VLCs: Managing Parallelism with Virtualized Libraries

Yineng Yan*, William Ruys*, Hochan Lee*, Ian Henriksen*, Arthur Peters*
Sean Stephens*, Bozhi You*, Henrique Fingler*, Martin Burtscher†, Milos Gligoric*
Keshav Pingali*, Mattan Erez*, George Biros*, Christopher J. Rossbach*‡

*The University of Texas at Austin, †Texas State University, ‡Microsoft

Abstract

As the complexity and scale of modern parallel machines continue to grow, programmers increasingly rely on composition of software libraries to encapsulate and exploit parallelism. However, many libraries are not designed with composition in mind and assume they have exclusive access to all resources. Using such libraries concurrently can result in contention and degraded performance. Prior solutions involve modifying the libraries or the OS, which is often infeasible.

We propose *Virtual Library Contexts (VLCs)*, which are process subunits that encapsulate sets of libraries and associated resource allocations. VLCs control the resource utilization of these libraries without modifying library code. This enables the user to partition resources between libraries to prevent contention, or load multiple copies of the same library to allow parallel execution of otherwise thread-unsafe code within the same process.

In this paper, we describe and evaluate C++ and Python prototypes of VLCs. Experiments show VLCs enable a speedup of up to 2.85× on benchmarks including applications using OpenMP, OpenBLAS, and LibTorch.

CCS Concepts

• **Software and its engineering → Virtual machines**; **Process management**.

Keywords

Virtualization, Software Libraries, Resource Management

1 Introduction

Configuring and composing external libraries within the same application can be challenging. Many modern libraries rely on parallelism internally and make strong assumptions about the system resources their runtimes use. Libraries such as OpenMP [13], OpenBLAS [64], Qthreads [61], RaftLib [6], TBB [48], PTask [49], Galois [41], and ARPACK [1], may assume they own all system resources or only allow configuration at the process level (e.g., through environment variables). When these libraries are not aware of each other's resource usage, they can over-allocate resources causing degraded performance. Even when configuration options are available, they are often limited in granularity and flexibility. Options may not be sufficient to avoid contention between libraries or to optimally exploit parallelism.

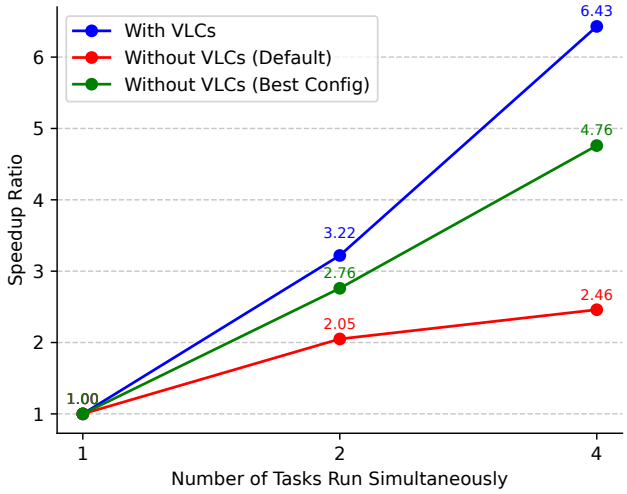

Figure 1: The speedup ratio of parallel hyperparameter tuning on a Transformer model in C++ LibTorch relative to sequential hyperparameter tuning.

Consider an application that uses OpenBLAS and OpenMP to orchestrate concurrent linear algebra operations, such as calling OpenBLAS kernels from OpenMP threads. By default, both libraries spawn one thread per core, leading to oversubscription and severe resource contention. OpenBLAS can be configured at the process level, but if threads process matrices of different sizes this equal allocation may be suboptimal. Contention can also arise from concurrent use of a single library, as has been reported for OpenMP [28]. In Section 2, we demonstrate that this slowdown, compared to an optimal allocation of resources, can be significant. Library APIs do not always allow the flexibility to effectively partition resources among threads.

Current solutions for managing cross-library interference require modifications to library code or the OS kernel. This

limits their practical usage and adoption, especially in scientific software which is often built on a stack of legacy libraries not under the developers' direct control [4]. Lithe [43] provides a standardized interface and thread abstraction primitives to coordinate resources with a shared runtime. Parallel libraries must be modified to use its runtime API. This limits its adoption on alternative, legacy, and closed-source runtime systems: only TBB [48] and OpenMP have been ported to Lithe. Bolt [28] provides an alternative OpenMP runtime that avoids oversubscription when calling multiple OpenMP-parallelized libraries with user-level threads. Effort is required to maintain the alternative runtime, and it does not provide a solution for non-OpenMP based workloads. Light-Weight Contexts (LwCs) [35] address contention between libraries with separate resource and protection domains within a single process. However, this isolation requires kernel level modifications, limiting usage on managed computing clusters.

We introduce *Virtual Library Contexts* (VLCs), an easy-to-use tool for controlling library composition that does not require modifying library code. VLCs are a user space, lightweight, library virtualization layer that enables users to load distinct sets of libraries into a single process and associate each set with specific system resources (e.g., a set of cores or GPUs). Different libraries can be allocated disjoint resources to mitigate contention. When a library is loaded into a VLC, its resource-management API calls are interposed and virtualized by intercepting system calls and file system accesses used to query system resources. VLCs require no modification of the library code and no recompilation of the library binary. For example, the available CPU cores visible to libraries within one VLC can be limited such that they will not be oversubscribed when used with libraries in other VLCs.

VLCs provide performance isolation but not data isolation. Libraries between VLCs still efficiently share data in the same address space. By loading multiple instances of the same library into separate VLCs, the instances can run in parallel using distinct internal static state and separate partitions of resources. This enables applications to make safe simultaneous calls into libraries that are otherwise not thread-safe. While VLCs enable application control over resources, finding the right resource configuration is non-trivial. VLCs solve this with an auto-tuner that automatically searches the configuration-space. This helps the user identify (often counterintuitive) optimal configurations.

We implement VLCs for C++ and Python and present several use cases based on a set of common and important libraries, including OpenMP, OpenBLAS, LibTorch [44], Kokkos [15], and ARPACK [1]. We show that VLCs enable efficient composition of libraries within a single-process application; current alternatives require the use of multiple processes and explicit inter-process communication. These examples overcome composability challenges that either degrade performance or prevent the application from running correctly, including those relating to libraries that either: (1) assume they own all process resources; (2) can utilize only a subset of available resources (e.g., a single GPU or a single core); or (3) assume a static and inflexible resource allocation. Our experiments show that the VLC mechanism has low overhead, allowing VLCs to increase performance for applications with multiple concurrent libraries that contend. VLCs achieve a speedup of up to 2.85× on benchmarks, a 1.41× speedup on a multi-GPU Heat3D application using Kokkos, and a 1.96× speedup on the ARPACK eigenvalue solver. We make the following contributions:

- We introduce the novel concept of library-level virtualization that offers fine-grained resource management without requiring library or OS modifications. Programmers can adopt VLCs with as few as 10-20 lines of application code changes.
- We demonstrate how library-level virtualization helps avoid resource contention, improves nested parallelism, and enables parallelization of thread-unsafe calls.
- We implement C++ and Python prototypes to demonstrate the applicability of VLCs on compiled and interpreted languages.

2 Motivation

Complex application codes often combine multi-threaded libraries that do not always support fine-grained management of resources.

`FFTW` [19], `SuperLU` [34], `OpenCV` [27], and `Sundials` [24] allow setting the number of threads globally, but do not allow allocation of specific threads to a call. The latter may result in suboptimal memory access and unnecessary communication overheads. `Mumps` [2] supports an arbitrary number of threads but only through the environment variable `OMP_NUM_THREADS` which affects not just `Mumps`, but libraries that rely on OpenMP.

