GPU Acceleration in Unikernels Using Cricket GPU Virtualization

Niklas Eiling
niklas.eiling@eonerc.rwth-aachen.de
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany

Martin Kröning
martin.kroening@eonerc.rwth-aachen.de
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany

Jonathan Klimt
jonathan.klimt@eonerc.rwth-aachen.de
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany

Philipp Fensch
philipp.fensch@rwth-aachen.de
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany

Stefan Lankes
slankes@eonerc.rwth-aachen.de
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany

Antonello Monti
amonti@eonerc.rwth-aachen.de
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany

ABSTRACT

Today, large compute clusters increasingly move towards heterogeneous architectures by employing accelerators, such as GPUs, to realize ever-increasing performance. To achieve maximum performance on these architectures, applications have to be tailored to the available hardware by using special APIs to interact with the hardware resources, such as the CUDA APIs for NVIDIA GPUs. Simultaneously, unikernels emerge as a solution for the increasing overhead introduced by the complexity of modern operating systems and their inability to optimize for specific application profiles. Unikernels allow for better static code checking and enable optimizations impossible with monolithic kernels, yielding more robust and faster programs. Despite this, there is a lack of support for using GPUs in unikernels. Due to the proprietary nature of the CUDA APIs, direct support for interacting with NVIDIA GPUs from unikernels is infeasible, resulting in applications requiring GPUs being unsuitable for deployment in unikernels.

We propose using Cricket GPU virtualization to introduce GPU support to the unikernels RustyHermit and Unikraft. To interface with Cricket, we implement a generic library for using ONC RPCs in Rust. With Cricket and our RPC library, unikernels become able to use GPU resources, even when they are installed in remote machines. This way, we enable the use of unikernels for applications that require the high parallel performance of GPUs to achieve manageable execution times.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Operating systems**; **Distributed systems organizing principles**.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624236

KEYWORDS

GPUs, Unikernel, High-Performance Computing, Cloud Computing, Heterogeneous Computing, CUDA, Virtualization, RustyHermit, Unikraft

ACM Reference Format:
Niklas Eiling, Martin Kröning, Jonathan Klimt, Philipp Fensch, Stefan Lankes, and Antonello Monti. 2023. GPU Acceleration in Unikernels Using Cricket GPU Virtualization. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3624062.3624236

1 INTRODUCTION

The advent of heterogeneous computing leads to new challenges in the management and development of software for high-performance clusters. Today, many applications, e.g., from the fields of machine learning and big data, require the high parallel performance and energy efficiency of GPUs. When introducing GPUs to clusters, they are often distributed homogeneously across the nodes. However, this leads to reduced utilization when the ranks of a clustered application cannot achieve full occupancy of the local GPUs. Consequently, clusters can only achieve high utilization when the application profile fits the cluster structure. Depending on what resources, such as CPU, GPU, and memory, are locally available and to what degree they can be utilized by applications, highly influences the computational throughput. Maintaining and improving flexibility and utilization while continuing to increase the amount of heterogeneous computing resources is the main challenge of future high-performance systems.

At the same time, the increasing complexity and diversity of operating systems and applications lead to operating system overhead becoming more noticeable and making application security and safety increasingly difficult to guarantee. Unikernels show potential to tackle this challenge by providing customizable, lightweight, and robust operating systems that allow optimizing the kernel itself for the application, thereby improving performance and predictability [13, 15]. The reduced complexity allows unikernels and applications to be compiled at the same time, enabling static code

Figure 1: Cricket inserts a virtualization layer into GPU applications

checking and optimizations otherwise impossible. While there is some literature on using unikernels in HPC and cloud environments, there is a lack of support for GPU accelerated applications running in unikernels. Thus, while many data-intensive applications rely on the parallel performance of GPUs, unikernels are unable to fully utilize this resource, thereby decreasing the available peak performance and utilization in clusters. To address the lack of GPU support by unikernels, this paper shows that GPU support can be integrated into unikernels. We use our previous work on the RPC-based GPU virtualization tool Cricket [8] to implement support for GPU applications in the RustyHermit[13] and Unikraft[12] unikernels. For this, we built a new Rust based ONC RPC [22] implementation, which allows interaction with Cricket while having a smaller dependence on Linux network features than the *libtirpc* implementation used previously. This way, unikernels become able to use CUDA APIs, which are forwarded via RPCs to a Cricket Server that interacts with the GPU devices.

Using GPUs in unikernel applications leads to similar challenges as using GPUs in Linux VMs. While GPUs allow sharing computing resources between applications, sharing GPUs across driver instances is only possible to a limited degree; E.g., the A100 GPU supports partitioning using SR-IOV, but only allows for seven such partitions [17]. Furthermore, this technology requires duplication of the drivers in each Linux VM or unikernel. The assignment of entire GPUs or partitions exclusively and statically to a virtual environment can be feasible for Linux VMs. However, for unikernels such an assignment is inefficient because they are typically deployed in larger numbers and only execute a single application each.

