Performance Characterization of Python Runtimes for Multi-Device Task Parallel Programming

William Ruys[1], Hochan Lee[1], Bozhi You[1], Shreya Talati[1], Jaeyoung Park[1], James Almgren-Bell[1], Yineng Yan[1], Milinda Fernando[1], Mattan Erez[1], Milos Gligoric[1], Martin Burtscher[2], Christopher J. Rossbach[1], Keshav Pingali[1], George Biros[1]

[1]The University of Texas at Austin, Austin, TX, USA.
[2]Texas State University, San Marcos, TX, USA.

Abstract

Modern Python programs in high-performance computing call into compiled libraries and kernels for performance-critical tasks. However, effectively parallelizing these finer-grained, and often dynamic, kernels across modern heterogeneous platforms remains a challenge. This paper designs and optimizes a multi-threaded runtime for Python tasks on single-node multi-GPU systems, including tasks that use resources across multiple devices. We perform an experimental study which examines the impact of Python's Global Interpreter Lock (GIL) on runtime performance and the potential gains under a GIL-less PEP703 future. This work explores tasks with variants for different different device sets, introducing new programming abstractions and runtime mechanisms to simplify their management and enhance portability. Our experimental analysis, using tasks graphs from synthetic and real applications, shows at least a 3× (and up to 6×) performance improvement over its predecessor in scenarios with high GIL contention. Our implementation of multi-device tasks achieves 8× less overhead per task relative to a multi-process alternative using Ray.

Keywords: GPU tasking systems, HPC in Python, GPU programming in Python, Global Interpreter Lock, Task parallel programming

1 Introduction

CPython's rich ecosystem of C-extension modules has allowed developers to quickly weave together powerful applications that leverage the efficiency of native code. Composable libraries of optimized primitives [1–3] and kernel generators [4–6] allow domain-specific modules to be developed effectively from within Python itself. However, parallelizing Python applications on modern heterogeneous platforms remains a challenge. Such applications must seamlessly coordinate across various devices, often operating with distinct memory spaces and programming models between them. A

popular way to manage these complexities is through a task-based programming model where an application is expressed as a directed acyclic graph (DAG) of tasks. Each task is a unit of work in the application, with data dependencies, resource constraints, and potentially multiple implementations depending on which devices it supports. A task-based approach reduces the developer's burden. The tasking runtime selects a device and task implementation, manages data movement between dependent tasks, and schedules the work for execution. This hides the complexities involved in manually partitioning work, communicating across devices, and coordinating execution through lower-level primitives on threads, processes, and concurrent hardware queues (e.g., CUDA/HIP streams) from the application programmer.

Many heterogeneous tasking systems have been developed for managing these complexities [7–9]. When it comes to Python, finer-grained multithreaded parallelism has faced a unique challenge: the Global Interpreter Lock (GIL) in the CPython interpreter allows only one thread at a time to execute Python bytecode, effectively serializing all work except C-extension library calls and I/O. While alternative, GIL-free Python implementations exist [10], they lack compatibility with popular CPython libraries and have not gained widespread adoption. For this reason, managing GIL interference remains a critical performance challenge for parallel Python applications. Recently, the CPython Steering Committee has approved PEP 703 [11] to make the GIL optional, with an build to be included in Python 3.13. But its wider adoption is questionable for the foreseeable future due to thread safety and compatibility concerns in the Python ecosystem. We study the feasibility and limitations of multithreaded parallelism in modern Python and under the PEP703 proposal to demonstrate that practical performance is both achievable now and will grow significantly in the future. We discuss several Python-specific task-based runtime systems and evaluate their performance using several DAGs and task granularity experiments.

Multi-device kernels and libraries that manage sets of devices internally—such as cuBLASmg [12] and cuFFTmg [13], are becoming prevalent in many problems in HPC. Examples include hierarchical algorithms like multigrid and the fast multipole method, multi-physics applications, and training and inference of neural networks. To support using these kernels inside of tasks, we propose a model for *multi-device tasks*: tasks that require multiple compute devices. Scheduling these tasks comes with new challenges: effectively managing variants of the same task over different device sets; managing data movement and partitioning; and ensuring hardware portability. Some existing runtimes, such as Ray [14] and PyCOMPSs [15], allow users to define tasks that require a certain number of CPU cores and GPU devices. PyCOMPSs additionally supports task variants with different resource constraints for each implementation. However, to our knowledge, all existing multi-device task solutions are multi-process and none of the available systems, in Python or otherwise, support the *run-ahead* execution and synchronization of multi-device tasks directly on hardware queues through CUDA/HIP events or provide an integrated data model for prefetching task inputs onto GPU devices. We provide this support, extend our programming model with *task environment contexts*, *specialized functions*, and *heterogenous distributed arrays* to enable the modular and portable development of tasks with internally distributed workloads, and optimize our throughput with a *multi-device queue* launching mechanism.