Many numerical methods require nested multi-library calls. Examples include: multigrid methods involving relaxation, restriction, and prolongation steps that at different levels may require a different number of threads [17]; Hierarchical matrix and N-body codes in which different phases may require a different number of resources for optimal performance [53]; Multiphysics and adaptive mesh refinement methods that may require radiation solvers, stencil solves, semi-Lagrangian steps, particle-in-cell steps, or boundary and internal interface physics that again require fine-grained control for optimal performance [66]. Furthermore, tasks like optimization, uncertainty quantification, and machine

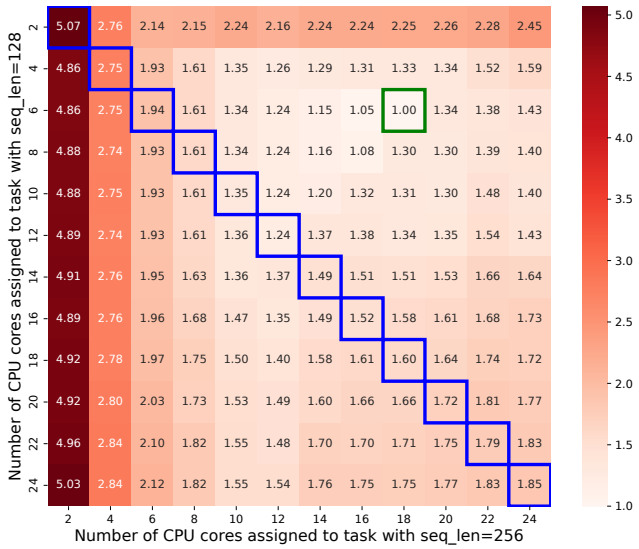

Figure 2: Heatmap of relative execution times for CPU core partitions between two concurrent hyperparameter tuning tasks. Lighter regions indicate shorter execution times. The optimal partition is marked with a green box. Blue boxes represent partitions achievable with LibTorch APIs. Without VLCs, the optimal partition cannot be achieved.

learning induce further diversity of libraries and workloads. Controlling resources in hierarchical and nested scientific computing modules can be quite challenging.

OpenMP spawns as many threads as there are logical cores by default. When an application composes OpenMP with other parallel libraries, each library may allocate a thread pool equal to the number of cores. This can lead to oversubscription, where threads compete for a limited number of cores. Developers have reported issues with degraded performance when combining OpenMP with parallel libraries like OpenBLAS [23]. Additionally, a persistent problem arises when two libraries using OpenMP are composed: they cannot set different OpenMP thread-pool sizes because they share the same OpenMP instance and configuration. This issue remains unresolved for PyTorch, Numba [3], Ray [30], and scikit-learn [59]. Similarly, while it is possible to manually set the number of available threads globally, libraries like PyTorch (and C++ LibTorch) do not provide an API to configure thread allocation at a per-operation granularity.

Example Hyperparameter tuning workflows for machine learning models can be accelerated by parallelizing runs to leverage the large memory capacity and massive parallelism [33, 65]. When running on the same node, it can be beneficial to run within a single process to efficiently share large datasets and intermediate computations [52]. We implement a C++ LibTorch parallel hyperparameter tuning workflow on a transformer-based language model with 8 heads, 6 layers, and a 512 embedding size. The training dataset is `wikitext2` [38].

We measure total training time for tuning a fixed number of hyperparameters with different numbers of concurrent tasks. Figure 1 shows the speedup relative to sequential tuning on a two-socket ARM architecture supercomputer node with a 144-core NVIDIA Grace CPU Superchip and 237 GB of DRAM. In the default LibTorch configuration, each of the training tasks spawns 144 threads (matching the total number of cores). Running 4 tasks concurrently creates 576 threads, severely oversubscribing the 144 cores and leading to poor scaling due to contention.

Without fine-grained control, such as that provided by VLCs, the best alternative using the current PyTorch API is to globally limit the threads per task (e.g., 36 threads each for 4 tasks). This prevents CPU oversubscription but results in suboptimal resource utilization, as models with different hyperparameters show varying scalability and execution times. Assigning them an equal number of threads fails to maximize parallelism. VLCs enable each task to be allocated a disjoint set of cores with a tailored number of threads. This strategy eliminates contention while improving core utilization. As shown in Figure 1, VLCs achieve up to a 6.43× speedup over the sequential baseline and 2.61× over the default concurrent configuration. VLCs outperform the best achievable configuration with the current PyTorch API by 1.35×.

Figure 2 illustrates how execution time varies when VLCs assign different cores to two tasks. This experiment specifically involves tuning a model with different sequence lengths (128 and 256) simultaneously on a 24-core x86 computer. The best performance is achieved by allocating 6 cores to the task with 128 sequence length and 18 cores to the 256 sequence length task. Since LibTorch does not support assigning different numbers of threads to each task, its only tuning option without VLCs is represented by the diagonal of the heatmap (highlighted with blue boxes). Without VLCs, it fails to achieve the best performance as the optimal configuration is not on the heatmap diagonal.

3 Related Work

Composition of Libraries. Software design techniques that separate work from workers can, in principle, solve the problem that we address with VLCs. For example, Lithe [43] enables efficient library composition by requiring libraries to expose APIs to an external thread manager. This enables performance improvements in applications with mixed or nested parallelism. Lithe requires the libraries to be ported to use those APIs. The adoption of such techniques has not

been widespread. To the best of our knowledge, only TBB and OpenMP have adopted Lithe techniques.

VLCs, in contrast, require no changes to existing libraries. Like VLCs, libcapsule [40] enables the use of incompatible libraries in the same application. However, it provides no mechanisms for resource management. For applications composed of multiple libraries that are all parallelized by OpenMP, Bolt [28] is a new runtime design using user-level threads to reach a balance between minimizing thread oversubscription while maximizing parallelism. However, this approach is OpenMP specific and requires a new implementation of the OpenMP library. A more general, non-library-specific solution, such as VLCs, is desirable.

Virtualization. Containers [39, 51] and VMs provide a way to isolate runtime environments, but they provide no support for simple data sharing between components. In fact, they are explicitly designed to prevent this kind of direct communication, which renders them inappropriate for separating components of a single application. VLCs support the virtualization of resources at library granularity within a single address space where data are shared natively. Figure 3 illustrates the layers of resource management in different virtualization techniques. VLCs provide a new technique for resource virtualization, which allows applications to manage the resources available to libraries.

Library OSes [5, 16, 45, 55], Unikernels [18, 36, 37, 47, 62, 67], and lightweight virtualization techniques such as Xax [14] and picoprocesses [26] have the goal of giving applications more granular and flexible control over low-level resources. These techniques require special kernel support or emulators, which makes the application non-portable.

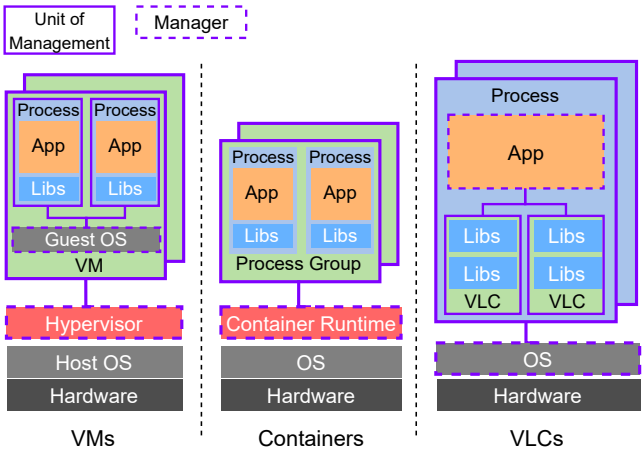

Figure 3: Overview of VLCs and other virtualization techniques. Dashed boxes represent resource managers and solid boxes are the unit of management.

