OpenKL Proxy Architecture

Version: 1.0

Document Revision: A

Stillwater/Simunova OpenKL Working Group

Editors: Peter Gottschling, Theodore Omtzigt

Table of Contents

[Acknowledgements 4](#_Toc36487674)

[Introduction 5](#_Toc36487675)

[Glossary 6](#_Toc36487676)

[The OpenKL Proxy Architecture 7](#_Toc36487677)

Copyright © 2020 Cloud-Native Supercomputing. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to Stillwater Supercomputing, Inc. and Simunova, UG. This specification, nor any of its component, may not be reproduced, republished, distributed, transmitted, displayed, broadcast, or otherwise exploited in any manner without the express prior written permission of Cloud-Native Supercomputing. You may use this specification to create a software implementation of the functionality defined therein. You may not alter or omit any trademark, copyright, or other notice from the specification. The receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

KPU, Knowledge Processing Unit, Domain Flow Architecture are trademarks of Stillwater Supercomputing, Inc. MTL4 and MTL5 are trademarks of Simunova, UG. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

# Acknowledgements

The OpenKL specification is the result of contributes of many people in the cloud-native and high-performance computing software and hardware communities. Following is a partial list of contributors, including the company or university they represented at the time of their contribution.

Peter Gottschling, Simunova, UG

Theodore Omtzigt, Stillwater Supercomputing, Inc.

# Introduction

Demand for high-performance computing solutions is rapidly expanding with key new applications in cloud, edge, and embedded and autonomous computing. The key attributes of these new architectures are that they are highly distributed, and frequently represent custom application-specific compute engines tightly integrated with sensors or actuators to autonomously act on continuous stimulus. The compute engines may not be continuously connected to remote compute infrastructures, thus requiring a dynamic check-in/report upload/download capability. Furthermore, it is possible that a device might only have the resources to orchestrate a large-scale computation and only consume the results, all the while communicating asynchronously with remote compute or storage. Fundamentally, these autonomous intelligent applications require seamless access to high-performance, specialized application hardware or infrastructure.

**OpenKL** is an open industry standard for programming a hierarchical distributed machine consisting of application-specific hardware devices, gateways, edge and cloud servers, and their respective proxies. The key idea behind OpenKL is how to solve the problem of scalability for deep computational operations. As an algorithm goes through different scales, the optimal organization of resources changes. For example, a matrix operation of 3x3 matrices is best expressed as one single basic block of instructions without any branching or looping. But that same operator on a matrix with one billion elements may require a distributed memory machine to solve quickly and reliably.

Other examples are provided by custom hardware accelerators, such as, Deep Learning tensor processors, media encoding/decoding, or cryptography processors. OpenKL is designed to make such diverse computational resources available in the same computational model.

In applications such as hierarchical sensor fusion, collective intelligence, or guidance, navigation and control, not all information required to solve the problem is localized, and thus algorithms by their very nature are distributed and connected.

Due to the requirement to directly and dynamically connect application data structures and operators to local and remote compute and storage infrastructures, OpenKL introduces a proxy layer to decouple the remote connection details from the local application. This document describes this proxy architecture in more detail.

# Glossary

**Application:** the combination of the program running on the host, hardware accelerated devices, connected remote compute infrastructures, and digital twins.

**Barrier:** a barrier ensures that all previous commands have finished execution before any following commands can begin execution.

**Command:** an operation representing an atomic amount of work. Commands can represent state manipulation operators such as creating memory objects, or represent large scale fine-grain parallel computation such as solving a system of equations.

**Device:** a device is a stateful execution unit consisting of local memory and a processor fabric capable of executing programs to transform the content of the local memory.

**Handle:** an opaque type that refers to an object, such as a dense vector or a sparse matrix. Any operations on an object occur by reference to its handle.