In contrast, Cricket GPU virtualization allows remote access to GPUs that can be located anywhere in the cluster and whose assignments to applications can dynamically adapt to the current workloads. Because this leads to more flexibility and allows for a larger number of unikernels to be deployed, Cricket fits better to the requirements of GPU applications running in unikernels. To achieve GPU virtualization, Cricket inserts a virtualization layer between GPU applications and the CUDA libraries used to interact with the GPU (see Figure 1). The virtualization layer forwards calls to the CUDA APIs via RPC to the Cricket server running either locally or remotely on a different machine. With Cricket, GPU applications are split into two processes, separating the application from the proprietary CUDA APIs that require access to a GPU. This way, structuring clusters in a more flexible manner becomes possible. Figure 2 shows an example, where nodes, which do not have GPUs, run GPU applications by using Cricket to provide virtual GPUs to applications. The physical GPU devices are located in dedicated GPU nodes, where the Cricket server makes them available for the entire cluster.

In previous work, we have shown the benefits of the decoupling provided by Cricket on the device side, i.e., increasing the flexibility

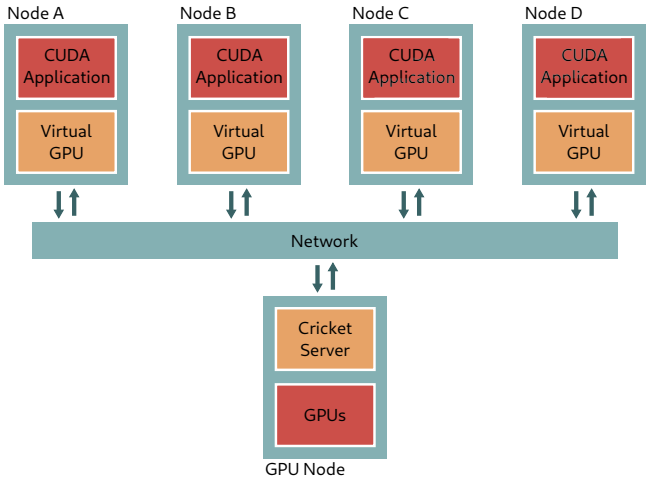


Figure 2: Conceptual overview of the Cricket virtualization layer. CUDA applications running on one of the nodes A-D can access GPUs on a dedicated GPU node via remote procedure calls.

of device assignments, providing Checkpoint / Restart support, and providing remote execution capabilities [8].

In this paper, we build on the virtualization layer provided by Cricket, and implement RPC-Lib, a Rust library for the automatic generation of RPC client code that interfaces with the virtualization layer of Cricket. Our implementation relies on the Rust standard library to abstract from the Linux system calls. The previous Cricket client code used libtirpc, which heavily relies on various Linux-specific APIs, making it difficult to support by unikernels. Thus, our Rust implementation simplifies the interface to unikernels, enabling them to participate alongside other Rust and C applications in a Cricket-enabled GPU cluster. To show the feasibility of this approach, we use RPC-Lib and Cricket to build GPU application running in RustyHermit and Unikraft unikernels. For this, we built and upstreamed support for executing Rust applications in Unikraft. For RustyHermit, this was not necessary because this support was already present.

The rest of the paper is structured as follows: In Chapter 2, we present previous work related to our approach. Chapter 3 deals with the details of our implementation of Cricket, the Rust RPC library, and how the GPUs can be interacted with from RustyHermit and Unikraft unikernels. An evaluation follows in Chapter 4, showing the applicability of our approach using micro-benchmarks and proxy applications. The paper is concluded with a discussion of our insights in Chapter 5.

2 RELATED WORK

There has been previous work in the fields related to our work: unikernels, GPU virtualization, and CUDA libraries in Rust. While we use RustyHermit and Unikraft as unikernels, Cricket as a GPU virtualization solution, and created a novel CUDA interface for Rust, our literature review showed several alternatives to these choices.

MirageOS [15] is a unikernel for OCaml applications targeting cloud environments. It promises robustness by having both unikernel and application code type-checked as a single unit. MirageOS

uses hypervisor-provided device abstractions to access hardware such as Ethernet and storage devices. Because it lacks hypervisor support, it does not support using GPUs for computation.