OS-level mechanisms including `cgroups` and `cpuset` provide a way to limit the resource usage at the process level. In contrast, VLCs provide a fine-grained resource isolation and management at an intra-process level.

Subunits of Processes. Process-in-Process (PiP) [25] loads entire programs into a shared address space, typically using an optimized MPI runtime for communication. In contrast, VLCs focus on managing libraries within a single process. Communication between VLCs occurs directly via function calls and standard shared memory mechanisms without MPI.

Lightweight isolation techniques such as LwCs [35], CubicleOS [50], and Mondrix [63] provide sub-address space *data* isolation. These systems require source-level changes and are motivated by increasing isolation for security. Additionally, LwCs require special kernel support. VLCs require no source code modification or recompilation of existing libraries and are motivated by providing performance isolation.

4 Design

VLCs are performance-isolated execution environments that encapsulate sets of libraries in a single process. Each VLC can have different assigned resources and environment variables. With VLCs, users can force a library to only use specific resources even if the library does not provide resource management APIs.

Libraries do not have to be unique to a VLC; the same library may be instantiated in several VLCs with different configurations in each. Each VLC has its own linker namespace and static state, allowing multiple, even potentially incompatible, library versions to coexist in one process.

Figure 4 illustrates the VLC system. The application creates VLCs via API calls to isolate libraries and assign resources. At runtime, a VLC Monitor transparently intercepts library system calls and virtualizes resource-related queries.

4.1 Library Isolation

VLCs use *linker namespaces* to provide isolation between libraries. Linker namespaces give users the ability to load a dynamic shared object (DSO) into an explicit namespace, such that libraries in one linker namespace cannot access symbols in other namespaces. This isolation prevents symbol conflicts when multiple copies of the same library are loaded into a single process. Since all libraries are still in the same address space, data can be shared between them through virtual memory. Multiple instances of the same DSO can be loaded into different namespaces without conflict by giving each instance distinct static internal state. This enables parallel calls into a library that may not be thread-safe since a call that changes the state of one copy of the library will not affect the state of the other. This also enables loading potentially incompatible versions of a library into the same

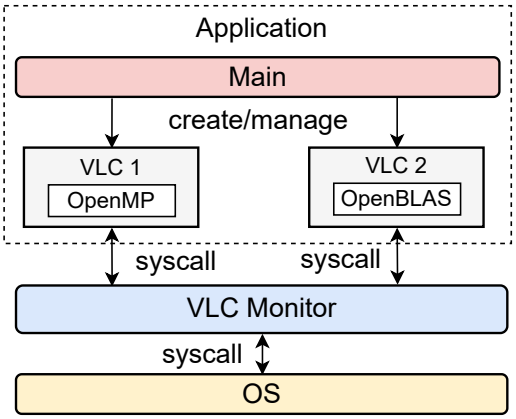

Figure 4: Programming model of VLCs. OpenMP and OpenBLAS are loaded into separate VLCs with different resource allocations; Resource query system calls are interposed by the VLC Monitor.

process, similar to libcapsule [40]. This feature is discussed in detail in Section 7.1.

4.2 Resource Virtualization

The VLC runtime virtualizes system resources such as the number of CPU cores available to a VLC via system call interposition and virtualized resource configuration files. When an application initializes the VLC runtime, a VLC Monitor process is forked to interpose system calls from the application process. We use `ptrace` to intercept system calls because it operates entirely in user space. This allows our tool to be used on externally-managed systems, like supercomputers, where users lack root privileges, unlike kernel-module-based tools such as SystemTap [29]. To reduce interception overhead, `Seccomp BPF` is used as a filter to avoid intercepting system calls that are not related to resource management.

Resource-query system calls are interposed by the VLC Monitor process. When libraries within a VLC attempt to query available system resources, the responses are modified so that the caller perceives only the resources allocated to the VLC. When a resource query system call is invoked multiple times during a library's life cycle, the forged resource result may be cached for each VLC to reduce overhead. Some libraries query resources by accessing resource configuration files directly. Galois [41], for example, counts the number of CPU cores by reading `/proc/cpu` files. Resource visibility for these libraries is restricted by redirecting I/O requests to virtual resource configuration files that list only the resources the user has assigned to the VLC. To ensure threads within a VLC are scheduled exclusively on their allocated cores, the VLC Monitor sets CPU affinity for all threads. Additionally, any `sched_setaffinity` system calls made by libraries in a

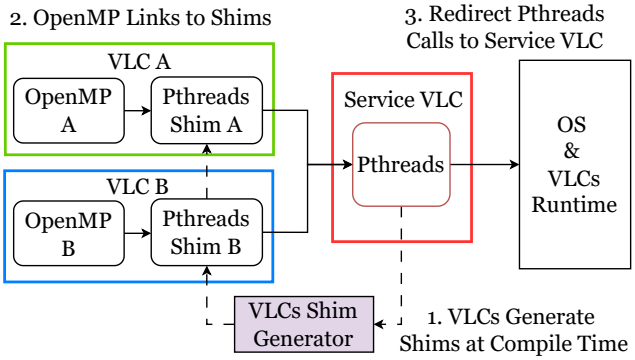

Figure 5: Service VLC Control Flow. OpenMP is loaded into a VLC and links to the generated shim of pthreads. The calls to pthreads are redirected to Service VLC.

VLC are interposed to prevent them from scheduling threads on cores outside their assigned set.

4.3 VLC Configuration and Management

VLCs are assigned resources by the application programmer or the VLCs auto-tuner. The assignments of different VLCs may overlap if desired. Resource allocations can be configured before a library is loaded and can be adjusted at any time. However, libraries and their associated thread pools within a VLC may not adapt well to dynamic changes. Environment variables for a specific VLC can also be reconfigured during program execution, though the frequency with which a library checks these variables is beyond our control. An alternative approach is to create multiple VLCs of the same library with different configurations and switch between them as needed. The low overhead of entering and exiting a VLC (see Section 6.1) makes this approach practical.

4.4 Service VLC

In Linux, a linker namespace is created by calling `dlmopen`. However, a few libraries including TBB, CUDA, and older versions of pthreads (prior to glibc 2.34) [22, 60] are not compatible with `dlmopen`. A common cause for this incompatibility is the library calling `dlopen` internally. A nested call to `dlopen` in a linker namespace created by `dlmopen` causes a segfault due to a bug in `dlmopen` [21]. To address these incompatible cases, we provide a Service VLC to load shared instances of the problematic libraries.

The Service VLC is a special context into which a `dlmopen`-incompatible library is loaded globally via `dlopen`. Unlike `dlmopen`, which creates a new linker namespace, `dlopen` loads libraries into the base namespace and avoids creating another copy of the library if it is already loaded. We provide a script to automatically generate shims for libraries loaded

into the Service VLC. These shims replicate library symbols for each context, making them available to other VLCs. When libraries in a VLC access methods from a `dlmopen`-incompatible library, the shim performs API forwarding to the underlying globally loaded library in the Service VLC. This ensures libraries in VLCs can depend on libraries that are incompatible with `dlmopen`.

Figure 5 illustrates how the Service VLC allows OpenMP to utilize the `dlmopen`-incompatible pthreads library within a VLC. Before a VLC is created, pthreads is loaded into the Service VLC. The addresses of pthreads methods are dumped into a page shared with the shim to direct where calls should be forwarded. When OpenMP is loaded into the VLC, a pthreads shim is loaded as its dependency. The shim is generated by the VLCs Shim Generator at compile time. When OpenMP is running, the shim is responsible for redirecting all pthreads calls to the Service VLC. Exchanging function addresses via a shared page occurs only once during Service VLC initialization. These addresses are cached in a jump table to minimize the cost of forwarding API calls.