We implement the proposed runtime for Parla [16], a single-process, heterogeneous-device, task-based runtime. Parla provides a baseline extendable framework and API for multithreaded execution of asynchronous dynamic task graphs on heterogeneous nodes. In this paper we have replaced Parla's entire backend and extended the interface to support multi-device tasks. To distinguish the two systems, we refer to Parla's original runtime of Lee et al.[16] as PyRun and the new one as CyRun. In summary, we make the following contributions:

- Using Dask and the two Parla runtimes, we study task parallelism for compute-bound DAGs in CPython to understand the effects of GIL contention on runtime design, sensitivity to task workload, and potential gains under a `PEP703` future. (§2, §5)
- We propose a programming model and scheduling mechanisms for multi-device tasks along with shared-view arrays for heterogeneous multi-device memory management. We compare our runtime with Ray. (§4)

To our knowledge CyRun has certain unique features for DAGs that have fine-grain multi-GPU tasks: it provides run-ahead scheduling; and it is a single process. Following [17] we use several synthetic graphs in our experiments (§5), as well as real-world applications. The new runtime will be made publicly available.

2 Challenges in Runtime Design

2.1 Background

Despite the limitations imposed by the GIL, parallelism can be still achieved in CPython through several methods: ❶ calling external non-Python kernels that release the interpreter lock during their execution; ❷ submitting asynchronous jobs, like I/O requests or GPU kernels, that can be processed externally while other workloads use the interpreter; ❸ using multiple processes, each with their own interpreter and GIL; ❹ using multiple sub-interpreters, each with their own GIL, within a single process; and ❺ using embedded compiled Python-like domain-specific languages, like Torchscript [18]. From a developer's perspective, gaining parallelism through ❶ and ❷ is preferable because they retain a shared memory space, allowing data to be shared between tasks without the need for serialization and inter-process copying. As tasks share the same interpreter state, they do not need to reload libraries or reinitialize data structures.

However, many Python tasking systems adopt ❸, employing multiple processes each with distinct interpreters. This introduces overheads, notably from inter-process communication and the initialization cost of loading libraries into each process. Some systems, like Ray and PyCOMPSs, mitigate these overheads by offering shared-memory buffers and key-value stores for efficient data sharing between processes. Although this strategy suits distributed systems, it may degrade single-node performance. Lee et al [16] showed that the multiprocess approach can be prohibitive for fine-grained GPU tasks that pass data between them. Recent innovations include PyTorch's TorchDeploy, which experiments with ❹. They introduced a per-thread interpreter each with a dedicated GIL. Language-level support for such parallel sub-interpreters was added in Python 3.12 [19]. Although sub-interpreters share a single process, they are still separate Python environments. Objects must be serialized and copied to communicate between them. StarPU's [20] Python interface [21] is approaching these challenges through cloudpickle serialization and a virtual shared-memory

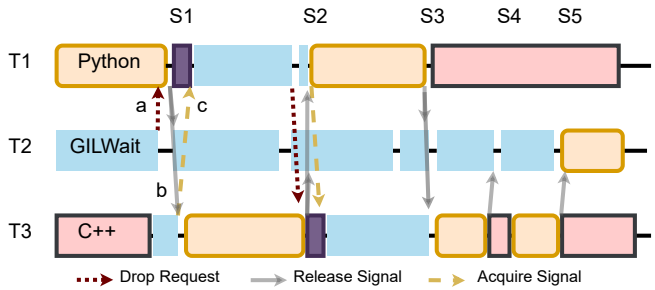

Fig. 1: A timeline of three threads running a mixed Python-C application. We illustrate how a thread (T2) can be delayed. Threads T1–T3 execute from left to right. S1–S5 indicate GIL release events. Blocks on each line represent when a thread is executing Python code (orange), executing compiled code (pink), waiting for the GIL (blue), or blocked until another thread acquires the GIL (purple). At S1, T2 and T3 are waiting for GIL access. After T2 has waited 5ms, (a) it sends a drop-request to T1. T1 is interrupted and releases the GIL. (b) It signals waiting threads. Because control was dropped through a request, (c) T1 is blocked until another thread acquires the lock. Due to GIL ownership being passed under the waiting interval, T2 is unable to issue a drop-request for the remainder of this timeline.

manager between sub-interpreters. While the consequences of the GIL on the performance of I/O and simple multi-threaded CPU applications have been explored [22], we have found its effects on general task parallelism to be understudied. In this work, we focus on ❶ and ❷ and show how far we can push performance for heterogeneous systems under CPython's constraints.

