications that typically run over long periods of time, such overhead would increase application run times dramatically, which is unacceptable on clusters where computing time is valuable.

For applications that do not use GPU accelerators, checkpoint/restart tools, such as CRIU, already allow the reliable saving and restoring of the execution state, even when it is dependent on some external resources. However, when an application uses the CUDA APIs to execute code on a GPU, CRIU becomes unable to save and restore the kernel execution progress and the internal states of the CUDA APIs and the GPU driver. As the CUDA APIs and GPU driver are not open-source, modifying them to enable access to the internal state is not possible. Therefore, we split the checkpoint of a CUDA application in two parts, as shown in Figure 3: The client checkpoint that contains the execution state of the user code executing on the CPU, outside any CUDA API, and the server checkpoint that represents the internal state of the CUDA APIs.

When launching CUDA applications without a virtualization layer, there is no way of distinguishing what state information, such as memory segments, interprocess communication resources, and so forth, belong to CUDA APIs and which belong to the user code that uses these APIs. With our virtualization layer, the CUDA APIs are executed in a separate process from the user code. Thus, we find the state information for the client and server checkpoints likewise in the two respective processes. The only connection between both is the TCP-based RPC connection between the client and server applications. As this connection is under our control, we can repair it when an application is restored. Because of the virtualization layer, the client application is thus an application running only on the CPU with no direct driver interactions. The execution state of such an application may be saved and restored using CRIU without further modifications. CRIU is even able to repair the TCP connection to the server process if the server process is not stopped between checkpoint creation and restoring. If the server process has been restored, our virtualization layer notices this when trying to use the TCP socket for the first time after restoring. In this case, the virtualization layer reconnects server and client processes by consulting the rpcbind service to find the port the restarted server process is listening on.

Another important aspect is the management of API resources. Namely, GPU memory allocations and internal data structures that are returned by the CUDA APIs. We have to be able to recreate CUDA API resources during restoring of an application, while keeping the original references to these resources valid. The API resources are represented by pointers to either the GPU memory space or to CPU memory allocated and used by the APIs. The underlying data of these pointers should never be accessed in any user code. For GPU memory pointer, the referenced address is meaningless and in CPU memory the pointers reference data structures that are neither documented nor declared in the accessible CUDA API code. They are exclusively intended for passing to CUDA API functions. Therefore, these resources do not have to actually exist in the client process. User code merely has to be able to uniquely reference certain API resources with the pointer values returned by the CUDA API functions. Consequently, we can return and accept arbitrarily modified pointer values at the server side, and there, map these values to actual memory locations that CUDA understands. This way, the pointer values on the client side stay persistent across multiple launches, while the actual memory location of CUDA resources on the server side may change. To differentiate these different pointer value spaces, we subsequently call pointer values used in the client process *resource references* and pointer values that are meaningful to CUDA, *CUDA pointers*.