The Service VLC is transparent to libraries that depend on it. It does not require any code modifications or recompilation of these libraries or applications. The overhead of forwarding an API call through the shim is negligible (see Section 6.1), as it is implemented in just 23 lines of assembly code and incurs a constant cost regardless of the workload.

4.5 Programming Interface

We provide an API in C++ and Python to create VLCs and manage their resources, listed in Table 1. Figure 6 shows an example of using the Python interface for VLCs. The VLCs `a` and `b` are created and configured in Lines 1–3. Each isolated execution environment may configure which cores are visible to libraries within that environment, as in Line 3. Control flow can enter and exit a VLC, Line 4, using Python's context manager `with` syntax.

Figure 7 shows the same example using the C++ interface. When the application starts, it initializes the VLC Monitor on Lines 2–3. For illustration purposes, we create two parallel threads using standard C++ on Lines 4 and 10. On Lines 5 and 11 two VLCs are created with the VLC ID and thread ID as arguments. The visible cores need to be configured for each VLC before the libraries are loaded. To load a library into a VLC, a `VLC::Loader` object is created with the library path and a pointer to an associated `VLC::Context`. The last boolean argument indicates whether transparent mode is enabled (see Section 4.6). By running in transparent mode, there is no need to load function pointers from namespaces manually. Once the library is loaded into a VLC, its methods become available for use. This minimizes the code changes required when using VLCs.

```
1 a, b = VLC(), VLC()
2 a.set_allowed_cpus([0])
3 b.set_allowed_cpus([1,2,3,4,5,6,7])
4 with a:
5     import numpy
6     async_computation()
7 with b:
8     import numpy
9     async_computation()
```

Figure 6: An example of VLCs usage in Python.

```
1  int main(int argc, char ** argv) {
2      VLC::Runtime monitor;
3      monitor.initialize();
4      std::async([&]() { /* on thread 1 */
5          VLC::Context a(1, gettid());
6          a.set_allowed_cpus("0-11");
7          VLC::Loader loader(&a, "lib.so", true);
8          computation();
9      });
10     std::async([&]() { /* on thread 2 */
11         VLC::Context b(2, gettid());
12         b.set_allowed_cpus("12-23");
13         VLC::Loader loader(&b, "lib.so", true);
14         computation();
15     });
16 }
```

Figure 7: An example of VLCs usage in C++. `std::async` is used to simplify thread allocation.

4.6 Programmability and Transparency

VLCs are designed to allow existing applications to use them with minimal code changes. To achieve this goal, the use of VLCs should be transparent so most of the application code can remain intact. The only new code that the developer must write is to configure and create a VLC.

One limitation in C++ is that symbols from libraries in the newly created linker namespace are not automatically visible to other namespaces. To call functions from a library loaded by `dlmopen`, one must obtain function pointers by calling `dlsym` explicitly. This must be done for every function and, without tooling, could become a burden on the developer when the number of functions is large. VLCs avoid the need to use `dlsym` explicitly on every function by providing a "transparent mode" in the C++ implementation. This mode uses automatically generated library shims that redirect calls through a runtime jump table (see Section 5.1).

`VLC::Monitor (C++)`	Create the VLC Monitor
`initialize (C++)`	Initialize the VLC Monitor
`VLC::Loader (C++)`	Open a shared library and its dependencies in this VLC
`VLC() (Python)`	Create a VLC
`__enter__ (Python)`	Mark VLC as the default for imports in this thread
`__exit__ (Python)`	Restore the state of imports to what it was before `__enter__` was called
`set_allowed_cpus (Both)`	Make only specific set of CPUs visible to this VLC
`setenv (Both)`	Set an environment variable in this VLC
`unsetenv (Both)`	Unset an environment variable in this VLC

Table 1: VLC API. `setenv`, `unsetenv`, and `set_allowed_cpus` have the same interface in C++ and Python. The `__enter__` and `__exit__` routines are called implicitly in Python's `with` syntax.

We achieve similar transparency in Python by overriding `builtin.__import__` and `getattr` (see Section 5.2).

5 Implementation

We prototyped VLCs in C/C++ and Python. Due to differences in how shared objects are managed and how they load dependencies, there are differences in the details of each implementation.

5.1 VLC++: VLCs for C/C++

VLC++ is a prototype of VLCs for C/C++ applications. VLCs provide namespaces and resource management contexts, while the VLC Monitor provides a supervisor for lifecycle management APIs and system call interposition. When libraries allocate resources, they often must first query the system to learn what system resources are available.

VLC++ creates an interposition layer between libraries and the OS to virtualize the visible resources. Whenever a library makes a system call, it is inspected by the VLC Monitor using `ptrace` and potentially modified to reflect the resources assigned to the current VLC.

Interposing every system call is both expensive and unnecessary. Most libraries will not query system resources frequently, and most of the system calls made by the application are unrelated to VLCs. VLC++ uses `Seccomp BPF` to reduce overhead and filter system calls before interposition by the monitor. Only targeted system calls like `sched_getaffinity` are intercepted by `ptrace` in the VLCs Monitor process. `Seccomp BPF` can filter out most of the system calls without triggering a more expensive trap to `ptrace`. To demonstrate that this overhead is minimal, experiments in Table 4 show the overhead introduced by VLCs themselves is less than 0.54%.

To support libraries that query system resources through `/proc/` files, VLC++ provides a virtualized resource filesystem. When requesting files such as `/proc/cpuinfo`, the *openat* system call will be interposed. This checks the filename in the system call arguments and, if the filename matches a resource file, it will be modified and redirected to a forged resource file. The forged resource file is dynamically generated during VLC Monitor initialization and only shows the subset of resources assigned to the VLC.

Transparency. VLCs maintain transparency and avoid explicit use of function pointers at the source level. In transparent mode, a library shim is linked to the application in place of the target library that is loaded into a VLC. When a VLC is created, the target library is loaded by `dlmopen` and its function addresses are resolved by the VLC++ runtime using `dlsym`. The VLC++ runtime then saves the function address in a jump table maintained by the shim. For each function call, the application calls the shim it is linked to. Then, the shim dispatches the call following the jump table so the call can be executed by a copy of the target library in the VLC. To distinguish which VLC the call should be executed in, the VLC++ runtime and the shim maintain a mapping between thread id and VLC id. All these steps are transparent to the application.

5.2 PyVLC: VLCs on Python

Typical C++ applications load dynamic libraries and their shared object dependencies using the dynamic linker when the application is loaded. However, modules in Python applications are loaded as the application is interpreted. Supporting these differences in how shared libraries are loaded, as well as managing the interface to forward methods from Python modules into specific VLC contexts requires the development of a separate VLC prototype for Python.

For simplicity, PyVLC prototype uses the simpler approach of interposing glibc calls for resource management, though it is straightforward to extend this to the more powerful system call interception approach of VLC++.

The Python import mechanism can be customized by overriding the `builtins.__import__` function without modifying the interpreter. This enables redirecting any module import to the appropriate VLC. To prevent unnecessary duplication of certain modules, they can be marked as *exempt* from the special import handling at the Python level. Such modules are resolved using the default import mechanism into the global namespace. This provides similar functionality to the Service VLC used in the C++ implementation. As the Python interpreter is shared across VLCs, we exempt

modules like sys that are directly connected to its core capabilities. We also exempt standard library modules by default.

When a module is loaded into a new namespace, its symbols are not available for resolution of subsequently loaded libraries [31]. This prevents a module from accessing its dependencies even if all of them are loaded in the same linker namespace. To lift this limitation, we apply a 31-line code patch [42] to the dynamic linker *libdl* that enables support for the RTLD_GLOBAL flag for dlmopen. In contrast, our C++ prototype does not require this patch as a library and its dependencies are loaded all at once by the dynamic linker, instead of dynamically as needed. Symbols of dependencies have been resolved as soon as this loading is finished.