2.2 A Primer on GIL Behavior

Understanding GIL behavior is critical to multithreaded runtime design. The GIL is a mutex around the loop that drives bytecode execution. Any additional threads that attempt to access this region are suspended until it is available. In CPython's original policy, the controlling thread would only release the GIL after a constant number of bytecode instructions, regardless of time spent. A thread that released the GIL, either manually or after this threshold, would signal any waiting threads and immediately attempt to reacquire the lock. For CPU bound workloads, this lead to a bias where the previously owning thread would acquire the lock.

In Python 3.2, this was changed to a system in which waiting threads can issue `drop_requests`. A waiting thread can only issue a `drop_request` after each time it waits for a fixed time period (5ms by default), and only if ownership of the GIL has not changed during its last period. This signals the owning thread to release the GIL after it's next bytecode instruction and prevents this thread from reacquiring the lock until it has switched to a new owner. The Python interpreter does not provide a scheduler and does not assign priorities between threads waiting for the GIL. All threads are treated the same regardless of how many prior waiting intervals they have completed. The operating system's (OS) scheduling policy determines which threads are woken each time the interpreter lock is released.

In Figure 1, we present a simple example to highlight how this behavior could potentially delay threads from executing for long periods of time, especially when the GIL is released frequently by calls into C kernels. We show a timeline of GIL exchanges

between three threads for a mixed Python-C application. At the first switch (S1), the thread that issues the `drop_request` may not be the thread that acquires the GIL. Thread 3 may execute before Thread 2. At the second switch (S2), Thread 2 could not issue an earlier `drop_request` because a switch occurred during its last waiting interval. At the third switch (S3), Thread 1 enters a C-extension kernel and releases the GIL explicitly. Because this was a manual release, no `drop_request` occurs. The figure highlights that it is possible for spurious wakeups on waiting threads to occur and reset their waiting interval. Even if the spurious wakeup does not occur, Thread 2 would be unable to issue a `drop_request` until S4, as a switch occurred while it was waiting. Finally, at the fourth switch (S4), Thread 3 enters a short C-extension kernel that releases the GIL. Because no `drop_request` is active, it can potentially complete and reacquire the lock before Thread 2 wakes. For three threads this is an unlikely scenario, Thread 2 could have been chosen at any of the five switches. However, there are no mechanisms that make Thread 2 more likely to acquire the lock as it waits longer. Frequent calls into short C-extension kernels prevent threads from issuing `drop_requests` and lead to long waiting periods. As the number of active threads increases, such cases become more frequent.

2.3 Designing Multithreaded Runtimes for Python tasks

We seek to understand *how fast tasks can be launched* and optimize *the minimal granularity at which Python tasking is effective*. To investigate this design space, we perform a comparative study of minimal runtimes for multithreaded execution of Python tasks. A runtime for dynamic task graph execution must, at minimum, be able to spawn new work, satisfy task resource and precedence constraints, and assign tasks to worker threads. Each presented minimal runtime in this section supports only these three operations for CPU tasks. Runtimes can be written in Python itself or driven from an external compiled language; We present the benefits, trade-offs, and design choices available to each.

Python Runtimes. Pure Python runtimes are portable across Python implementations and can easily interface with Python objects. However, all worker threads that execute Python code contend for the interpreter with the scheduler. This increases scheduling latency, reduces parallelism, and leads to stalls when the scheduler is unable to assign work to threads sufficiently often. *Dask-threading* is a widely used runtime for Python tasking. It runs the scheduler in a loop from the main application thread. At each iteration, the scheduler launches a batch of ready tasks, up to one per thread, and pops a completed task from a global queue. The completed task is processed on the main thread. Dependent tasks are notified by removing the dictionary key for the completed dependency. Newly ready tasks, those without remaining dependencies, are moved to the list of launchable tasks. Worker threads concurrently add tasks to the completed queue when finishing a task. Tasks are launched by submission to Python's standard library `ThreadPoolExecutor`, which adds tasks to a global work queue. Each worker thread continuously drains work from this queue until empty. There are no runtime mechanisms that interrupt the GIL to switch between workers. As the runtime is driven by a dedicated Python thread, newly ready tasks can only be launched when the scheduler thread wins control of the interpreter. If the global work queue is not filled fast enough to feed worker threads this introduces latency into the schedule.