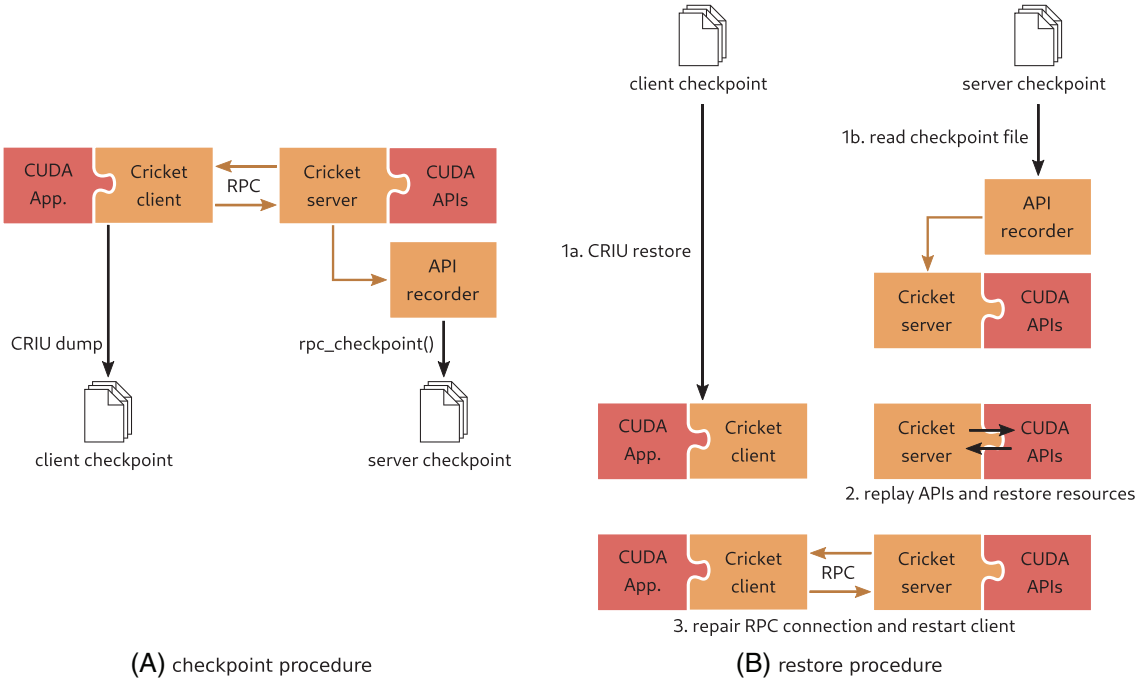


FIGURE 3 Illustration of the checkpoint creation and restoring of the two parts of a virtualized CUDA application

A disadvantage of separating these pointer value spaces is that it requires searching for the CUDA pointer that corresponds to a resource reference in a list of mappings each time a resource reference is used. This is reduces the performance of CUDA API calls. For this reason, we only use different pointer values for resource references and CUDA pointer when it is inevitable. That is only when a restored application is launched and already existing resources are rebuilt. Resources that are newly created during the first launch or after restoring, are assigned a resource reference that is identical to the CUDA pointer. This way, the list of mappings that have to be searched stays small and does not change after launching or restoring of an application. In addition, sorting the mappings allows the use of a binary search algorithm.

The CUDA APIs also exhibit internal state that is not visible when only interfacing with the API functions. For example, a call to `cudaSetDevice` would change the API state such that subsequent calls operate on a specific GPU. A following call to `cudaMalloc` would reserve memory on that particular GPU. To restore this internal state from a checkpoint, we use API replaying, that is reexecuting the CUDA APIs in the original order, for example, executing `cudaSetDevice`and `cudaMalloc` as before. For this, we log each CUDA API function call, including its parameters, during normal execution. When a server checkpoint is requested, we store the list of past CUDA API calls and the resource references to a checkpoint file. We can perform some optimizations to the list of API calls and resource reference, for example, remove memory allocations that have subsequently been freed. During restoring of the server checkpoint, we execute all remaining CUDA APIs stored in the checkpoint file. When a CUDA API recreates an API resource, we add the respective mapping from the resource reference retrieved from the checkpoint file to the new CUDA pointer returned by the API call. For example, a call to `cudaMalloc` during checkpoint will create a different pointer value than during the original execution. When a CUDA API is called with the original pointer value, Cricket replaces this with the new pointer value before forwarding the API call. After restoring is complete, the server process listens for a client process to reconnect and then continues to receive and process RPC requests.

5 | EVALUATION

Virtualization generally introduces overhead as a result of the additional code executed for the realization of a virtualization layer. For Cricket, we expect the virtualization layer to increase the latency and reduce the throughput of communication between CUDA applications and CUDA APIs because of the use of RPCs instead of regular function calls. This influences the time required for calling CUDA APIs and the achievable bandwidth for memory transfers between host and device memories. When using checkpoint/restart, the time in which the CUDA application is stopped to save the execution state and the time required for restoring application states are important metrics. In this section, we evaluate Cricket in terms of virtualization overhead introduced to CUDA applications for local and remote executions and analyze the checkpoint/restart performance.

For the evaluation we use one system equipped with two Intel Xeon Gold 6128 CPUs and Tesla P40 and Tesla T4 GPUs and one system with two AMD Epyc 7301 CPUs and no GPUs. Both systems are connected via two Infiniband 100 Gb/s links using Mellanox MCX556A-ECAT ConnectX-5