Raza et al. propose including applications into the Linux kernel by executing the application code from within kernel space [21]. This allows using optimizations similar to those offered by unikernels while preserving the broad soft- and hardware support and ecosystems of the Linux kernel. While this approach also allows using the hardware support of the Linux drivers, the authors report issues interacting with GPUs.

rCUDA and HFGPU are both GPU virtualization solution based on intercepting CUDA APIs and making them available remotely [10, 11]. They allow the distributed execution of GPU applications while also being able to utilize high-performance interconnects that are typically found in compute clusters. Furthermore, rCUDA features integration into cluster-wide scheduling and HFGPU uses an I/O forwarding mechanism to accelerate access to distributed files systems. While rCUDA is distributed under a proprietary license, making interoperability with and integration into other products impossible, the source code of HFGPU is distributed under an open-source license.

With vAccel [16], Unikraft applications can use Jetson Inference, a GPU-accelerated high-level inference library. vAccel uses a virtio plugin to forward API functions to the hypervisor, which can execute them as if called from the host. While similar to our approach but for a higher-level API, forwarding calls via the hypervisor is less flexible because it requires hypervisor modifications and does not allow remote execution.

NVIDIA vGPU is a product offered by NVIDIA itself that allows subdividing supported GPUs so that they can be used by multiple virtual machines simultaneously [18]. While this technology allows the sharing of GPUs, the assignment of GPUs to applications remains static and does not allow control over the interactions between applications and GPUs.

Interacting with GPUs via CUDA from Rust code requires bindings to interface with the C++ APIs provided by NVIDIA. The Rust CUDA Project [5] provides such a CUDA interface that aims to be intuitive for Rust developers. It also allows compiling Rust code to CUDA kernels by using LLVM and the NVIDIA PTX compiler, so that even GPU code can be written in Rust. Currently, the Rust CUDA project supports a large subset of the available CUDA libraries.

With *onc_rpc* there is another implementation of the ONC RPC interface used by Cricket [1]. However, the implementation lacks support for fragmented messages, meaning that function parameters are limited in size. Cricket requires support for large parameters because they are used to transferring data to the GPU. Furthermore, the project is currently not actively developed, with the last contribution being made in mid-2021.

3 GPU VIRTUALIZATION FOR UNIKERNELS

For this paper, we chose Cricket for the GPU virtualization because of its open-source code base and support for the latest generation of GPUs. We selected RustyHermit [13] and Unikraft [12] as unikernels because they capture different unikernel design strategies. Unikraft mimics Linux and tries to be a drop-in replacement by aiming for full compatibility with existing Linux applications on all levels. RustyHermit is a language-specific Unikernel with an independent API, written in Rust, and mainly targeting Rust applications. We created a new ONC RPC library for Rust because of the unavailability of an existing implementation that fulfills our requirements.

3.1 RustyHermit

RustyHermit [13] is a language-specific Rust unikernel focused on research. It is derived from the HermitCore unikernel, which was written in C [14]. The adoption of the Rust language made development easier, as Rust has many modern language features due to its young age. One of the main strengths of Rust are its strong safety guarantees, enabled through the strong type system and the borrow checker, which validates references at compile time.

As RustyHermit mainly supports Rust applications, Rust's benefits translate to the whole unikernel image. RustyHermit is designed to run on hypervisors, such as the widely used QEMU and the RustyHermit-specific hypervisor Uhyve. It supports the x86-64 and AArch64 architectures, and multiprocessing. On QEMU, the preferred network interface is the virtio network device.The TCP stack of Hermit is implemented using smoltcp [6]. Previous evaluations of RustyHermit showed lower memory footprint, disk overhead, and system call latencies when compared to a Linux VM [13].

For this paper, we heavily improved the performance of the network stack by implementing support of several new virtio features such as `VIRTIO_NET_F_GUEST_CSUM` and `VIRTIO_NET_F_CSUM`, for avoiding creating additional IP packet checksums, and `VIRTIO_NET\_F_MRG_RXBUF` for merging receive buffers. We also reduced the amount of internal copies, which helped to increase the network bandwidth.

3.2 Unikraft

Unikraft [12] is a young and quickly growing unikernel development kit. The build system of Unikraft allows fine-grained specialization of the kernel to the application and the virtual machine. The core of Unikraft consists of over fifty libraries, which can be configured individually and combined freely. On top of this foundation, library ports of standard libraries and network stacks, e.g., *musl* and *lwIP*, satisfy most application needs. Unikraft aims at matching the API and behavior of Linux to support the large pool of Linux applications without adjustments.

Unikraft can run applications in different modes: intercepting raw system calls by using the technique from [20], intercepting calls to C standard library functions, and recompiling the application. For maximum performance, we opted for the last option, compiling our application for Unikraft specifically. We have previously developed Unikraft support for Rust[1], reusing the Rust support for *musl* on Linux. We have also added support for integrating the Unikraft build system with external compilers[2,3], such as the Rust compiler. This allows us to make the compilation of our Rust applications for Unikraft as straightforward as possible.

