An Open-Source Virtualization Layer for CUDA Applications

Niklas Eiling, Stefan Lankes, and Antonello Monti

E.ON Energy Research Center, Institute for Automation of Complex Power Systems, RWTH Aachen University, Aachen, Germany
{niklas.eiling,slankes,amonti}@eonrc.rwth-aachen.de
http://www.acs.eonrc.rwth-aachen.de

Abstract. GPUs have achieved widespread adoption for High-Performance Computing and Cloud applications. However, the closed-source nature of CUDA has hindered the development of otherwise commonly used virtualization techniques. In this paper, we evaluate the feasibility of building a GPU virtualization layer that isolates the GPU and CPU parts of CUDA applications to achieve better control of the interactions between applications and the CUDA libraries. We present our open-source tool that transparently intercepts CUDA library calls and executes them in a separate process using remote procedure calls. This allows the execution of CUDA applications on machines without a GPU and provides a basis for the development of tools that require fine-grained control of the GPU resources, such as checkpoint/restore and job schedulers.

Keywords: GPU · Virtualization · CUDA · Remote execution

1 Introduction

Hardware accelerators continue to attract significant interest in the cloud- and High-Performance-Computing (HPC) fields. Today, a growing number of clusters employ GPUs as hardware accelerators, because of the high peak performance and efficiency they offer at a reasonable cost [1,6]. The reason for these advantages is the optimization of GPUs for application profiles that are common in many computing applications: Highly parallel programs that use similarly executing threads to process large amounts of data. This leads to clusters with GPUs having better energy efficiency and a higher performance/price ratio than those without GPUs [6]. As power efficiency is increasingly becoming a limiting factor for performance [4,18], GPUs will certainly play an even more important role in future high performance systems. Despite this, they are rarely integrated into otherwise commonly used virtualization techniques that have the ability to improve availability and utilization of resources by allowing dynamic allocation and/or restriction of computing resources [5,8].
GPU virtualization is challenging because of 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 \textit{device memory} and the CPU memory \textit{host memory}. While there are several frameworks that developers might use to create GPU applications \cite{12}, CUDA is most commonly used for the implementation of computing applications. 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. 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. Nevertheless, this paper focuses on NVIDIA GPUs and CUDA, as these products are the most commonly used for computing tasks\footnote{The Top500 list (https://www.top500.org/) from November 2019 that ranks the fastest HPC clusters contains no cluster that uses GPUs from different vendors.}.
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, thus enabling several usage scenarios for GPUs that are not possible with standard NVIDIA tools (see Fig. 1). GPU virtualization enables remote execution, i.e., the sharing of GPUs by multiple CUDA applications, which may be located on different systems. This makes a cluster setup possible, where GPUs are concentrated on a few nodes, instead of being homogeneously distributed across all nodes. With such a setup, a higher GPU utilization is achievable because the amount of GPU and CPU resources assigned to jobs is flexible [16]. Furthermore, the fine-grained control of the computing resources assigned to individual CUDA applications allows the implementation of custom schedulers that can balance, limit and track the use of GPU resources. Different CUDA applications may then be isolated from the influence of other processes on the system, in respect of performance and resources. GPU virtualization is also a requirement for the implementation of checkpoint/restart schemes for CUDA applications, where the execution state is saved and may be restored later. Checkpoint/restart requires virtualization for the ability to record the interactions with the GPU driver. During the execution of GPU code, the NVIDIA driver exhibits a changing internal state, which cannot be trivially reconstructed without knowledge of past driver interactions. Checkpoint/restart may be used to increase the flexibility and fault tolerance of clusters and to facilitate task migration thereby enabling dynamic load balancing [8].

This paper presents a virtualization layer that enables the realization of these scenarios. 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. The rest of the paper is structured as follows: In Sect. 2, we provide an overview of the current state of research into GPU virtualization. Section 3 presents the implementation of our virtualization layer. We evaluate our solution in Sect. 4 and finally draw a conclusion in Sect. 5.