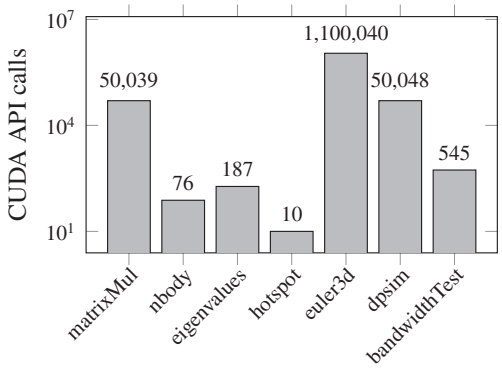
Adapter Cards. One link is configured for IP over Infiniband (IPoIB) and one is configured for RDMA communication. The Top500 List of the fastest supercomputers shows that interconnects with similar bandwidths are very common in clusters. Because of their widespread availability, we focus our evaluation on communication across the Infiniband interconnects. With these, the achievable bandwidth is near that of PCIe (128 Gb/s for the PCIe 3.0 16x slots we use) promising remote execution to be competitive with local executions. Since our evaluation system only consists of a cluster of two systems, we are unable to analyze the performance impact of interconnect congestion when many remotely executed GPU applications are running in parallel. While we confirmed the compatibility of Cricket with both the Tesla T4 and Tesla P40 GPUs, we chose the faster Tesla P40 for our evaluation because the performance impact of virtualization will be higher when GPU applications are executed faster. All measurements presented in this section have been performed using CUDA toolkit version 11.1. We execute the applications pinned to the NUMA domain where both, the GPU and the Infiniband device are connected. We compare the execution time of several applications when using our virtualization solution locally and remotely to the case where no virtualization is employed and measure the time required for creating and restoring checkpoints. In addition, we perform several microbenchmarks to assess potential sources of overhead.

5.1 | Benchmarks

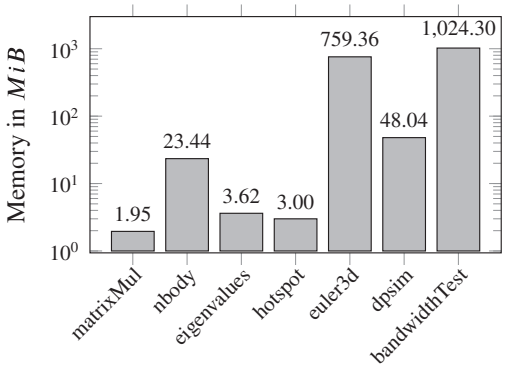
To analyze the virtualization overhead on CUDA applications, we evaluate Cricket with three of the example applications distributed with the CUDA Toolkit and three third-party applications. The *matrixMul* application performs a series of densely filled matrix-matrix multiplications without repeatedly copying data between host and device. It uses memory transfers using the pinned memory API. The *nbody* application is a physics simulation that computes the gravitational interaction between a configurable amount of bodies. *Eigenvalues* computes the eigenvalues of a tridiagonal symmetric matrix. *hotspot* from the Rodinia Benchmark Suite[29] is a thermal simulation application that solves differential equations. The *euler3d* application from the Rodinia Benchmark Suite is a finite volume solver on unstructured grids for the three-dimensional Euler equations. *DPsim* is a power system simulator for dynamic phasor and electromagnetic transient simulations.[30] In contrast to the other applications, it does not use the CUDA runtime API but the cuSOLVER API. For the evaluation of attainable bandwidths between CPU and GPU memories, we employ the *bandwidthTest* application from the CUDA samples. The variety of considered applications shows the real-world applicability of Cricket. While Cricket does not yet support every CUDA API function, for example, the CUDA graph API and some functions concerning texture memory are currently not supported, neither the applications presented here nor any other we used for verification uses unsupported API functions.

Since the overhead introduced to individual applications is dependent on both the number of performed CUDA API calls and the amount of memory transferred between CPU and GPU memories, we measure these values for the considered applications. Figure 4 shows the varied behavior of these applications. While *hotspot* only performs 10 API calls and transfers 3 MiB of data, *euler3d* performs more than 1.1 million API calls and transfers 759.36 MiB of data.

In Figure 5, we compare the reference execution time of the applications without virtualization with the execution time when Cricket is used for virtualization. We measure the time required for a local execution using a loopback TCP connection for data transfers and a remote execution using a TCP connection between two systems. The data shows a similar execution time for the three cases with all applications. An execution using Cricket locally does not introduce significant overhead. In contrast to our previous work,[12] where we obtained worse results for hotspot, improvements to Cricket reduced the virtualization overhead for this application to a negligible amount. For all applications, the execution time of memory transfers and kernels outweighs any increased latency of CUDA API calls. Because of the asynchronous nature of these two types of API functions, even