[1] https://github.com/rust-lang/rust/pull/113411
[2] https://github.com/unikraft/unikraft/pull/957
[3] https://github.com/unikraft/kraftkit/pull/703

(a) Cricket with RustyHermit client **(b) Cricket with Unikraft client**

Figure 3: Cricket inserts a virtualization layer between CUDA applications running in unikernels and the CUDA APIs.

3.3 Virtualization of GPUs using Cricket

Traditional GPU applications in clusters use only locally installed GPUs. Even in virtualized execution environments, the assignment of GPUs to applications is static and cannot change during execution. To solve this inflexibility, the GPU virtualization layer Cricket uses RPCs to decouple applications from the execution of CUDA APIs, thereby allowing better sharing of GPU resources and remote execution [8]. This enables a more flexible cluster architecture, as shown in Figure 2, where all GPUs in a cluster can be used by any application, regardless of the locality of the applications. This way, GPUs can be fully utilized, even if applications provide non-uniform loads during their execution. Furthermore, being able to fully control the interactions between applications and GPUs enables employing custom scheduling strategies and Checkpoint / Restart.

The decoupling provided by Cricket is also beneficial for introducing GPU support to architectures that are unable to use the binary distributions of CUDA provided by NVIDIA. Instead of requiring a CUDA implementation, Cricket only requires the implementation of the ONC RPC interface. The Cricket server executes the CUDA APIs and forwards the results back to the application. While we previously showed the low virtualization overhead of Cricket when using C applications [8], in this paper we use Cricket with Rust applications running in RustyHermit unikernels (see Figure 3).

While many applications use CUDA libraries such as cuSolver, cuBLAS, or cuFFT that contain kernels for common GPU operations, applications can also include custom kernels. The NVIDIA C Compiler (NVCC) compiles the code for kernels into an ELF file, separate from the application code running on the host. NVCC either embeds this ELF file into the application binary as a fat binary or saves it in a separate *cubin* file.

We extended Cricket for using it with Rust applications by introducing support for using GPU kernels compiled into cubins, and loaded using the `cuModule` API. This contrasts with the previous method of loading kernels that relied on the hidden initialization routines added by the `nvcc` compiler to CUDA applications and only worked for fat binaries. Now, applications can read compiled GPU Kernels from cubin files at the client side and send them via RPC to Cricket server. Here, Cricket extracts metadata from the cubin. This includes kernel names, kernel parameter information and global variables. As cubins can be compressed for increased loading performance, we developed a decompression routine that enables access to the metadata even for compressed kernels [2].

3.4 ONC Remote Procedure Calls in Rust

We use ONC or RFC 5531 RPCs [22] as an interface between the unikernel and Cricket. Compared to other solutions, ONC RPCs offer a high performance because it does not use a human-readable data format. The RPC interface is specified using the language-agnostic *Remote Procedure Call Language* (RPCL). Based on the RPCL specification, the networking code and serialization code necessary for calling RPCs is automatically generated.

Previously, Cricket used only the *libtirpc* implementation of ONC RPCs. However, for using Cricket with unikernels, we developed a new implementation that generates the Client side RPC routines based on the RPCL specification [4]. This implementation is called RPC-Lib. To ease building support for RPC-Lib, it has only the Rust standard library as a runtime dependency. RPC-Lib is written in rust, which is beneficial for use with RustyHermit, as having both, the kernel and the application, be Rust code does not weaken the ability of the Rust compiler to check and optimize the executable.

For the code generation of the RPC procedures, we use Rust procedural macros that generate the necessary code during compilation of the application. Keeping to the RPCL specification and making no assumption on operating system features makes our approach universal, in that we can generate an RPC client not only for Cricket but for any RPC application. To additionally support the Rust concept of lifetimes for GPU memory, we wrap the `cudaMalloc` and `cudaFree` APIs, making GPU allocations work like local heap allocations. This way, we can guarantee the absence of use-after-free and double-free errors for the CUDA allocation API. We published the RPC-Lib code at https://github.com/RWTH-ACS/RPC-Lib under a permissive open-source license.

3.5 Integration

Using the previously discussed Cricket, RPC-Lib, RustyHermit, and Unikraft, we become able to build unikernel applications that can use GPU resources to accelerate computations.