Several internal module caches must be overridden to ensure state isolation in Python VLCs:

Shared Object-Handles Cache: All current Python versions (3.14 and older) maintain a finite cache of shared object handles. Entries of this cache are reused to avoid calls to dlopen when possible. Since VLCs are designed to allow potentially loading *the exact same shared object* multiple times into different VLCs, this caching must be disabled. We disable the cache to ensure that the Python interpreter always receives the handle from the current VLC linker namespace.

Module-Specification Cache: The Python interpreter maintains an internal cache of module specification objects associated with each native Python extension module. Prior to loading any new portion of a library, we swap all known module specification objects associated with that library from the current VLC into the module specification cache.

Module Object Cache: The VLC machinery must manage the module object cache in *sys.modules*. We manage this cache by ensuring that it is correctly populated with references to objects that are appropriate for the current VLC. We refer to the objects we insert into the cache as forwarding modules. When an import occurs that will load a new module, all forwarding modules in *sys.modules* associated with the corresponding library are replaced by those modules in the VLC.

These modifications are implemented by dynamically overriding the interpreter's behavior, enabling PyVLC to operate with the native Python interpreter and ensuring its compatibility. None of these changes require modifying the Python interpreter.

6 Evaluation

VLCs demonstrate low overhead, broad applicability, and effectiveness in improving performance in a range of scenarios. All experiments are performed on a machine with two 12-core Intel E5-2650 v4 sockets, 128 GB of RAM, and four NVIDIA Tesla P100-SXM2 GPUs, running Ubuntu 22.04 with kernel 5.15.0. VLC++ uses the unmodified dynamic linker from glibc 2.35 and is compiled with g++ 11.4.0. PyVLC is implemented on top of Python 3.9 and the patched dynamic linker from glibc 2.30. For libraries involved in the evaluation, we used Open MPI 5.0.3, OpenBLAS 0.3.20, CUDA version 11.7, and Kokkos version 4.6.1

Initialize PyVLC Modules	198 ms
Initialize VLC++ Monitor	0.54 ms
VLC++ Create VLC	40.2 μs
PyVLC Create VLC	3.96 ms
Enter VLC	7.8 μs
Leave VLC	6.9 μs
Enter VLC (w/ Affinity)	40 μs
sched_getaffinity (w/ VLC)	20.44 μs
sched_getaffinity (w/o VLC)	0.64 μs
open("/proc/cpuinfo") (w/ VLC)	266.59 μs
open("/proc/cpuinfo") (w/o VLC)	5.81 μs
mmap (w/ VLC)	0.82 μs
mmap (w/o VLC)	0.78 μs
cudaMemcpy (w/ Service VLC)	2.63 μs
cudaMemcpy (w/o Service VLC)	2.58 μs

Table 2: VLC Management Benchmarks. Times are the mean of 120,000 samples.

We conduct experiments using micro- and macro-benchmarks to address the following research questions (RQs):

- **(RQ1)** What is the overhead of using VLCs?
- **(RQ2)** How much effort is required to use VLCs?
- **(RQ3)** Does performance improve with VLCs?
- **(RQ4)** Do VLCs improve nested parallelism?
- **(RQ5)** Can VLCs parallelize thread-unsafe libraries?

6.1 Overhead

To evaluate **RQ1**, the overhead of VLCs, we run a set of simple microbenchmarks that repeatedly exercise the VLC API and make resource-related and other system calls. We report the average time for each call in Table 2.

PyVLC API calls take much longer than those of VLC++, but both are reasonable in context. The main reason for the longer initialization time of PyVLC is that it must load libraries fully at initialization time (e.g., librt and pthreads), while VLC++ defers those loads. Calls to resource-query APIs must be interposed and do incur an overhead, although it is quite low at only 20μs for sched_getaffinity and 260μs when accessing /proc/cpuinfo. These calls are rare because resources are not regularly reconfigured. Importantly, the runtime overhead of system calls that are not interposed is very low (2%). The overhead of making the library API calls themselves is negligible.

We also evaluate the overhead of loading libraries into VLCs, as shown in Table 3. Library loading via VLC++ is

Library	Prototype	VLCs	Base
numpy	PyVLC	240 ms	181 ms
scipy	PyVLC	333 ms	192 ms
LibTorch	VLC++	524 ms	492 ms
OpenBLAS	VLC++	8 ms	11 ms
OpenMP	VLC++	5 ms	3 ms
Kokkos	VLC++	11 ms	6 ms
Arpack	VLC++	11 ms	16 ms

Table 3: VLCs Library Load Time Comparison. We show time taken to load 4 copies of the same library into VLCs compared to standard loading without VLCs.

Library Name	Application Name	LoC Changes	VLCs Overhead
OpenMP	Kmeans	27	0.11 %
	Hotspot3D	27	0.35 %
	CFD	27	0.42 %
OpenBLAS	Cholesky	13	0.54 %
	GEMM	13	0.38 %
	GESV	13	0.31 %
LibTorch	Transformer	27	0.45 %
	DNN	27	0.14 %

Table 4: VLC application overhead. *Lines of Code Changed* shows the number of code modifications required to use VLCs. *VLCs Overhead* shows the performance slowdown when the application runs serially within a single VLC.

efficient; specifying the full library filepath can lead to faster resolution than when the path is searched without VLCs. The results are different with PyVLC because it overrides Python's default import machinery, disables caches, and forces additional iterations over the Python interpreter's `module.__dict__`. The conclusion is that VLCs have reasonable overheads during initialization and for management calls, whereas overheads from the Service VLC and for system calls that are not interposed are negligible.

In Table 4, we present the end-to-end overhead of using VLCs in applications. All overheads are below 1%, demonstrating that the overall impact of VLCs on application performance is minimal.

6.2 Programmability

To evaluate the programmability of VLCs, **RQ2**, we measure the lines of code and the effort to optimize VLCs resource usage. VLCs do not require library code changes, but do require initialization and interaction with client libraries. To illustrate the ease of porting an existing application to VLCs, Table 4 details the number of lines we needed to modify in each benchmark discussed in Section 6.3. None of the codes require more than 27 lines to be changed.

Although VLCs allow developers to manage resources at a fine granularity, determining the optimal resource partition is not always straightforward. To address this challenge, we developed a tuning framework that performs a grid search over all possible partitions and reports the optimal configuration. Figure 2 presents a heatmap generated by our tool, illustrating which partitions yield the best performance for the LibTorch application discussed in Section 2. The time required by the VLCs auto-tuner to find the optimal configuration depends on the grid size. For instance, determining the configuration for a pair of OpenBLAS benchmarks requires 64 runs with a grid size of 3, taking approximately 10 minutes on a 24-core machine. This time can be significantly reduced if users provide hints to narrow the search space, e.g. by coarsening the grid or reducing the range of values.

6.3 Reducing Contention and Balancing Load

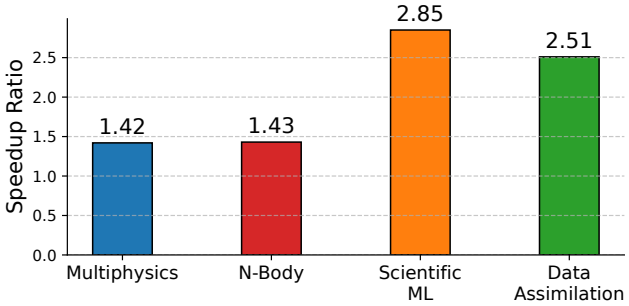

Figure 8: Speedup achieved by using VLCs on representative synthetic applications (higher is better).

Due to the vast design and software codebases, we devised synthetic experiments based on the Rodinia 3.1 test suites [7], OpenBLAS kernels and LibTorch models listed in Table 4. We compose different benchmarks to resemble typical workflows in scientific computing.