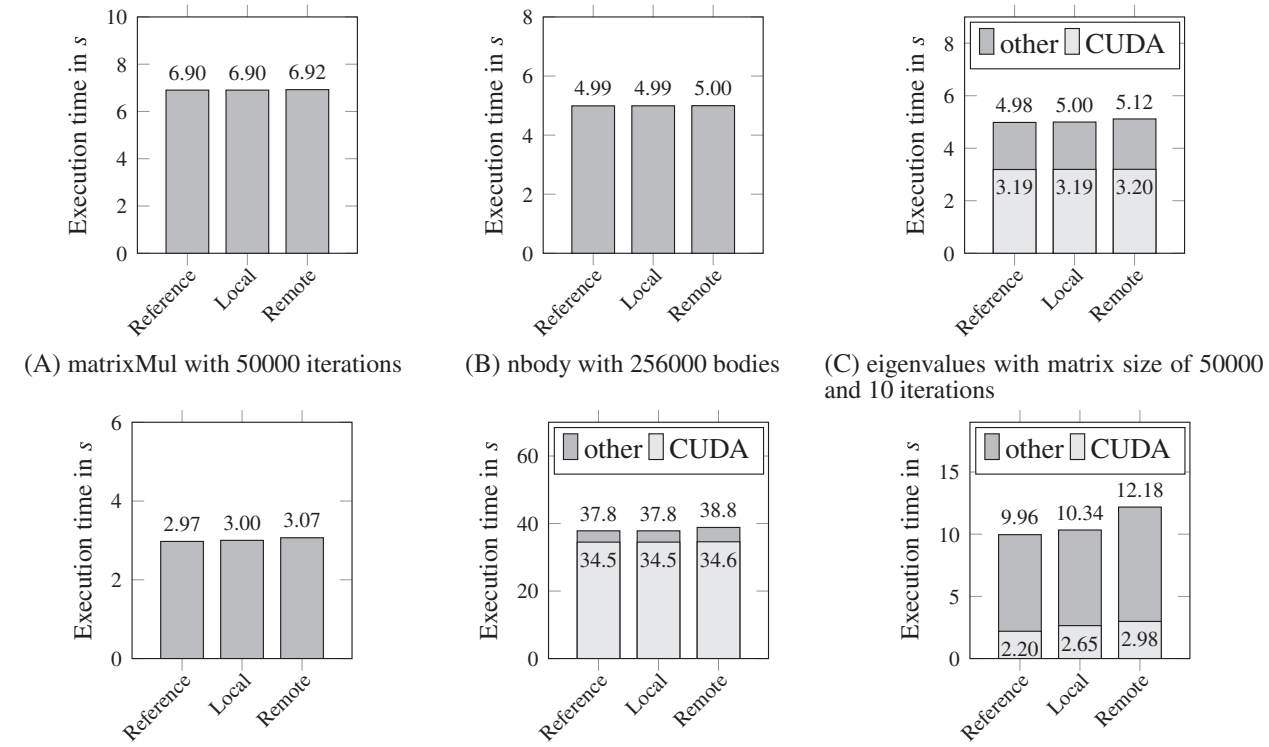


(A) Number of CUDA API calls performed by the applications.



(B) Amount of memory transferred between host and device memories.

FIGURE 4 The application chosen for the evaluation have varied behaviors that expresses itself in different numbers of performed CUDA API calls and amounts of transferred memory



(A) matrixMul with 50000 iterations (B) nbody with 256000 bodies (C) eigenvalues with matrix size of 50000 and 10 iterations

(D) hotspot with a gridsize of 512 by 512 (E) euler3d with the missile dataset (F) DPsim simulation of the 9-bus example (10 copies)

FIGURE 5 The execution times of the example applications based on 10 averaged runs show only a small overhead when we use Cricket

applications with a very large numbers of API calls such as euler3d, are not slowed down to a large degree. When an application, such as euler3d and matrixMul, performs many consecutive asynchronous API calls, the increased latency of individual API functions is effectively hidden when the application afterward has to wait for the finishing of the asynchronous operations.

Remote executions via the IPoIB connection of the considered applications achieve performance comparable to local executions for most applications. The increased latency and decreased bandwidth for remote communication does not impact the performance of the applications to a large amount. For euler3d, DPsim, and eigenvalues, the remote execution is visibly slower than the reference case. To investigate the cause of this behavior, we inserted code into these applications that measures the time spent only on CUDA API functions. Figure 5 shows that most of the overhead of remote execution is not due to these function. Instead, the applications euler3d, DPsim, and eigenvalues also have CPU-intensive computations, which are slower when executed remotely, because in our evaluation setup the remote CPU is slower than the local one. Only for DPsim, there is a pronounced differences between the reference, local and remote cases in terms of time spent on CUDA API functions. This is due to DPsim exchanging data between GPU and CPU memories very frequently, requiring synchronization and thereby not being able to benefit from latency hiding due to overlapping executions.