Figure 4 shows the interactions among the components of our setup. The rust application in green uses the Rust Standard Library to interact with the unikernel and RPC-Lib to interact with the GPU resources. RustyHermit and Unikraft run within the QEMU hypervisor with KVM acceleration, via which they access the hardware resources of the system. RPC-Lib uses the Rust Standard Library to communicate with the Cricket Server via a TCP socket. The communication between RPC-Lib and Cricket Server is based on the Cricket RPCL specification file. This file lists each CUDA API function, their parameters, and the data types necessary for communication. On the client side, we use RPC procedural macros to generate an RPC interface that is callable from application code. On the server side, we use the `rpcgen` tool to generate the RPC server code in C. Once the Cricket Server receives an RPC request, it executes the respective CUDA API function and returns the result to the application running in the unikernel. While adding or changing RPCs requires a new implementation in the Cricket Server, our

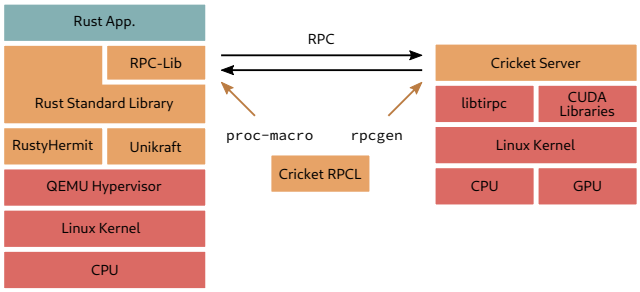


Figure 4: Overview of component interactions with user-provided code in green, third-party components in red, and components we added contributions to in yellow.

approach to generating the client code means that no new implementation is required in RPC-Lib. Functions listed in the RPCL file are immediately available for applications.

RPC-Lib uses no special Unikernel API, but only features that are also available from the Rust standard library when running on Linux. This means, without any code modification, we can run the same Rust application without the RustyHermit unikernel directly on Linux. In Figure 4 RPC-Lib would then use Linux APIs instead of RustyHermit APIs for communication.

4 EVALUATION

Our evaluation setup is composed of two nodes connected via 100 Gbit/s Ethernet. One node runs the application while the application's GPU code is executed on the other node. The GPU Node has two AMD EPYC 7313 processors, 1.5 TiB of memory, and one NVIDIA A100 GPU, two T4 GPUs, and one P40 GPU. While we verified our solution with all of these GPU generations, we limited this evaluation to using the A100 GPU, as it is the most recent generation available to us. The other node has two AMD EPYC 7301 processors and 128 GiB of memory. Both nodes run Rocky Linux 8.7 and Linux 4.18.0. The interconnect consists of Mellanox ConnectX-5 network cards configured for IP over InfiniBand (IPoIB) operation.

We evaluate our approach using proxy applications that represent realistic workloads as they occur in compute clusters and using micro-benchmarks that assess the bandwidth and latency of interaction with remote GPUs. In Cricket we deactivate native InfiniBand support and multithreaded network transfers for `cudaMemcpy` because the unikernels do not support Infiniband. The scripts and versions of Cricket, RPC-Lib, RustyHermit, Unikraft, and CUDA Samples used for this evaluation are accessible at [9].

In this evaluation, we compare the performance of running GPU applications in the two unikernels RustyHermit and Unikraft with a classic Linux virtual machine (VM) and native execution without a hypervisor. We use QEMU/KVM as a hypervisor and a TAP device using virtio for network virtualization for the unikernels and the VM. All network devices were configured with an IP-MTU of 9000. For native execution, we include Rust applications that use RPC-Lib and C applications that use libtirpc to connect to Cricket. The Linux VM uses a Fedora 37 image.

4.1 Application Benchmarks

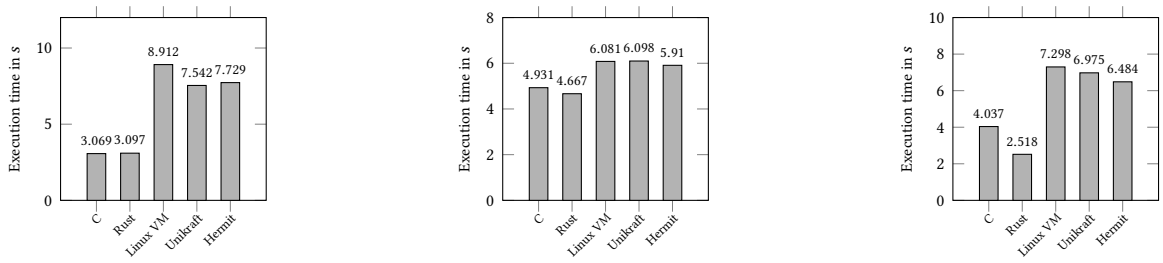
For the proxy applications, we use `matrixMul`, `cuSolverDn_LinearSolver`, and `histogram` from the CUDA Samples project provided by NVIDIA [3]. `matrixMul` performs repeated multiplications of two matrices. `cuSolverDn_LinearSolver` performs a LU decomposition of a system of linear equations and solves the system. The `histogram` application calculates the histogram of a randomly initialized array of data. We ported these applications to Rust, so we could compare the performance of the unikernels to that of the original C applications. The entire execution time of the applications, including initialization and reading or converting input data, was measured using the GNU time command.