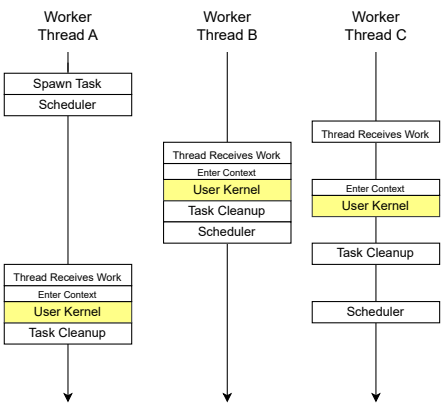

Fig. 2: Outline of PyMinimal-Callback runtime.

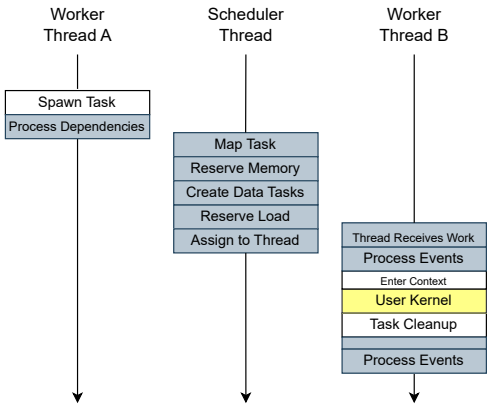

Fig. 3: Outline of CyMinimal-Optimized Runtime.

PyMinimal is our design and implementation. We provide a baseline implementation that runs the scheduler from a *Dedicated* thread. We do not use Python's `ThreadPoolExecutor`; Tasks are assigned directly to available threads and each thread may only be assigned one task at a time. Worker threads without assigned tasks are suspended until they are assigned new work. We notify dependent tasks of a task's completion on the worker thread immediately when a task finishes by decreasing their counters of remaining dependencies. We developed a second implementation, *Callback*, that attempts to alleviate GIL contention by running scheduler operations on threads where the interpreter is already acquired. Instead of having a dedicated scheduler thread, the scheduler is invoked from worker threads. Each time a task is spawned or completed, the active thread checks resources and launches a batch of ready tasks onto waiting worker threads. Threads are maintained on a stack and the active thread will assign to itself first. This design is summarized in Figure 2 where white regions represent operations in Python and yellow regions are user operations in the task body that may release the GIL.

External Runtimes. In contrast to pure Python implementations, runtimes driven by an external compiled language may not need to vie for the interpreter to notify and assign tasks. This allows them to be more responsive to state changes, like freed resources, newly spawned, and newly ready tasks. They can also use more efficient data structures and algorithms due to synchronization primitives that are not available in Python. However, each callback from Python into the external runtime is treated as a single byte-code operation. It must either release or hold the GIL for the entire duration. If each call releases the GIL for too brief a period, it can lead to cases like Figure 1 where drop requests are prevented from being issued. On the other-hand, if callbacks hold the GIL for too long, workers will be unable to start their assigned tasks. Any newly created `drop_requests` will be ignored until the callback completes.

CyMinimal is our external runtime, written in C++. We consider a few variants of its design. In the *Dedicated* and *Callback* designs, workers and task objects are still orchestrated at the Python layer. These designs are drop-in C++ implementations of the interfaces used in the PyMinimal variants. Each task creation, dependency

addition, dependent notification, resource release, and task-to-worker assignment is a separate call into the C++ runtime from Python. Because the workerpool is managed in Python, the C++ launcher must acquire the interpreter when assigning tasks to Python worker threads. When driven from a dedicated C++ thread, these short GIL acquisitions cause thrashing. In *Callback*, the GIL is already held by the thread that runs the launcher, which leads to a large performance improvement on short tasks. However, in these implementations, nearly all C++ function calls are too short to gain a benefit from releasing the GIL. We consider two optimized designs, using *Dedicated* as the base. The first, *Fused*, moves the bulk of the task orchestration into the C++ layer. This allows us to combine the interactions between Python and the runtime into only 3 function calls per task: spawning, task completion, and task-to-worker assignment. On tasks with many dependencies, which have longer spawn and completion times, this provides a performance improvement. On our 40×40 block Cholesky graph with 1ms tasks, releasing the GIL for task completion provides an 8% improvement relative to holding it for each task. The second design, *Optimized*, goes further and additionally moves the worker pool into the C++ layer. This removes the need to acquire the interpreter when assigning Python tasks to workers greatly reducing contention. This provides the minimum possible 3 GIL-acquisitions per-task. Our final design is summarized in Figure 3. A throughput comparison for all runtime designs is shown in Table 1, where bold values are the best performing implementation.