5.2 | Microbenchmarks

To quantify the impact the virtualization layer has on the execution time of CUDA API calls, we measure the latency of two typical CUDA API functions in different virtualization scenarios (see Figure 6). cudaMalloc allocates a region of device memory and represents a commonly used API function. cudaGetDeviceCount returns the number of available GPUs, and as such, this function causes the transfer of only one single integer. Hence, almost all observable latency is caused by communication delays.

Euler3d does not show significant virtualization overhead despite performing over a million CUDA API calls because many API calls are asynchronous. Thus, additional latency is hidden from the application when server and client perform execution progress in parallel. For both API functions we measure an overhead between 23.33 and 26.12 µs when using the virtualization layer. This is caused by the redirection API calls that require the execution of additional code, the copying of parameters, and copying of results. While the impact of the virtualization layer on individual functions is comparatively large, most applications do not perform a high number of CUDA API calls. For example, the previously considered applications nbody, eigenvalues hotspot and DPsim perform 72, 10, and 72 API calls, respectively. Only matrixMul performs a higher amount of 10,033,

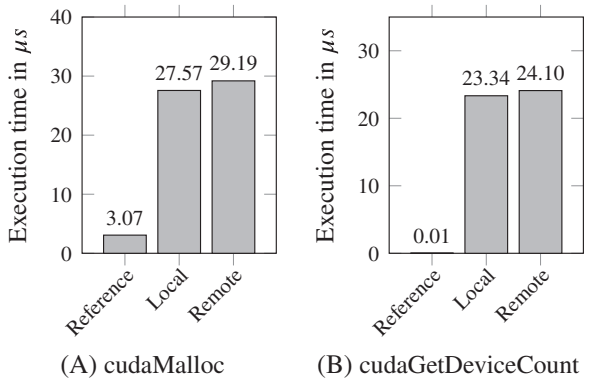


FIGURE 6 Cricket increases the latency of the CUDA API calls cudaMalloc and cudaGetDeviceCount by approx. 23–26 µs

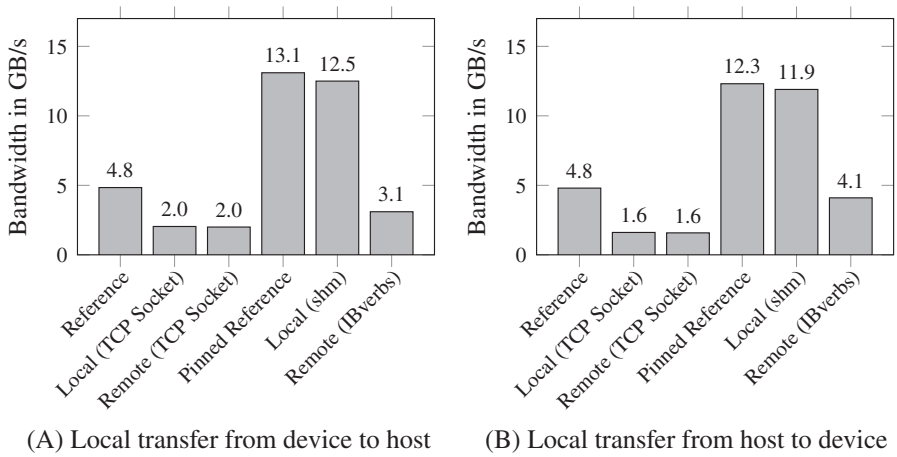


FIGURE 7 Cricket influences the memory transfer bandwidth of transfers between host and device memories

while still showing only a small overhead caused by the virtualization layer (see Figure 5(A)). Instead of performing more calls to CUDA functions with increasing problem sizes, most applications require the transfer of more data between host and device memory. Therefore, the CUDA API functions responsible for transferring data between host and device memory also require an analysis.