We note that all evaluated applications are I/O intensive because they execute many kernels with small execution times, rather than long-running, large workload kernels. With the configuration used in this evaluation, the `matrixMul` application requires 100,041 CUDA API calls and 1.95 MiB of memory transfers, the `cuSolverDn_LinearSolver` application requires 20,047 CUDA API calls and 6.07 GiB of memory transfers, and the `histogram` application requires 80,033 CUDA API calls and 64 MiB of memory transfers.

Figure 5 shows only minor differences between the C and Rust implementations for `matrixMul` and `cuSolverDn_LinearSolver`. For `histogram`, the Rust implementation is approx. 37.6 % faster than the C application. Profiling of the two implementations showed that the C applications use a slower random number generator for initialization, and that this application has particularly short-running kernels. Without considering the initialization, the Rust application is still 27.3 % faster. This means the latency of kernel launching code has a greater impact for these applications. For the other applications, the execution time of the kernels hides this latency. Across all applications, the unikernel and Linux VMs are slower than native execution. The differences in the overhead vary across applications, suggesting a dependency on the application profile. For `matrixMul` and `histogram`, unikernels require more than double the time compared to native execution. However, for `cuSolverDn_LinearSolver` the RustyHermit unikernel only introduces approx. 26.6 % overhead. Unikernels consistently perform similar or better than the Linux VM, supporting the promise of unikernels of having lower virtualization overhead due to a smaller, specialized kernel and a single-address-space OS. Comparing the results shown in Figure 5 to the number of API calls and transferred memory of the applications suggests that the overhead of unikernel and Linux VMs is mostly related to the number of API calls made. While `cuSolverDn_LinearSolver` requires the largest amount of memory to be transferred, the overhead on the execution time is the smallest. The `matrixMul` application requires a higher number of API calls than the `histogram` application, but shows a smaller overhead. Consequently, the amount of data transferred between the GPU and application influences performance.

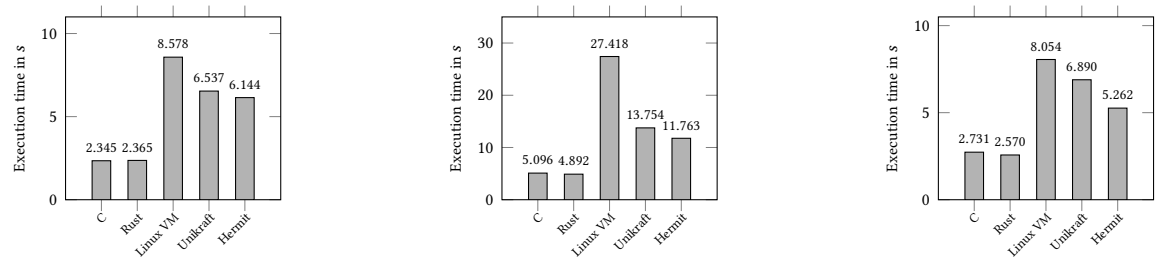
4.2 Micro-Benchmarks

For a more in-depth understanding of the effects found in the previous subsection, we performed micro-benchmarks measuring the overhead of API calls and the bandwidth of memory transfers to and from the GPU. We measured the time taken for 100



(a) matrixMul with 100 000 iterations **(b) cuSolverDn_LinearSolver LU with 900x900 matrix, 1000 Iterations** **(c) histogram**

Figure 5: Comparison of execution time based on 10 averaged runs on a Tesla A100 via 100 Gbit/s Ethernet.

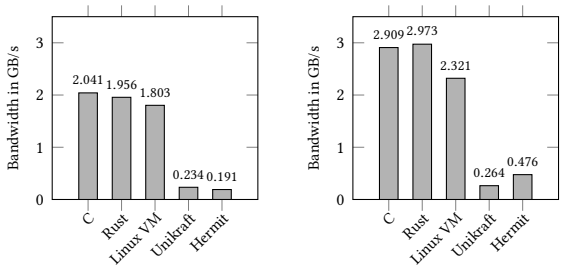


(a) cudaGetDeviceCount **(b) cudaMalloc and cudaFree** **(c) Kernel launch**

Figure 6: Execution time of 100 000 calls of CUDA APIs.

Name	app.	OS	Hypervisor	Network
C	C	Rocky Linux	-	native
Rust	Rust	Rocky Linux	-	native
Linux VM	Rust	Fedora VM	QEMU	virtio
Unikraft	Rust	Unikraft	QEMU	virtio
Hermit	Rust	Hermit	QEMU	virtio

Table 1: Overview of configurations for the evaluation



(a) Memory transfer from device to host **(b) Memory transfer from host to device**

Figure 7: Memory transfer bandwidth based on 10 averaged runs of `bandwidthTest` example application shipped with CUDA on a Tesla A100 over a 100 Gbit/s Ethernet link with 512 MiB of memory.