**Kernel:** a kernel is a fine-grain parallel program that can execute atomically on a device.

**Local Memory:** a memory region associated with a device.

**Object:** an object is an abstract representation of a complex data structure that can be manipulated by the OpenKL API and kernel programs. Examples are a sparse column-oriented matrix of 20-bit posit numbers.

**Proxy:** a server application that acts as an intermediary for requests from the OpenKL run-time seeking resources from other servers, possibly remote, that can provide those resources.

# The OpenKL Proxy Architecture

The key to seamless access to local and remote resources is the OpenKL Proxy. The OpenKL Proxy acts as the window into a vast universe of remote edge and cloud resources that can be incorporated into solving a deep computational problem, such as optimization, approximation, or simulation.

What functionality goes into the proxy?

Do we support data structure migration, duplication, resilience, etc. ?

Capabilities of the Proxy

* collect a compute target database that the application can utilize to determine where to compute or gather data
  + is this database dynamic, or static?
    - Static: at time of boot, the proxy goes out and finds resources and is fixed after that
    - Dynamic: periodically, the proxy goes out and finds resources and updates the database accordingly
    - The proxy should be HA, resilient, and dynamic to be of most value
* Aid in securing connections between local and remote resources

Other services the proxy could be used for

* A functional simulation of different devices to provide emulation services to applications. For example, a emulated RISC-V environment with posits, or custom hardware accelerators under development.
* Virtualization of data stores; the proxy can act like a cache, or a prefetcher.
* Check-pointing state that might have to migrate to a remote site for resilience, voting, restarts, or process migration. This is likely to be very useful for embedded systems as they would benefit most from a virtual scalable resource model.

The OpenKL proxy is different from an HTTP proxy in the sense that HTTP proxies are stateless, whereas a typical compute workload is stateful.

The OpenKL proxy could create virtual private network overlays to connect a set of proxies to form a larger compute infrastructure. In this use case, the OpenKL proxy is much like a tinc server:

**What is tinc?**

tinc is a Virtual Private Network (VPN) daemon that uses tunnelling and encryption to create a secure private network between hosts on the Internet. tinc is Free Software and licensed under the [GNU General Public License](https://www.gnu.org/licenses/old-licenses/gpl-2.0.html) version 2 or later. Because the VPN appears to the IP level network code as a normal network device, there is no need to adapt any existing software. This allows VPN sites to share information with each other over the Internet without exposing any information to others. In addition, tinc has the following features:

**Encryption, authentication and compression**

All traffic is optionally compressed using zlib or LZO, and LibreSSL or OpenSSL is used to encrypt the traffic and protect it from alteration with message authentication codes and sequence numbers.

**Automatic full mesh routing**

Regardless of how you set up the tinc daemons to connect to each other, VPN traffic is always (if possible) sent directly to the destination, without going through intermediate hops.

**NAT traversal**

As long as one node in the VPN allows incoming connections on a public IP address (even if it is a dynamic IP address), tinc will be able to do NAT traversal, allowing direct communication between peers.

**Easily expand your VPN**

When you want to add nodes to your VPN, all you have to do is add an extra configuration file, there is no need to start new daemons or create and configure new devices or network interfaces.

**Ability to bridge ethernet segments**

You can link multiple ethernet segments together to work like a single segment, allowing you to run applications and games that normally only work on a LAN over the Internet.

**Runs on many operating systems and supports IPv6**

Currently Linux, FreeBSD, OpenBSD, NetBSD, OS X, Solaris, Windows 2000, XP, Vista and Windows 7 and 8 platforms are supported. See our section about [supported platforms](https://tinc-vpn.org/platforms) for more information about the state of the ports. tinc has also full support for IPv6, providing both the possibility of tunneling IPv6 traffic over its tunnels and of creating tunnels over existing IPv6 networks.

The quickest way to experiment with this functionality is to include tinc daemons in the design.

Initially, I was thinking that we would target containers and their public or private interfaces, but being able to create a mesh of connected devices feels like a great feature.