Table 1: Throughput comparison of runtime designs. Speedup over serial execution of 1024 GIL-releasing independent CPU tasks (each 1ms or 500μs). Using a default switch interval and no GIL Interrupt methods.

		Speedup (1ms Tasks)		Speedup (500μs Tasks)	
Runtime	Options	8 Threads	16 Threads	8 Threads	16 Threads
PyMinimal	Dedicated	6.5×	11.1×	5.4×	6.9×
	Callback	6.5×	11.2×	5.4×	7.4×
CyMinimal	Dedicated	6.8×	11.9×	5.8×	6.3×
	Callback	6.9×	12.1×	5.9×	9.5×
	Fused	6.9×	12.2×	6.1×	9.5×
	Optimized	**7.0**×	**12.6**×	**6.2**×	**10.2**×
Dask	Base	6.6×	11.0×	5.1×	6.5×

One of the unique challenges when scheduling dynamic task graphs in Python is interleaving newly spawned tasks into the current execution. As spawning each new task only requires a few microseconds, starting newly created tasks that might be launchable without the latency of the 5ms GIL switching interval is critical to achieving high performance. We can see this effect by tuning the GIL switching interval with `sys.setswitchinterval` to a lower value (e.g., 5μs). Unfortunately this performance gain is lost if running larger tasks due to overabundant drop-requests. To address this in a more portable manner, we manually interrupt the interpreter on each thread whenever it has reached a spawning quota (by default equal to the number of cores). In essence, if a task is creating a bunch of new work, we pause to give this work a

chance to start running. Throughput results for this strategy are shown in Table 2. Guided by the performance profiling of these minimal runtimes, *we reimplement and extend the full Parla library using a CyMinimal-Optimized design*. We call this full runtime CyRun. Performance results using this full runtime on a suite of graphs and kernels are presented for this updated runtime in Section 5.2.

3 Parla Overview

Our second main contribution is the definition and mechanisms to support *multi-device tasks*. We introduce and test these features in the context of the Parla programming model. Parla is a library for dynamic online DAG execution of heterogeneous tasks. Our proposed CyRun runtime implements the full Parla API and extends it to the multi-device cases. In this section, we briefly summarize the relevant parts of the original Parla library to the current work.

Tasks, in Parla, are created by decorating a Python function with a `@spawn` decorator, shown in Listing 1. This provides: (1) the task's name within an indexable namespace; (2) a set of dependencies; (3) a list of `places` where the task may run; and (4) resource constraints like `memory` & `thread` usage. Each element of `places` is a device that a variant of the task could execute on. For example, a task that defines `places=[cpu, gpu]` can be scheduled at runtime to launch on either a CPU or a GPU device. The `places` list may contain specific devices or generic architecture types.

Parla supports data management for CuPy and NumPy arrays. These can be moved with *manual* annotations, or specified in `spawn`, as read and read-write dataflow dependencies, to allow *scheduled* prefetching of input data. Scheduled data movement manages replication and coherency between devices. GPU tasks support *run-ahead* scheduling when they only contain GPU kernels. Using CUDA/HIP events tasks are dispatched to hardware queues before their dependencies complete. Cross-stream events preserve a valid ordering of kernels and data movement. Devices can run more than one task at a time (up to their resource constraints). Concurrent GPU tasks are run on separate streams. Parla tasks may spawn tasks internally. These nested tasks do not synchronize with their parent unless specified. A task's lifecycle through the runtime is shown in Figure 4: Each spawned task is *mapped* by picking one of its `places` that satisfies its resource constraints; A task is *reserved* when memory on its target device has been provisioned for itself and it's data; Finally, a task is *launched* onto

Table 2: Throughput comparison of GIL interrupt methods. Speedup over serial execution of 1024 GIL-releasing independent CPU tasks (each 1ms or 500μs).