000 calls of `cudaGetDeviceCount`, of memory allocations by alternating `cudaMalloc` and `cudaFree` calls, and of kernel launches. `cudaGetDeviceCount` is a simple CUDA function, having no parameters and only returning the number of available GPUs. Memory allocations represent a functionality typically used during a CUDA application and require some internal bookkeeping rather than just retrieving a simple result. Kernel launches are of interest because they account for the largest number of API calls in the previously discussed CUDA applications.

Figure 6 shows a significant overhead of unikernel and Linux VMs over the non-virtualized executions in Rust and C. The Linux VM requires the most time for all evaluated APIs, while RustyHermit shows the smallest overhead, but still requires more than double the time of the native executions. As Unikraft, RustyHermit, and the Linux VM are all affected, we can conclude that the cause of the overhead stems, at least partly, from the virtualization of the network interface by the hypervisor and host operating system. However, the unikernel and Linux VMs show different overheads, suggesting that guest-side network management is also responsible for overhead. Because with unikernels, the kernel and the application share a single address space, no classic context switches within the guest are necessary. In contrast, the Linux VM shows a significant delay introduced by the Linux kernel. The natively executed C and Rust implementations show only minor performance differences. For kernel launches, the Rust implementations perform approx. 6.3 % better than the C implementation because the Rust implementations omit some logic required in the C implementation to ensure compatibility with launching CUDA kernels using the <<<...>>> operator.

To evaluate the bandwidth achievable via the Cricket virtualization layer, we ported the `bandwidthTest` application from the CUDA Samples to Rust and used it to measure the bandwidth of device-to-host and host-to-device memory transfers. Figure 7 shows that the unikernels are unable to achieve bandwidths comparable to the native C and Rust implementations. While the Linux VM can retain at least 80 % of performance, RustyHermit can only reach approx. 9.8 % in one direction. The low performance of the unikernels is to a large degree due to them not supporting all hardware offloading features of the networking hardware, e.g., TCP segmentation offloading. Additionally, Unikraft does not support checksum offloading, yet, although this feature has been proposed[4]. When we deactivate TCP segmentation offloading, transmit checksum offloading, and scatter-gather in the Linux VM, the bandwidth is reduced to approx. 923.9 MiB/s in the host-to-device direction. However, the device-to-host direction is influenced much less, leading to the conclusion that there are significant inefficiencies when reading from the network.

Cricket implements multiple methods for transferring device memory between applications and devices: RPC arguments, parallel sockets, InfiniBand and shared memory. Transferring memory via the RPC arguments means using the same TCP connection also used by the RPCs. Cricket receives the data in a buffer and transfers it to the GPU using the CUDA API. As the RPC library is single-threaded, it cannot achieve the full bandwidth of the 100 Gbit/s Ethernet connection of our evaluation system but is instead bound by the CPU's single-core performance. In this paper, we evaluated only the use of RPC arguments for memory transfers because there is currently no support for the other methods in the unikernels or our Rust implementation.

Transferring memory using multiple threads and sockets makes higher bandwidths possible. However, because we have to use a buffer to store the transferred memory before starting to move it to the GPU, we cannot achieve full bandwidth with this method either. The highest bandwidth is achievable using GPUdirect RDMA [19] via InfiniBand for remote GPU access and shared memory for local GPU access. These methods eliminate the need for a buffer, allowing the direct transfer of data from the application to the GPU. While supported by Cricket, this is not possible with RustyHermit or Unikraft because of the lack of support for sharing memory between host and unikernels or drivers for the network card.

Improving the network performance of unikernels directly translates to improved performance in our approach to building GPU applications with unikernels. Our results indicate that the network bandwidth of unikernels is lacking compared to Linux VMs and native executions. Better utilizing the already possible offloading mechanism of the virtio driver used to achieve virtualized network access will certainly improve performance. Another promising approach is vDPA [7], which removes the virtualization overhead from the data path by allowing direct access to hardware queues for VMs and unikernels.

5 CONCLUSION

In this paper, we demonstrated how unikernels can benefit from the computational performance of GPUs by using GPU virtualization. With our work, unikernels can access GPUs flexibly and remotely, without the need of implementing GPU drivers. Our evaluation reveals that the performance of our approach is limited by the network performance of RustyHermit and Unikraft. Applications running in unikernels cannot achieve comparable performance to applications running without virtualization. Compared to traditional Linux VM virtualization, RustyHermit and Unikraft perform better for the applications we evaluated.