Figure 7 shows the achieved memory transfer bandwidth for the reference case and with our virtualization layer using local and remote communication. We obtain the results by using the bandwidthTest application from the CUDA samples, which reports the bandwidth for memory transfers from and to the GPU using either pinned or nonpinned host memory. The reference measurements that we perform without using Cricket, show that transfers from and to pinned host memory are more than 2.5 times faster than from and to nonpinned memory. When using Cricket, the virtualization layer decreases the transfer bandwidth. For transfers involving nonpinned host memory, the bandwidth is reduced to 42% for device to host transfers and 33% for host to device transfers, regardless of whether the Cricket server and client applications reside on the same system. This suggests, that the bandwidth of our network interconnect and of the loopback network interface are not limiting the transfer bandwidth with the GPU. Instead, the reduced performance is a result of the additional data transfer between CUDA application and RPC server.

When memory transfers involve pinned host memory, Cricket avoids this additional data transfer using shared memory or RDMA using IBverbs. For the local case, where shared memory is used for the data transfer, the achieved bandwidth is comparable to the reference case without virtualization. When client and server reside on different computing nodes, Cricket uses RDMA for data transfers. In this case, the achieved bandwidth remains below the interconnect capacity and lower than in the reference case without virtualization. While further optimization efforts into increasing the achieved bandwidth seem promising, the application evaluation showed already low additional overhead on the overall execution time for remote execution compared with local execution.

5.3 | Checkpoint/restart

For checkpoint/restart, important metrics are the time required for creating checkpoints and for restoring them. When used for increasing fault tolerance, checkpoints are typically created more regularly than restored, making the time required for the checkpoint procedure more important.

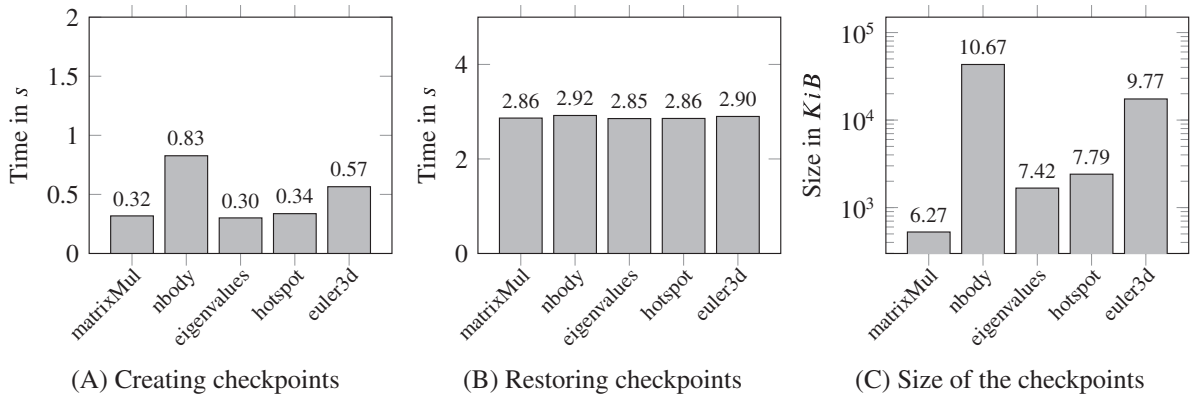


FIGURE 8 The time required for the checkpoint/restart procedures (10 averaged runs) correlates with the checkpoint size that differs for each application

When used for the migration of applications, a created checkpoint is copied to a different node where it is restored. In this scenario both: checkpoint creation and restore times and additionally the checkpoint size are important values.

Figure 8 shows the time required to checkpoint and restore the example applications[#]. Figure 8(C) shows the size of the created checkpoints as a sum of the checkpoint created by CRIU and the checkpoint created by Cricket. The creation of checkpoints requires between 0.3 and 0.83 s, while restoring requires between 2.85 and 2.92 s. Cricket takes longer for restoring than for checkpointing since the CUDA API has to be first initialized. We measure an average of 2.36 s for this initialization, which explains much of the size of the values in Figure 8(B). The differences of checkpoint and restore times between applications correlates with the size of the checkpoints. This relation is more obvious for the checkpoint creation than for restoring because the constant time required for the CUDA API initialization makes the differences in Figure 8(B) less pronounced. The checkpoint size is determined mainly by the reserved host and device memory at the time of checkpointing. The nbody application shows the largest checkpoint size, and also the largest checkpoint and restore times. When restoring, we can overlap the restoring of device memory with the replaying of CUDA APIs, thereby reducing the impact of the checkpoint size on the restore time compared with the creation of the checkpoint.