		Speedup (1ms Tasks)		Speedup (500μs Tasks)	
Runtime	Options	8 Threads	16 Threads	8 Threads	16 Threads
PyMinimal	5μs Interval	7.0×	11.7×	5.5×	6.6×
	Spawn Interrupt	6.4×	11.0×	5.3×	7.4×
CyMinimal	5μs Interval	7.4×	13.0×	6.4×	10.7×
	Spawn Interrupt	**7.7**×	**14.8**×	**7.0**×	**12.1**×
Dask	5μs Interval	6.6×	11.1×	5.2×	6.5×

a worker thread. Once mapped, tasks are placed into a separate work-queue for the device they have been assigned.

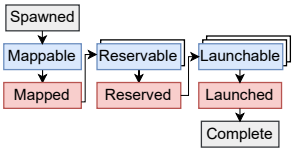

Fig. 4: **Lifecycle of a Parla task. The runtime processes events from queues (blue/top) into their resolved states (red/bottom).**

Listing 1: **Example of a Parla Task**

```
@spawn(T[i], [T[i-1]], places=[gpu])
def task():
    ...
```

4 Multi-Device Tasks

Multi-device tasks are tasks that reserve resources across multiple devices. Supporting such tasks enables application programmers to call optimized external libraries that may use resources across multiple devices from within a task. By tracking resource usage and data movement, effective parallelization of calls into such libraries becomes possible. This enables greater composability and nested hierarchical parallelism. However, designing a runtime for multi-device tasks introduces additional challenges in: (i) selecting device-sets from possible variants; (ii) scheduling device-sets of varying size due to a greater fragmentation of node resources; (iii) data management when task input is partitioned and distributed across devices; and (iv) programming portable task variants. We introduce a model for programming multi-device tasks and propose solutions to these runtime challenges.

4.1 Multi-Device Task Interface

We extend Parla's programming model to support device-sets in a task's list of valid `places`. Each *device-set* is a tuple of devices and/or architectures; For example, `(gpu(0), gpu)` is the set containing GPU 0 and any other GPU device. This set defines a one-to-many mapping where a single task requests resources across multiple devices. The shorthand, `gpu*n`, can be used to define a sets containing n devices of the same type. As in the single-device case, a single task may have multiple valid configurations. For example, a task specified with `places=[(cpu, gpu), gpu*4]` can be executed on either a set containing both a CPU and a GPU, or four GPUs. The scheduler dynamically chooses the best configuration for the current system state. Resource constraints, like memory usage, can specified for each device in each set. We introduce *task environments* and *specialized functions* as part of a proposed API to help users query and interact with tasks on device-sets. Then we discuss a data abstraction for heterogeneous memory global-address space arrays and conclude with details on the runtime mapping and scheduling of multi-device tasks.

Task Environments. As task mapping is dynamic, tasks must be able to query their current set of devices during execution. The *task environment* object provides their properties, how to dispatch work to each of them, and potentially how to synchronize their execution. This is shown at L7 in Listing 3. For each device assigned to the task, this provides the streams and events that have been assigned by the runtime. The size of an environment is the number of devices it contains. The environment is a sliceable Python context manager [23]. Entering the task environment, via `with`, sets all associated objects active on the local thread. Workloads inside a multi-device task may need to dispatch different kernels over different subsets of its devices. To support

Listing 2: **Function Variants**

```
@specialize
def fft(A):
    fA = numpy.fft(clone_here(A))

@fft.variant(arch=gpu, max=4)
def mgpu_fft(A):
    env = get_current_env()
    A = clone_here(A)
    cufftmg(A, env.devices)

```

Listing 3: **Multi-Device Tasks**

```
@spawn(places=[cpu, gpu*4])
def task1():
    C = fft(A)

@spawn(placement=[gpu*4])
def task2():
    env = get_current_env()
    #task reserves 4 GPUs
    with env[:2] as half:
        #restricts to 2 GPUs
        fA = fft(A)
        half.synchronize()
```

Listing 4: **Vector Addition using CrossPy**

```
@spawn(places=[(cpu,gpu)])
def task():
    e = get_current_env()
    ax = xp.array(a)
    bx = xp.array(b)
    cx = ax + bx
```