Considering the overhead we found in this paper, our approach is best suited to GPU applications that have long-running, high-workload GPU kernels, which consequently require less communication. To reduce the overhead found in this paper, the development of unikernels should focus on better supporting networking features that improve bandwidth. For both, RustyHermit and Unikraft, there are ongoing efforts to support TCP segmentation offloading, which we expect to increase performance significantly.

Because the use case of unikernels involves using many unikernels to run isolated applications, mapping entire GPUs to individual unikernels is not feasible. In contrast, our approach allows the flexibility of sharing GPU devices across many unikernels, managing the shared access through configurable schedulers and runtime reorganization of tasks through checkpoint/restart. This way, large-scale deployments of unikernels in heterogeneous clusters become possible.

[4] https://github.com/unikraft/lib-lwip/pull/12

ACKNOWLEDGMENTS

This research was supported by the German Federal Ministry of Education and Research (BMBF) under Grant 16ME0688 (Project ScalNEXT), by the European Union's Horizon 2020 research and innovation program under grant agreement No 957246 (IoT-NGIN), and by the European Union's Horizon Europe research and innovation program under grant agreement No 101070118 (NEMO).

REFERENCES

[1] 2021. *ONC RPC*. https://crates.io/crates/onc-rpc [Online; accessed 15-August-2023].
[2] 2023. CUDA Fatbin Decompression. https://github.com/n-eiling/cuda-fatbin-decompression [Online; accessed 17-August-2023].
[3] 2023. Cuda Samples. https://github.com/NVIDIA/cuda-samples [Online; accessed 15-August-2023].
[4] 2023. RPC-lib - Rust implementation of RFC 5531 RPCs. https://github.com/RWTH-ACS/RPC-Lib [Online; accessed 15-August-2023].
[5] 2023. The Rust CUDA Project. http://web.archive.org/web/20230816104444/https://github.com/Rust-GPU/Rust-CUDA [Online; accessed 16-August-2023].
[6] 2023. smoltcp. http://web.archive.org/web/20230818131937/https://github.com/smoltcp-rs/smoltcp [Online; accessed 18-August-2023].
[7] 2023. vDPA - virtio Data Path Acceleration. http://web.archive.org/web/20230815101911/https://vdpa-dev.gitlab.io/ [Online; accessed 15-August-2023].
[8] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. 2022. Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. *Concurrency and Computation: Practice and Experience* 34, 14 (2022), e6474. https://doi.org/10.1002/cpe.6474
[9] Niklas Eiling, Marting Kröning, Jonathan Klimt, Phlip Fensch, Stefan Lankes, and Antonello Monti. 2023. *Evaluation Resources for "GPU Acceleration in Unikernels Using Cricket GPU Virtualization"*. https://doi.org/10.5281/zenodo.8263489 Derived from https://git.rwth-aachen.de/acs/public/publications/cricket-hermit.
[10] Nelson Mimura Gonzalez and Tonia Elengikal. 2021. Transparent I/O-Aware GPU Virtualization for Efficient Resource Consolidation. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 131–140. https://doi.org/10.1109/IPDPS49936.2021.00022
[11] Sergio Iserte, Javier Prades, Carlos Reaño, and Federico Silla. 2021. Improving the management efficiency of GPU workloads in data centers through GPU virtualization. *Concurrency and Computation: Practice and Experience* 33, 2 (2021), e5275. https://doi.org/10.1002/cpe.5275

[12] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[13] Stefan Lankes, Jonathan Klimt, Jens Breitbart, and Simon Pickartz. 2020. RustyHermit: A Scalable, Rust-Based Virtual Execution Environment. In *High Performance Computing*, Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 331–342. https://doi.org/10.1007/978-3-030-59851-8_22

[14] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: A Unikernel for Extreme Scale Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (Kyoto, Japan) *(ROSS '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2931088.2931093

[15] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 461–472. https://doi.org/10.1145/2451116.2451167

[16] Nubificus LTD. 2023. *vAccel*. http://web.archive.org/web/20230816105317/https://docs.vaccel.org/ [Online; accessed 16-August-2023].

[17] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. Technical Report. http://web.archive.org/web/20230816105457/https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf [Online; accessed 16-August-2023].

[18] NVIDIA Corporation. 2021. *Virtual GPU Software, User Guide*. http://web.archive.org/web/20230816105820/https://docs.nvidia.com/grid/latest/pdf/grid-vgpu-user-guide.pdf [Online; accessed 16-August-2023].

[19] NVIDIA Corporation. 2023. *GPUDirect RDMA*. http://web.archive.org/web/20230816105138/https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf [Online; accessed 16-August-2023].

[20] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[21] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. 2023. Unikernel Linux (UKL). (2023), 590–605. https://doi.org/10.1145/3552326.3587458

[22] Robert Thurlow. 2009. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531. https://doi.org/10.17487/RFC5531