2 Related Work

There has been some previous work on the virtualization for CUDA applications. However, for most virtualization solutions no source code is available and others support only outdated CUDA versions.

rCUDA allows CUDA application to use GPUs installed in a remote system [3], as shown in Fig. 1a. 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 re-implemented
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 generation. rCUDA is not open-source and the authors make only a small amount of implementation details available, making detailed evaluation of the approach and code reuse impossible. The authors target users who want to remotely execute existing applications 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 shown in Fig. 1.

Another GPU virtualization approach is DS-CUDA [13], which targets a scenario where a cloud provides the GPU resources for local CUDA applications. 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 newer than from the Pascal generation 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 [15]. 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 forwards them to the CUDA driver. However, vCUDA only supports CUDA version 1.1, which predates support for any data center grade GPU from the Tesla line of products. Additionally, the source code of vCUDA is not available anymore.

The CUDA Multi-Process Service (MPS) enables multiple GPU jobs to be executed concurrently, thus increasing GPU utilization compared to the case where only a single application may occupy a GPU at any given time [11]. 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. MPS uses named pipes and domain sockets for this communication. 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 customizations and reuse impossible.

None of the previously discussed solutions represents an open-source virtualization solution for GPUs that supports the latest GPU generation. In this paper, we present a novel tool that offers all the benefits of previous work, while supporting the latest GPUs and being released under an open-source license.²

² The code is available at https://github.com/RWTH-ACS/cricket.
3 GPU Virtualization for CUDA Applications

The goal of our virtualization solution for GPUs is to offer a basis for the development of resource management strategies that require control over how applications use computing resources. 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 the latest GPU generation and CUDA toolkit\(^3\).

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, e.g., 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 Remote Procedure Calls (RPC) to send requests to an RPC server that is responsible for the management of the available GPUs.

The remaining chapter 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. 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 (Fig. 2). 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.

![Diagram](image)

**Fig. 2.** With default setting the CUDA runtime API is linked statically. It loads the driver API dynamically during startup using `dlopen` and `dlsym`.

\(^3\) As of writing the latest GPU generation and CUDA version are Turing and CUDA 10.2.
Going one layer up the software stack, the next possible point of separation is at the CUDA driver API. However, as a result of undocumented interactions, the driver API cannot be cleanly separated from the runtime API. Consequently, a virtualization layer at the driver API level does not allow the use of the original runtime API on top of it. 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 [7,10].

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. 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 Remote Procedure Calls (RPCs)

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. Unlike other approaches, such as rCUDA, we execute the original API function even for the runtime API and do not re-implement the runtime API using the driver API. We use the Transport Independent Remote Procedure Calls (TI-RPC) implementation of the Remote Procedure Call Protocol Specification Version 2 [17] as a basis for this communication.

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
Replacing the runtime API with RPC code allows separating the CUDA application from the internal state of both runtime and driver APIs. 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, e.g., pointers to device memory and internal data structures. These pointers are only intended for passing to the CUDA APIs and user programs should not dereference them 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 a small amount of data for parameter and return values. In 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 3 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 remote procedure call 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 Evaluation

For the evaluation we use one system equipped with two Intel Xeon Gold 6128 CPUs and Tesla P40 and Tesla T4 GPUs and one node with two AMD Epyc 7301 CPUs and no GPUs. Both nodes are connected via a Gigabit Ethernet and an Infiniband 100 Gb/s link using Mellanox MCX556A-ECAT ConnectX-5 Adapter Cards. Unless otherwise noted, we are using the Infiniband Link with IP over Infiniband (IPoIB) for the communication between the nodes. While we confirmed the compatibility of our virtualization layer with the previously mentioned GPUs, the performance impact of virtualization is independent of the specific GPU. Therefore, this section presents results only for the Tesla T4, as it is from the latest generation. All measurements have been performed using version 10.2 of the CUDA toolkit. We compare the execution time of several applications when using our virtualization solution to the case where no virtualization is employed. Additionally, we performed several micro benchmarks to assess potential sources of overhead.