When used for fault-tolerance the comparably large restore times are not a problem, as restoring is only performed in case of a fault. However, for migration an application would stop execution for the sum of checkpoint and restore times[‖]. Stopping execution for up to 3.75 s is too much for the here considered application run times, because in many cases it would be better to restart the applications in case of faults, instead of using checkpoint/restart. Future improvements to cricket should include performing the initialization of the CUDA API before checkpoint creation. This would trivially reduce the time an application is stopped to below 1.4 s. Another improvement enabled by the control provided by our virtualization layer could be incremental checkpointing, which would further improve the migration performance.

6 | CONCLUSION

Cricket adds a virtualization layer to CUDA applications that enables remote execution and checkpoint/restart. Despite the closed source nature of CUDA APIs and the GPU driver, Cricket achieves full control of the interactions between CUDA applications and GPU devices. The virtualization layer is transparent to applications, keeping them unaware of the indirect interaction with the GPU. As our evaluation shows, Cricket introduces only a small performance overhead to the CUDA applications, proving the real-world usability.

Cricket enables us to implement remote execution making the execution of CUDA applications possible on system without a GPU by instead employing a device installed in a remote system. The virtualization overhead when using remote execution stays low despite the introduced inter-system communication. This shows that with modern high speed interconnects, such as Infiniband, the data transfer between systems caused by remote execution has only a small performance impact. Our checkpoint/restart implementation allows saving and restoring the execution progress of CUDA applications preemptively and transparently. That is, checkpoints may be created at any point during the execution, while the application code does not have to be modified.

Cricket is open-source, thus providing a basis for future research and implementations of GPU virtualization techniques, such as custom GPU task schedulers, fault detection mechanism, or migration strategies.

[#]The checkpoint/restart implementation does not yet support cuSOLVER, thus not supporting checkpoint/restart for DPsim.
[‖]Because we stored checkpoints on a network drive, copying the checkpoint is not necessary

ACKNOWLEDGMENTS

This research and development has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957246 and the German Federal Ministry of Education and Research under Grant 01IH16010C (Project ENVELOPE). Open Access funding enabled and organized by Projekt DEAL.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in the Cricket repository at https://github.com/RWTH-ACS/cricket, reference number 24.