(1) **Multiphysics workflow:** Two small-sized Hotspot3D (a heat transfer calculation) with a medium-sized CFD (computational fluid dynamics) and Cholesky (direct symmetric factorization): This workflow resembles a multiphysics/multifidelity simulation that involves a dense linear solve [10].
(2) **N-body workflow:** Combinations of multiple GEMM, GESV, and Cholesky solves of different problem sizes that resemble a hierarchical matrix approximation factorization, a common technique for boundary integral equation solvers and Jacobians of neural networks [8].

(3) **Scientific Machine Learning Workflow:** Combination of a CFD code with Kmeans and DNN, resembling a physics simulation that is corrected & augmented with machine learning corrections [9].
(4) **Data assimilation workflow:** Transformer + many small CFD resembling a time-series transformer followed by an ensemble CFD calculation [46].

We evaluate the performance improvements provided by VLCs in eliminating resource contention (**RQ3**). This involves comparing performance under two configurations: a baseline without VLCs, and a setup where VLCs partition the resources. In both cases, the libraries operate with their default configurations. For each workflow, between 3 to 4 instances of VLCs are created, each with a different CPU allocation. The ability to partition resources between libraries at a fine granularity also enables performance improvements through better load balancing. Here, we assigned more cores to the larger workloads in an experiment.

Figure 8 shows the speedup of composing benchmarks with VLCs relative to composing without VLCs. The result indicates that VLCs can improve performance by up to 2.85×, with an average speedup of 2.05× across all synthetic experiments, demonstrating that VLCs effectively reduce contention and improve load balancing by virtualizing and partitioning resources for libraries.

6.4 Improving Nested Parallelism

To evaluate **RQ4**, we consider workloads dependent on dense matrix multiplication.

Libraries such as OpenBLAS can only be configured with a single fixed level of parallelism by setting a process-level environment variable, making it impossible to tune the level of parallelism for each matrix size in the application.

Programmers are forced to choose a best fit configuration, with the best choice often being to not overlap the execution of large and small GEMMs.

We explore such a scenario in which we perform 21 independent GEMMs of square matrices: twenty 2048 × 2048 GEMMs and a single 8192 × 8192 GEMM. Each GEMM is computed using the OpenBLAS `cblas_dgemm` routine in C++ and `numpy.matmul` in Python (numpy uses OpenBLAS under the hood). Figure 9 shows the execution times for different implementations.

The baseline performs all GEMMs sequentially, each using all 24 cores in our machine. This allows the large matrix to execute fast, while consuming unnecessary resources for the small GEMMs. When parallelized with a conventional approach in C++, the multi-threaded (pthreads) implementation obtains a 1.3× speedup. With Python multi-threading, the benefits of parallelization are negated by the overheads of the global interpreter lock for the short numpy calls of the

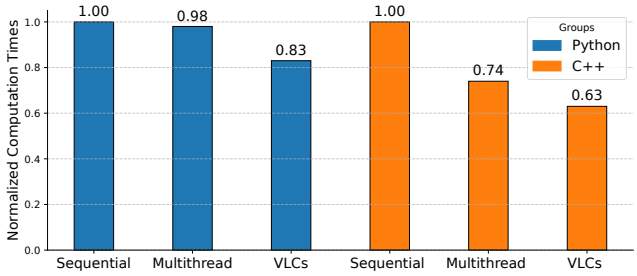

Figure 9: Normalized execution time of matrix multiplication implementations (lower is better). Using VLCs to manage nested parallelism yields an improvement of up to 1.58× over the sequential baseline.

small matrices. While reducing the core allocation of OpenBLAS may allow multiple GEMMs to run in parallel, this hurts performance because reducing the number of cores for the large matrix so substantially degrades overall performance.

With VLCs, we can expose nested parallelism and execute the large and small GEMMs concurrently by dedicating most of the cores to the VLC that performs the large GEMM and the rest of the cores to another VLC that performs the small GEMMs. This approach works well, providing a 1.58× speedup over the C++ sequential baseline (1.17× over the multithreaded version) and 1.2× over the Python sequential baseline (1.17× over the multithreaded version). Allocating 17 cores to the large GEMM and 7 cores to the small GEMM results in the best performance.

6.5 Parallelizing Thread-Unsafe Calls

We evaluate **RQ5** by exploring how VLCs can be used to replicate libraries that are not thread-safe to enable safe concurrent calls to those libraries.

ARPACK [1] is a Fortran library that is utilized by popular libraries like SciPy [57] and MATLAB for solving sparse eigenvalue and singular value decomposition problems. Internally, ARPACK uses an unsynchronized static state, so simultaneous calls into ARPACK are not thread-safe [56]. Libraries that depend on ARPACK have to guard all calls to ARPACK routines with a lock to avoid race conditions. This restriction can be relaxed by importing ARPACK into different VLCs, where the separate static state in each instance makes simultaneous calls safe.

Figure 10 shows the results of computing the top 10 eigenvalues for two 10000 × 10000 matrices using ARPACK. Two instances of ARPACK are loaded into VLCs with either no resource allocation specified or allocating 12 cores to each. The Python implementations use `scipy.sparse.linalg.eigsh`

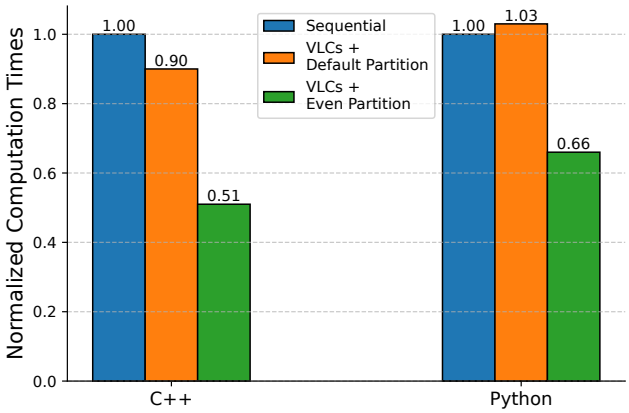

Figure 10: Execution time for computing the eigenvalues of two matrices using ARPACK (lower is better). The default partition loads two instances of ARPACK without specifying resource splits. The even partition allocates 12 cores to each ARPACK call.

as a proxy to the ARPACK routine. The baseline serial execution without VLCs does not utilize resources well. VLCs offer up to a 1.96× speedup in C++ (1.51× in Python) by tuning the cores allocated to each ARPACK call and running two calls concurrently.

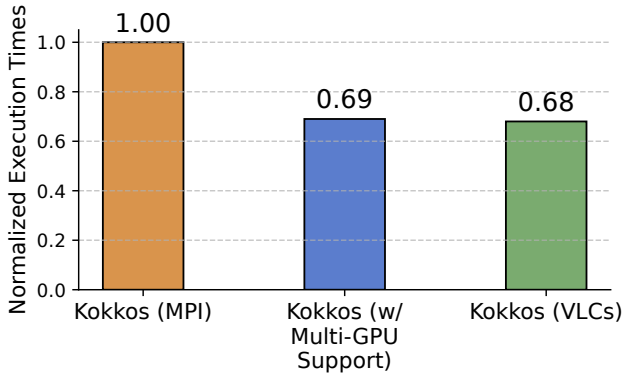

Figure 11: Normalized execution time of Kokkos multi-GPU Heat3D implementations (lower is better). Using VLCs to manage multiple Kokkos instances yields a 1.46× speedup over the MPI implementation and achieves identical performance to newer Kokkos native multi-GPU support.

