Interception Inception: Analyzing CUDA call Interception methods for Serverless Environments

Pavlidakis[1,2], Anargyros Argyros[1], Stelios Mavridis[1], Giorgos Vasiliadis[1,3], & Angelos Bilas[1,2]
[1]Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH), Greece
[2]Computer Science Department, University of Crete, Greece
[3]Department of Management Science & Technology, Hellenic Mediterranean University, Greece

GPUs are necessary for accelerating applications. However they often remain underutilized [1, 5–8, 13–15] because solo executions cannot always fully utilize their resources. To share a GPU to multiple applications, previous work try to virtualize accelerators [3, 9, 14] or to provide Infrastructure as a Service (IaaS) [2, 4, 11]. Other virtualization techniques, such as full- [10] and para [12] virtualization, have limited applicability due to the requirement for custom drivers [14]. API remoting [2, 9, 14] is the only stable and efficient technique for accelerator abstraction. Existing API remoting approaches [2–4, 9, 11, 14] intercept partially the CUDA driver and runtime API, as well as the high-level calls of CUDA accelerated libraries (e.g., cuBLAS and cuDNN).

This three-level interception approach requires handling more than 2000 calls though, most of which have complex semantics that change often. For example, just to support cuBLAS, cuRAND, and cuFFT, we need to handle more than 1600 high-level calls, a process that is usually performed offline by hand. As a consequence, previous work [2, 3, 14] offer limited support for complex frameworks, such as PyTorch, Tensorflow, and GROMACS. To make matters worse, high-level calls of CUDA accelerated libraries (e.g., `cublasIsamax`) perform implicit CUDA calls that are hidden from the developer. These calls execute both host and device code, which does not scale in client-server setups where the CPUs are not designed to perform computations.

In this paper, we propose CUInterposer, a fine-grain interception mechanism at the CUDA driver and runtime library. The design of CUInterposer provides a clear boundary between CPU and GPU code, that allows to fully support popular frameworks, such as PyTorch. More specifically, CUInterposer intercepts the whole CUDA driver and runtime library that consists of 400 relatively simple calls. Due to the simplicity of these calls, the interception process is automated, leading to zero manual effort as opposed to previous works. Finally, we can distinguish host and device calls because our approach intercepts the implicit calls performed from closed-source high-level function calls of CUDA accelerated libraries. As a result, with CUInterposer, only the device code is forwarded to the server, whereas the host code is executed in the client. We effectively address the following challenges:

Intercept only CUDA runtime and driver libs. CUInterposer intercepts all the CUDA driver and runtime calls by dynamically preloading the execution of the applications. CUDA libraries make use of an undocumented data structure named export table that contains function pointers to hidden CUDA calls. CUInterposer uses a minimal implementation of these hidden CUDA calls, which is, however, adequate to run complex frameworks. Additionally, we have found that only the static version of CUDA closed-source accelerated libraries link with CUDA driver and runtime. In contrast, the shared version includes the whole CUDA, making intercepting prohibitive. To avoid linking applications with the static version of CUDA accelerated libraries, we create a shared version that internally uses the static versions of libraries.

Support closed-source libraries. CUInterposer intercepts the implicit calls performed from high-level CUDA accelerated. For the CUDA runtime library, it replaces the original library using `ld_preload`. For the CUDA driver library, we intercept the `dlopen` system call, which CUDA uses to decouple applications from GPUs. CUDA kernels are registered to the GPU driver using `cudaRegisterFunction` and are launched using the `cudaLaunchkernel`. `cudaLaunchkernel` uses a pointer provided by the `cudaRegisterFunction` to issue a kernel for execution. However, this pointer is not valid to the server address space. As a result, CUInterposer client in our `cudaRegisterFunction` creates a map with the kernel pointer and name. During `cudaLaunchkernel`, the client sends the kernel string to the server. CUInterposer during an offline extracts all the PTXs for the available frameworks and libraries and places them in the server. During the server startup, we create a key-value pair with the kernel name as the key and the pointer as the value, so upon the receipt of a string, we can find the appropriate pointer kernel.

The main contributions of this work are:

- We design, implement, and evaluate two interception approaches at different granularity levels, the one at the CUDA driver and runtime level, and the other includes the high-level library calls.
- We implement tools that automatically generate stubs for intercepting CUDA applications. Our tool requires only the CUDA header files; hence, it supports all CUDA versions fully automatically.
- Our preliminary evaluation shows that our approach can fully support complex frameworks, such as PyTorch, which we use for performance evaluation.

Acknowledgment: We thank the EUPILOT project, which has received funding from the European High-Performance Computing Joint Undertaking (JU) under GA 101034126. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Italy, Switzerland, Germany, France, Greece, Sweden, Croatia, and Turkey.

References

[1] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *ASPLOS '18*.

[2] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. 2011. Enabling CUDA acceleration within virtual machines using rCUDA. In *HiPC '11*.

[3] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. 2022. Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. In *Concurrency and Computation: Practice and Experience*.

[4] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. 2022. DGSF: Disaggregated GPUs for Serverless Functions. In *IPDPS '22*.

[5] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters. In *SoCC '22*.

[6] NVIDIA. 2022. Multi-Instance GPU. Retrieved April 2023 from https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf

[7] NVIDIA. 2022. Multi-Process Service. Retrieved May 2023 from https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[8] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU Concurrency with Elastic Kernels. In *ASPLOS '13*.

[9] Manos Pavlidakis, Stelios Mavridis, Antony Chazapis, Giorgos Vasiliadis, and Angelos Bilas. 2022. Arax: A Runtime Framework for Decoupling Applications from Heterogeneous Accelerators. In *SoCC '22*.

[10] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?. In *USENIX ATC'14*.

[11] Lukas Tobler. 2022. Gpuless–serverless gpu functions. In *Master Thesis*.

[12] Dimitrios Vasilas, Stefanos Gerangelos, and Nectarios Koziris. 2016. VGVM: Efficient GPU capabilities in virtual machines. In *HPCS'16*.

[13] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Saugata Ghose, Abhishek Bhowmick, Rachata Ausavarangnirun, Chita Das, Mahmut Kandemir, Todd C Mowry, and Onur Mutlu. 2016. A Framework for Accelerating Bottlenecks in GPU Execution with Assist Warps. In *ArXiv*.

[14] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. In *ASPLOS '20*.

[15] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-Net: Effective GPU Sharing in NFV Systems. In *NSDI'18*.