ORCID


Niklas Eiling https://orcid.org/0000-0002-7011-9846

REFERENCES

1. Baker ZK, Gokhale MB, Tripp JL. Matched filter computation on FPGA, cell and GPU. Paper presented at: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007). Napa, CA, USA; 2007:207-218. https://doi.org/10.1109/FCCM.2007.52.
2. Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. *Supercomputing*. 2004. Proceedings of the ACM/IEEE SC2004 Conference 2004: 47-47. ;https://doi.org/10.1109/SC.2004.26
3. Li H, Ota K, Dong M, Vasilakos AV, Nagano K. Multimedia processing pricing strategy in GPU-accelerated cloud computing. *IEEE Trans Cloud Comput*. 2020;8(4):1264-1273. https://doi.org/10.1109/TCC.2017.2672554
4. Hong CH, Spence I, Nikolopoulos DS. GPU virtualization and scheduling methods: a comprehensive survey. *ACM Comput Surv*. 2017;50(3):1–37. https://doi.org/10.1145/3068281
5. Milojičić DS, Douglis F, Paindaveine Y, Wheeler R, Zhou S. Process migration. *ACM Comput Surv*. 2000;32(3):241-299. https://doi.org/10.1145/367701.367728
6. Gavrilovska A, Kumar S, Raj H, et al. High-performance hypervisor architectures: Virtualization in HPC systems. Paper presented at: Proceedings of the Workshop on system-level virtualization for HPC (HPCVirt); 2007; Lisbon, Portugal.
7. Silla F, Prades J, Iserte S, Reaño C. Remote GPU virtualization: is it useful? Paper presented at: Proceedings of the 2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB); 2016:41-48; Barcelona, Spain.
8. Elnozahy ENM, Alvisi L, Wang YM, Johnson DB. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput Surv*. 2002;34(3):375-408. https://doi.org/10.1145/568522.568525
9. NVIDIA Corporation. *NVIDIA(R) CUDA(TM) Architecture. Technical Report*. Santa Clara, CA: NVIDIA Corporation; 2009 Online; Accessed May 10, 2020.
10. Karimi K, Dickson NG, Hamze F. A performance comparison of CUDA and OpenCL. *arXiv e-prints*. 2010;arXiv:1005.2581.
11. NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical Report*. Santa Clara, CA: NVIDIA Corporation; 2009 Online; Accessed May 10, 2020.
12. Eiling N, Lankes S, Monti A. An open-source virtualization layer for CUDA applications. Paper presented at: Proceedings of the Euro-Par 2020: Parallel Processing Workshops. Warsaw, Poland; Vol. 12480, 2021:160-171. https://doi.org/10.1007/978-3-030-71593-9_13
13. Duato J, Peña AJ, Silla F, Mayo R, Quintana-Ortí ES. rCUDA: reducing the number of GPU-based accelerators in high performance clusters. Paper presented at: Proceedings of the 2010 International Conference on High Performance Computing Simulation; 2010:224-231; Caen, France. https://doi.org/10.1109/HPCS.2010.5547126
14. Reaño C, Silla F. A performance comparison of CUDA remote GPU virtualization frameworks. Paper presented at: Proceedings of the 2015 IEEE International Conference on Cluster Computing; 2015:488-489; Chicago, IL, USA. https://doi.org/10.1109/CLUSTER.2015.76
15. Oikawa M, Kawai A, Nomura K, Yasuoka K, Yoshikawa K, Narumi T. DS-CUDA: a middleware to use many GPUs in the cloud environment. *Proceedings of the 2012 SC Companion: High Performance Computing Networking Storage and Analysis*; 2012:1207-1214. https://doi.org/10.1109/SC.Companion.2012.146.
16. Shi L, Chen H, Sun J, Li K. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans Comput*. 2012;61(6):804-816.
17. NVIDIA Corporation *Multi-Process Service. Technical Report*, Santa Clara, CA: NVIDIA Corporation 2019. Online; Accessed May 04, 2020.
18. NVIDIA Corporation *NVIDIA A100 Tensor Core GPU Architecture. Technical Report*. Santa Clara, CA: NVIDIA Corporation; 2020. Online; Accessed December 14, 2020.
19. Duell J. The design and implementation of Berkeley Lab's linuxcheckpoint/restart. Lawrence Berkeley National Laboratory; 2005. https://doi.org/10.2172/891617
20. Takizawa H, Sato K, Komatsu K, Kobayashi H. CheCUDA: a checkpoint/restart tool for CUDA applications. Paper presented at: Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies; 2009:408-413; Higashi-Hiroshima, Japan.
21. Shi L, Chen H, Li T. Hybrid CPU/GPU checkpoint for GPU-based heterogeneous systems. In: Li K, Xiao Z, Wang Y, Du J, Li K, eds. *Parallel Computational Fluid Dynamics*. Berlin/Heidelberg, Germany: Springer; 2014:470-481.
22. Kang J, Lim J, Yu H. Partial migration technique for GPGPU tasks to prevent GPU memory starvation in RPC-based GPU virtualization. *Softw Pract Exper*. 2020;50:948-972. https://doi.org/10.1002/spe.2801
23. Parasyris K, Keller K, Bautista-Gomez L, Unsal O. Checkpoint restart support for heterogeneous HPC applications. Paper presented at: Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID); 2020:242-251; Melbourne, VIC, Australia. https://doi.org/10.1109/CCGrid49817.2020.00-69
24. Eiling N. RWTH-ACS/cricket; cricket source code repository, 2021. https://github.com/RWTH-ACS/cricket.
25. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. Paper presented at: Proceedings of the ACM Association for Computing Machinery PLDI '07; San Diego, CA, USA; 2007:89-100; ACM, New York, NY.
26. Laurenzano MA, Tikir MM, Carrington L, Snavely A. PEBIL: efficient static binary instrumentation for Linux. Paper presented at: Proceedings of the2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS); 2010:175-183; White Plains, NY, USA.

27. Srinivasan R. RPC: remote procedure call protocol specification version 2. IETF; 1995.
28. Hargrove PH, Duell JC. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *J Phys Conf Ser*. 2006;46(1):494.
29. Che S, Boyer M, Meng J, et al. Rodinia: a benchmark suite for heterogeneous computing. Paper presented at: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC). Austin, TX, USA; 2009:44-54.
30. Mirz M, Vogel S, Reinke G, Monti A. DPsim—a dynamic phasor real-time simulator for power systems. *SoftwareX*. 2019;10:100253. https://doi.org/10.1016/j.softx.2019.100253

How to cite this article: Eiling N, Baude J, Lankes S, Monti A. Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. *Concurrency Computat Pract Exper*. 2022;34(14):e6474. doi: 10.1002/cpe.6474