6.6 Performance Comparison with MPI

Kokkos [15] has become a cornerstone C++ library for high-performance scientific computing. It provides a unified programming model that abstracts the underlying hardware, allowing developers to write parallel programs that can be compiled across heterogeneous architectures. Historically, Kokkos only supported a single GPU per process [54]. Developers who wished to use Kokkos with multiple GPUs on a single machine needed to decompose their application into multiple processes and use IPC frameworks for data sharing, such as MPI. This multi-process MPI-based design has become the de facto standard for a vast ecosystem of Kokkos applications [32].

While single-process multi-GPU support was recently added in Kokkos 4.3 (for CUDA) and Kokkos 4.6 (for HIP), it also brings a migration challenge. Porting an existing MPI-based Kokkos application requires code changes for data sharing and stream synchronization. Migrating a large body of legacy MPI-based codes to the new native multi-GPU model is a non-trivial task. VLCs offer an alternative approach for enabling single process multi-GPU support for Kokkos by allowing multiple instances of Kokkos to be loaded into separate VLCs, where each manages a separate GPU. This approach enables developers to continue using their existing MPI-based Kokkos applications without significant code changes.

We ported Kokkos's Heat3D MPI-based implementation [11] to thread-based orchestration both using VLCs and with the recent native multi-GPU support. We compare three implementations: (1) Kokkos with CUDA-aware MPI, (2) Kokkos with VLCs, and (3) Kokkos with native multi-GPU support. Comparing these approaches allows us to evaluate the performance and migration cost of porting legacy Kokkos MPI-based applications to use multi-GPU in a single process.

This application solves the 3D heat equation on a rectangular box with dimensions $L_x \times L_y \times L_z$ using a Forward Time-Centered Space finite-difference scheme. We solve the problem in a zero-temperature heat bath, with radiative heat loss on its surfaces, and an incoming heat flux on the bottom ($z = 0$) surface that is removed halfway through the simulation. In our experiment, the domain is split evenly between 2 GPUs, each holding $L_x \times L_y \times \frac{L_z}{2}$ grid points. At every time step, the GPUs exchange the $L_x \times L_y$ boundary values between them. In the MPI implementation, CUDA-aware MPI is used to avoid copying data to the host and back when communicating between processes. We evaluate on a $200 \times 200 \times 200$ cube.

Figure 11 shows the resulting performance of different Heat3D implementations. We can see that avoiding inter-process communication overheads when exchanging data between GPUs leads to a significant reduction in application runtime, even when avoiding host communication. Both the native multi-GPU support and support through VLC isolation provide a 1.46× speedup over the typical MPI implementation. VLCs provide a competitive alternative that achieves

the same performance as using the modern multi-GPU API, while working on legacy versions of Kokkos prior to 4.3.

We note that in terms of migration costs the VLC implementation required less direct code modification than porting the application to use multi-GPU execution spaces. The code within each VLC is nearly identical to that within each MPI process, the only difference is that MPI send and receives are replaced with direct memory copies. In contrast, native multi-GPU support required more careful data management and changes to kernel launching parameters and CUDA stream synchronization throughout. While no longer required to enable single process multi-GPU support in Kokkos, VLCs provide a low-overhead within-application solution to overcome library configuration limitations that would typically require multiple processes.

7 Discussion

7.1 Loading Incompatible Libraries

OpenBLAS, Atlas [12], and MKL [58] are different implementations of the C-language BLAS standard. Applications are given the flexibility to choose an implementation based on its performance needs and environment. To reach optimal performance for various environments and application settings, a BLAS also provides many versions to choose from, such as different parallel backends, i.e., pthreads or OpenMP.

However, when multiple libraries that use BLAS are composed together in a single process, they can only dynamically link a single BLAS library regardless of the fact that the different libraries may have their own preferences. It is also not possible to compose libraries that statically link to different BLAS builds as all BLAS implementations share the same symbols, which would cause name conflicts.

The incompatibility between different BLAS implementations can be addressed by using VLCs. Applications can load different BLAS builds into multiple VLCs, where each is isolated with linker namespaces so name conflicts will not occur. Each BLAS instance will also have private static data when loaded into a VLC, making it safe to run in parallel with other BLAS libraries within a single process.

7.2 Limitations

Each VLC creates a linker namespace to avoid name conflicts and provides private static data when loading multiple instances of the same shared library into a single process. The maximum number of VLCs is limited by the number of linker namespaces provided by the dynamic linker. The dynamic linker provides tunable `glibc.rtld.nns` [20] to set the limit of linker namespaces up to 16. While 16 VLCs should be sufficient for most applications, it is possible to modify glibc to increase the limit to 32 or more.

7.3 Future Work

The current implementation of VLCs focuses on virtualizing CPU cores and GPU devices, but the underlying principle of library-level resource virtualization holds significant potential for broader applicability. A primary direction for our future research is to explore the extension of this paradigm to other critical system resources, including the virtualization of memory capacity and network interfaces, which could enable more fine-grained control over application performance in distributed and memory-bound scenarios.

While many applications achieve optimal performance through straightforward partitioning schemes (e.g., uniform distribution or workload-based allocation), these approaches are often insufficient for complex hardware topologies. For these non-trivial scenarios, such as optimizing resource allocation across multi-socket NUMA architectures, an adaptive tuning mechanism is essential. Our current auto-tuner employs a grid search policy, and this exhaustive approach can be computationally expensive. To address this, our future efforts will focus on building a machine learning model that can intelligently prune the search space, enabling the auto-tuner to converge on near-optimal configurations within a significantly reduced number of iterations.

8 Conclusion

We explore library-level virtualization with VLCs, subunits of a process that encapsulate sets of libraries and provide performance isolation between them. VLCs enable applications to partition compute resources among libraries, avoiding contention while retaining the benefits of a shared address space. Evaluations highlight how VLCs improve performance by eliminating resource contention, balancing workloads, safely parallelizing otherwise thread-unsafe code, and enabling nested parallelism that is not natively supported.

Acknowledgments

We thank our anonymous reviewers for their helpful comments, which strengthened our work. This work was partially funded by the U.S. Department of Energy, National Nuclear Security Administration Award Number DE-NA0003969; and by NSF award SHF-2505085.

References

[1] 2023. Sparse eigenvalue problems with ARPACK. https://docs.scipy.org/doc/scipy/tutorial/arpack.html

[2] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001), 15–41.

[3] Iztok Lebar Bajec. 2024. numba.get_num_threads/set_num_threads resets the value of torch.get_num_threads. https://github.com/numba/numba/issues/9387

[4] Wolfgang Bangerth. 2025. Experience converting a large mathematical software package written in C++ to C++20 modules. arXiv:2506.21654 https://arxiv.org/abs/2506.21654

[5] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. 2013. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 239–252. doi:10.1145/2465351.2465375

[6] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. 2015. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (San Francisco, California) *(PMAM '15)*. Association for Computing Machinery, New York, NY, USA, 96–105. doi:10.1145/2712386.2712400

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[8] Chao Chen, Severin Reiz, Chenhan D. Yu, Hans-Joachim Bungartz, and George Biros. 2021. Fast Approximation of the Gauss–Newton Hessian Matrix for the Multilayer Perceptron. *SIAM J. Matrix Anal. Appl.* 42, 1 (2021), 165–184. doi:10.1137/19M129961X

[9] Chao Chen, Severin Reiz, Chenhan D. Yu, Hans-Joachim Bungartz, and George Biros. 2021. Fast Approximation of the Gauss–Newton Hessian Matrix for the Multilayer Perceptron. *SIAM J. Matrix Anal. Appl.* 42, 1 (2021), 165–184. doi:10.1137/19M129961X

[10] Jong Youl Choi, Choong-Seock Chang, Julien Dominski, Scott Klasky, Gabriele Merlo, Eric Suchyta, Mark Ainsworth, Bryce Allen, Franck Cappello, Michael Churchill, Philip Davis, Sheng Di, Greg Eisenhauer, Stephane Ethier, Ian Foster, Berk Geveci, Hanqi Guo, Kevin Huck, Frank Jenko, Mark Kim, James Kress, Seung-Hoe Ku, Qing Liu, Jeremy Logan, Allen Malony, Kshitij Mehta, Kenneth Moreland, Todd Munson, Manish Parashar, Tom Peterka, Norbert Podhorszki, Dave Pugmire, Ozan Tugluk, Ruonan Wang, Ben Whitney, Matthew Wolf, and Chad Wood. 2018. Coupling Exascale Multiphysics Applications: Methods and Lessons Learned. In *2018 IEEE 14th International Conference on e-Science (e-Science)*. 442–452. doi:10.1109/eScience.2018.00133

[11] Jan Ciesko. 2023. kokkos-remote-spaces Github. https://github.com/kokkos/kokkos-remote-spaces

[12] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35. doi:10.1016/S0167-8191(00)00087-9 New Trends in High Performance Computing.