4.1 Benchmarks

The virtualization layer introduces overhead as a result of the communication between CUDA applications and the RPC server. To analyze the impact of this overhead on CUDA applications, we evaluate our virtualization layer with two of the example applications distributed with the CUDA Toolkit and two third-party application. The \textit{matrixMul} application performs a series of densely filled matrix-matrix multiplications without repeatedly copying the data between host and device. The \textit{nbody} application is a physics simulation that computes the gravitational interaction between a configurable amount of bodies. \textit{hotpsot} from the Rodinia Benchmark Suite [2] is a thermal simulation application that solves differential equations. \textit{DPsim} is a real-time capable power system simulator for

![Fig. 4. Execution time overhead based on 25 averaged runs on a Tesla T4.](image)
dynamic phasor and electromagnetic transient simulations [9]. The variety of considered applications shows our virtualization layer supports a set of CUDA features is sufficient for productive use.

Figure 4 compares the reference execution time of the applications without virtualization with our virtualization layer using a domain socket connection, a local loopback TCP connection and a remote TCP connection between two systems. It shows a low overhead due to the virtualization layer for matrixMul, nbody and dpsim. For hotspot, the remote execution introduces an overhead of approx 25%. This is because of all applications hotspot transfers the largest amount of memory between host and device memories, resulting in the transfer bandwidth reduction of the virtualization layer having a higher impact on the execution time. The matrixMul applications mostly launches kernels and performs only a few other calls to the CUDA API. The low overhead for this application thus shows that kernel dispatches are not significantly slowed down by the virtualization layer.

For all applications, the execution times when communicating locally via a domain socket are larger than when communication via a local TCP socket, suggesting that the RPC implementation is not as efficient for domain sockets as it is for TCP sockets. Remote executions across the IPoIB connection of the considered applications have comparable performance to local executions. This shows, that with modern high speed interconnects the data transfer between systems has a smaller performance impact than the virtualization layer itself for data-intensive applications such as hotspot.

4.2 Micro Benchmarks

To quantify the impact the virtualization layer has on the execution time of calls to the CUDA API, we measure the latency of two typical CUDA API functions in different virtualization scenarios. `cudaMalloc` allocates a region of device memory and represents a commonly used API function. `cudaGetDeviceCount` returns the number of GPUs available to the CUDA API. As such, this function

---

**Fig. 5.** CUDA API overhead based on 100 averaged runs on a Tesla T4.
requires the transfer of only a single integer, resulting in almost all latency being due to communication delays.

For both API functions we measure an overhead between 11.4 to 36µs when using the virtualization layer. This increase is due to the execution of additional code and copying of parameters and results that is necessary for the redirection of API calls. 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, hostpot and DPsim perform 72, 10 and 72 API calls, respectively. Only matrixMul performs a higher amount of 10033, while still showing only a small overhead due to the virtualization layer (see Fig. 4a). 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 memories also require an analysis.

Figure 6 shows the achieved memory transfer bandwidth for the reference case and with our virtualization layer using local and remote communication. The virtualization layer decreases the bandwidth for transfers from and to the GPU, as a result of the additional data transfer between CUDA application and RPC server. The previous observation that the RPC implementation is not able to fully utilize the bandwidth of domain sockets is here again visible. When the pinned memory API is used on local data transfers, our virtualization layer avoids the additional transfer by instead using shared memory. Therefore, the performance for this case is comparable to the reference case of using the pinned memory API without virtualization (see Fig. 6a and Fig. 6b).

![Figure 6](image)

**Fig. 6.** Memory transfer bandwidth based on 100 averaged runs of `bandwidthTest` application from the CUDA toolkit.
When client and server reside on different computing nodes the interconnect bandwidth limits the overall transfer bandwidth up to approx. 1.2 GB/s (see Fig. 6c and Fig. 6d). Using Gigabit Ethernet we measure a bandwidth of approximately 112 MB/s for device to host transfers and 114 MB/s for host to device transfers, which is close to the maximum bandwidth of Gigabit Ethernet. However, using the IPoIB interconnect the virtualization layer does not fully utilize the available interconnect bandwidth. Even when using IBverbs for the transfer, the achieved bandwidth is 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 to local execution.