Listing 5: **K-Means using CrossPy**

```
@spawn(places=[gpu, gpu*2, gpu*4])
def kmeans():
   e = get_current_env()
   # Redistribute inputs as CrossPy arrays
   xpoints = clone_here(points)
   xlabels = clone_here(labels)
   xclusters = clone_here(clusters)
   for i in range(iterations):
      # Assign points to nearest clusters
      label(xpoints, xclusters, xlabels)
      # Determine new cluster means
      reduce(xpoints, xclusters, xlabels)
```

Listing 6: **CrossPy Communication Interface**

```
# (A) Python Assignment
dst[dst_ids] = src[src_ids]
# (B) Reusable All2All
exchange = xp.alltoall(dst_ids, src_ids)
exchange(dst, src)
exchange(dst, src)  # Can be reused
```

this, task environments can be split, nested, joined, and looped over. An example of this is seen in Listing 3. There a sub-environment context of the 4-GPU task (L7) is created containing the first two GPUs (L9). The sub-environment sets the visible devices for any specialized functions called within its scope.

Specialized Functions. A *specialized function* is a function that has been overloaded with variants for specific device or architecture sets. When a specialized function is called within a task environment, it dispatchs to the implementation defined for the most specific matching device-set. If task environments are nested, the function only sees the innermost one. In Listing 3, function variants are used to define a single `fft` routine that can be invoked in either single-CPU or multi-GPU environments. The variant implementations are provided by appropriate external libraries for each.

CrossPy Heterogeneous Arrays. We introduce *CrossPy*, an array abstraction that provides a shared index space over heterogeneously partitioned arrays. A CrossPy array is made of partitioned blocks, either NumPy or CuPy arrays, that provide a unified view to the user. Listing 4 is an example that creates two CrossPy arrays from input arrays a and b, which are either a NumPy or CuPy array (L4 and L5). The vector addition on L6 is computed over both the CPU and GPU in the task. By default, a CrossPy array will partition an input array evenly among the active devices in a 1D row-partitioning. Non-uniform partitions can be defined by providing coloring maps, as a function from array indexes to devices, to the array. CrossPy

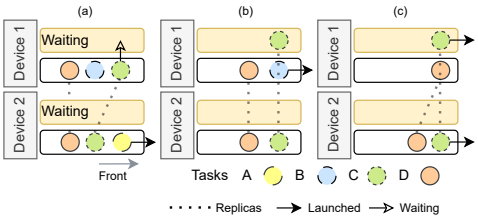

Fig. 5: Multi-device tasks insert replica tasks into each device's work queue. Tasks can only be launched if all replicas have reached the head of their respective queues.

Listing 7: **Multi-Device Queue Algorithm**

```
1  if len(waiting_queue) > 0:
2      head = waiting_tasks.head()
3      if head.remaining_instances() < 1:
4          return waiting_tasks.pop()
5
6  if len(task_queue) > 0:
7      head = task_queue.head()
8      head.decrement_instances()
9      if remaining_instances <= 1:
10         return head
11     else:
12         waiting_queue.push(head)
13         return task_queue.pop()
```

arrays can be manually moved between task environments with `clone_here()`. This will copy the first partition to the first local device, the second partition to the second device, and so on. If the current CrossPy array and the target task environment have a different number of devices, the array will be repartitioned. While *scheduled* data movement is not currently supported by CrossPy, multi-device tasks can use the existing Parla array annotations in `spawn` to prefetch non-distributed arrays onto each of their devices. When using these annotations the data locations are tracked by the runtime. Listing 5 shows an example of CrossPy usage in a multi-GPU K-Means implementation over a task with three possible variants: 1 GPU, 2 GPUs, and 4 GPUs. Points are wrapped by CrossPy and partitioned over the devices in a balanced manner (L5/7). The user-written functions `label()` and `reduce()` execute in parallel on the partitioned data across the device-set. CrossPy supports scattering operations over arbitrary index sets. These can be executed through Python assignment and slicing or via a static *MPI all-to-all*-like interface. To optimize for repeated communication with the same source to destination index mapping, CrossPy caches the procedure. It can be reused as a callable operation. Performance evaluations are discussed in Section 5.3.