[13] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. doi:10.1109/99.660313

[14] John R Douceur, Jeremy Elson, Jon Howell, and Jacob R Lorch. 2008. Leveraging legacy code to deploy desktop applications on the web.. In *OSDI*, Vol. 8. 339–354.

[15] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*. 18–24. doi:10.1109/XSW.2013.7

[16] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. 1995. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 251–266.

[17] Maurice S Fabien, Matthew G Knepley, Richard T Mills, and Béatrice M Riviére. 2019. Manycore parallel computing for a hybridizable discontinuous Galerkin nested multigrid method. *SIAM Journal on Scientific Computing* 41, 2 (2019), C73–C96.

[18] Henrique Fingler, Amogh Akshintala, and Christopher J Rossbach. 2019. USETL: Unikernels for serverless extract transform and load why should you settle for less?. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. 23–30.

[19] Matteo Frigo and Steven G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[20] GNU. 2024. Dynamic Linking Tunables. https://www.gnu.org/software/libc/manual/html_node/Dynamic-Linking-Tunables.html

[21] Taylor Goodhart. 2023. dlmopen()'ing a shared library that dlopen()'s a non-existent library during initialization returns prematurely. https://sourceware.org/bugzilla/show_bug.cgi?id=31164

[22] Taylor Goodhart. 2023. libtbb.so cannot be used with dlmopen(). https://github.com/oneapi-src/oneTBB/issues/1283

[23] Niklas Hambüchen. 2020. Could you elaborate on the combination of OpenBLAS with multi-threading? https://github.com/OpenMathLib/OpenBLAS/issues/2543

[24] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 363–396.

[25] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 131–143.

[26] Jon Howell, Bryan Parno, and John R Douceur. 2013. How to Run {POSIX} Apps in a Minimal Picoprocess. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 321–332.

[27] Itseez 2014. *The OpenCV Reference Manual* (2.4.9.0 ed.). Itseez.

[28] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji. 2019. BOLT: Optimizing OpenMP parallel regions with user-level threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 29–42.

[29] Bart Jacob. 2009. SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems. https://www.redbooks.ibm.com/redpapers/pdfs/redp4469.pdf

[30] Yuxuan Jiang. 2023. Conda Pytorch set processor affinity to the first physical core after fork. https://github.com/pytorch/pytorch/issues/99625

[31] Michael Kerrisk. 2024. dlopen(3) — Linux manual page. https://man7.org/linux/man-pages/man3/dlopen.3.html

[32] Samuel Khuvis, Karen Tomko, Jahanzeb Hashmi, and Dhabaleswar K Panda. 2020. Exploring hybrid mpi+ kokkos tasks programming model. In *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+ X (PAW-ATM)*. IEEE, 66–73.

[33] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-Tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A system for massively parallel hyperparameter tuning. *Proceedings of machine learning and systems* 2 (2020), 230–246.

[34] Xiaoye S. Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.* 31, 3 (Sept. 2005), 302–325. doi:10.1145/1089014.1089017

[35] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance.. In *OSDI*. 49–64.

[36] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-in-time summoning of unikernels. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 559–573.

[37] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 461–472.

[38] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

[39] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux j* 239, 2 (2014), 2.

[40] Vivek Das Mohapatra. 2019. libcapsule. https://gitlab.collabora.com/vivek/libcapsule.

[41] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 456–471.

[42] Carlos O'Donell. 2015. RFC: Treat RTLD_GLOBAL as unique to namespace when used with dlmopen. https://patchwork.ozlabs.org/project/glibc/patch/55A73673.3060104@redhat.com/

[43] Heidi Pan, Benjamin Hindman, and Krste Asanović. 2010. Composing Parallel Software Efficiently with Lithe. *SIGPLAN Not.* 45, 6 (jun 2010), 376–387. doi:10.1145/1809028.1806639

[44] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[45] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the library OS from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 291–304.

[46] Ilan Price, Alvaro Sanchez-Gonzalez, Ferran Alet, Tom R Andersson, Andrew El-Kadi, Dominic Masters, Timo Ewalds, Jacklynn Stott, Shakir Mohamed, Peter Battaglia, et al. 2025. Probabilistic weather forecasting with machine learning. *Nature* 637, 8044 (2025), 84–90.

[47] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The next stage of linux's dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 7–13.

[48] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.".

[49] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 233–248.

[50] Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 546–558.

[51] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 121–135.

[52] Ahnjae Shin, Joo Seong Jeong, Do Yoon Kim, Soyoung Jung, and Byung-Gon Chun. 2022. Hippo: sharing computations in hyper-parameter optimization. *Proc. VLDB Endow.* 15, 5 (Jan. 2022), 1038–1052. doi:10.14778/3510397.3510402

[53] Claudio E Torres, Hossein Parishani, Orlando Ayala, Louis F Rossi, and L-P Wang. 2013. Analysis and parallel implementation of a forced N-body problem. *J. Comput. Phys.* 245 (2013), 235–258.

[54] Christian Trott. 2018. Is it possible to use on-node multiple GPUs without MPI? https://github.com/kokkos/kokkos/issues/1610

[55] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[56] Pauli Virtanen. 2023. scipy.sparse.linalg.eigen.arpack.arpack. https://github.com/scipy/scipy/blob/3d3358a7740727fb8ad9cf4740de34c3bf71b287/scipy/sparse/linalg/_eigen/arpack/arpack.py#L1099

[57] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. doi:10.1038/s41592-019-0686-2

[58] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. *Intel Math Kernel Library*. Springer International Publishing, Cham, 167–188. doi:10.1007/978-3-319-06486-4_7

[59] Nicolas Weber. 2023. PyTorch's packaged libgomp causes significant performance penalties on CPU when used together with other Python packages. https://github.com/pytorch/pytorch/issues/98836

[60] Florian Weimer. 2019. Bug 24776: pthread_key_create, pthread_setspecific are incompatible with dlmopen. https://sourceware.org/bugzilla/show_bug.cgi?id=24776.

[61] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. doi:10.1109/IPDPS.2008.4536359

[62] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*. 199–211.

[63] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 31–44.

[64] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th international conference on parallel and distributed systems*. IEEE, 684–691.

[65] Fan Zhang, Melissa Petersen, Leigh Johnson, James Hall, and Sid E O'Bryant. 2022. Hyperparameter tuning with high performance computing machine learning for imbalanced Alzheimer's disease data. *Applied Sciences* 12, 13 (2022), 6670.

[66] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *The Journal of Open Source Software* 4, 37 (2019), 1370.

[67] Yiming Zhang, Chengfei Zhang, Yaozheng Wang, Kai Yu, Guangtao Xue, and Jon Crowcroft. 2021. Kylinx: Simplified virtualization architecture for specialized virtual appliances with strong isolation. *ACM Transactions on Computer Systems (TOCS)* 37, 1-4 (2021), 1–27.