5 Conclusion

The virtualization solution for GPU devices presented in this paper provides a basis for future research on advanced task management techniques, which may increase flexibility, utilization, and fault tolerance. Our virtualization layer is fully transparent to CUDA applications, i.e., requires no source code modifications or recompilation. This is despite the closed-source nature of the CUDA software that makes the development of virtualization solutions for GPUs difficult. By intercepting CUDA API calls and redirecting them to a separate process, we achieve isolation and full control of GPU resources used by applications. Even though the virtualization layer increases the individual CUDA API functions latency and reduces memory transfer bandwidths, this incurs only a small overhead to the overall execution time of CUDA applications. The overhead when communicating across different systems is mainly due to the bandwidth limitations of the considered Gigabit Ethernet interconnect. Thus, faster interconnects, such as 10 Gigabit Ethernet or Infiniband should be able to increase the memory transfer bandwidth. Because we publish our code under an open source license, others may customize and reuse it to implement software that addresses the growing need for increased flexibility in HPC clusters.

Acknowledgment. This research and development was supported by the German Federal Ministry of Education and Research under Grant 01IH16010C (Project ENVELOPE).

References

1. Baker, Z.K., Gokhale, M.B., Tripp, J.L.: Matched filter computation on FPGA, cell and GPU. In: 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), pp. 207–218, April 2007. https://doi.org/10.1109/FCCM.2007.52
2. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 44–54. IEEE (2009)
3. Duato, J., Peña, A.J., Silla, F., Mayo, R., Quintana-Ortí, E.S.: rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In: 2010 International Conference on High Performance Computing Simulation, pp. 224–231, June 2010. https://doi.org/10.1109/HPCS.2010.5547126
4. Esmaeilzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. IEEE Micro 32(3), 122–134 (2012). https://doi.org/10.1109/MM.2012.17
5. Gavrilovska, A., et al.: High-performance hypervisor architectures: virtualization in HPC systems. In: Workshop on System-Level Virtualization for HPC (HPCVirt). Citeseer (2007)
6. Kutzner, C., Páll, S., Fechner, M., Esztermann, A., de Groot, B.L., Grubmüller, H.: More bang for your buck: improved use of GPU nodes for GROMACS 2018. J. Comput. Chem. 40(27), 2418–2431 (2019). https://doi.org/10.1002/jcc.26011
7. Laurenzano, M.A., Tikir, M.M., Carrington, L., Snively, A.: PEBIL: efficient static binary instrumentation for Linux. In: 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS), pp. 175–183 (2010)
8. Mílojičić, D.S., Dougis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Comput. Surv. 32(3), 241–299 (2000). https://doi.org/10.1145/367701.367728
9. Mirz, M., Vogel, S., Reinke, G., Monti, A.: DPsim—a dynamic phasor real-time simulator for power systems. SoftwareX 10, 100253 (2019). https://doi.org/10.1016/j.softx.2019.100253
10. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 89–100. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1250734.1250746
11. NVIDIA Corporation: Multi-process service. Technical report. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed 04 May 2020
12. NVIDIA Corporation: NVIDIA(R) CUDA(TM) architecture. Technical report. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf. Accessed 10 May 2020
13. Oikawa, M., Kawai, A., Nomura, K., Yasuoka, K., Yoshikawa, K., Narumi, T.: DS-CUDA: a middleware to use many GPUs in the cloud environment. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1207–1214 (2012)
14. Reaño, C., Silla, F.: A performance comparison of CUDA remote GPU virtualization frameworks. In: Proceedings of the 2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, pp. 488–489. IEEE Computer Society (2015). https://doi.org/10.1109/CLUSTER.2015.76
15. Shi, L., Chen, H., Sun, J., Li, K.: vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans. Comput. 61(6), 804–816 (2012)
16. Silla, F., Prades, J., Iserte, S., Reaño, C.: Remote GPU virtualization: is it useful? In: 2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), pp. 41–48 (2016)
17. Srinivasan, R.: RPC: remote procedure call protocol specification version 2 (1995)
18. Villa, O., et al.: Scaling the power wall: a path to exascale. In: SC 2014: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 830–841, November 2014. https://doi.org/10.1109/SC.2014.73