4.2 Multi-Device Task Mapping

During DAG execution, the runtime must pick both the variant of the task, from `places`, and specific devices to fill the chosen device-set. This is done when the task is *mapped*, as shown in Figure 4. A task may be mapped as soon as all of its dependencies have been mapped. For each set in `places`, we find and calculate a suitability score for a group of selected devices that can satisfy its constraints. We map the task to the variant and device-set with the highest score. The process to chose and score devices is as follows: Specific devices are chosen for each architecture slot in the task variant one-by-one. The mapping function iterates potential devices of that architecture type and scores them based on (i) the expected task workload already mapped but not yet completed on that device, and (ii) the cost to move the task's data from their expected locations to this device. We use the same heuristic that Parla uses to map single-device tasks [16]. This heuristic maintains a table of the expected locations of all tracked data objects under the assumption that all prior mapped tasks will complete before the current task. The process is repeated until all slots in the device-set have been filled. The score for the set is the average of it's device's scores. This prefers task variants that use fewer devices unless the data movement cost would be extreme.

4.3 Multi-Device Task Launching

Tasks are inserted into per-device priority queues once they have been mapped. This enables high-throughput for single-device tasks, as it avoids scanning a global queue for ready tasks that can fit onto each device. We adapt these per-device queues to support multi-device tasks. Our proposed Multi-device Queue (MDQ) algorithm inserts replicas of the task into all device queues associated with its mapped device-set. We guarantee that tasks are always be dequeued in a valid ordering, without deadlock, across all devices, even if each queue has a different relative ordering of tasks. This allows safe concurrent insertion of tasks into device queues from multiple-threads, without locking all of them, and allows each device to sort their tasks by a potentially different priority algorithm. Our algorithm is described in Listing 7. It can be described simply: A task may only launch after all of its replicas have reached the head of their respective queues. If a replica reaches the head of a queue before its siblings, then it is removed from the head and added to a list of waiting (not yet dequeue-able) multi-device tasks. If the head of this waiting queue has replicas remaining in other device queues, it is non-blocking. Further dequeue requests from this queue will proceed to pull from the head. When the head of the waiting queue has no remaining replicas in other device queues, it becomes blocking and must be the next task that this device launches. We show an example of task's launching in Figure 5 over three snapshots: (a), (b), and (c). In each snapshot, a task is dequeued for Device 1 and then for Device 2. For simplicity, we assume launched tasks take all resources on their devices and complete before the next snapshot. At time (a), Task C is moved to Device 1's waiting queue as its replica in Device 2's queue remains. As Task A has no replicas it may be launched onto Device 2. At (b), Task B is launched on Device 1. As Device 1 is now busy, the multi-device Task C cannot launch. At the final time (c), Task C is launched from the head of Device 1's waiting queue.

Evaluation of multi-device tasks is presented in Section 5.3.

5 Evaluation

5.1 Experimental Setup

Our evaluation is organized into two sections. In §5.2, we present a detailed study of overheads in multi-threaded task parallelism in Python. In §5.3, we evaluate the proposed multi-device extension and it's performance. Performance results were collected on the Frontera system at the Texas Advanced Computing Center (TACC) [24]. For Table 1 & 2 we used a CLX node equipped with two 2.7GHz Intel Xeon Platinum 8280s CPUs (14 cores/CPU). The remaining experiments were collected on a single node with four NVIDIA Quadro RTX 5000 GPUs and two Intel Xeon E5-2620 v4 CPUs (8 cores/CPU). Timing results are collected as the median of 5 runs. Our evaluations are summarized below.

- **Study of Python Performance** [Section 5.2]
  - DAGs and Task workloads (Figures 8, 7)
  - Evaluation under `nogil` CPython (Figure 9, 10)
- **Multi-device Evaluation** [Section 5.3]
  - Multi-device tasking overheads (Table 4)
  - Communication costs of CrossPy (Table 6)
  - Multi-device tasks in applications (Table 7)

5.2 Python Overheads

We evaluate our runtime, and the performance sensitivity of multithreaded Python applications, with a collection of synthetic task graphs. Evaluations are performed with full runtime systems. Relative to the minimal prototypes PyMinimal and CyMinimal, PyRun and CyRun have higher overheads due to additional checks of task metadata at the Python layer, environment configuration, and DAG processing in the runtime. This is especially pronounced in the case of PyRun, as it has more Python-level locks in its runtime than PyMinimal. If the GIL switches from the thread that holds a lock, progress is delayed until the holding thread is able to reacquire the interpreter. Dask is used as baseline reference for multithreaded performance. When used with its threading backend, Dask does not support heterogeneous devices or resource constraints. Ray is not used as a comparison point in §5.2 as it is a multi-process runtime.

Task Workloads. A task's *workload* is it's *size*, the total execution time, and *how this time is used*. Specifically, we consider tasks that perform a sequence of GIL-releasing operations (which we call *kernels*) that may require the GIL between each of them. This workload is specified by three parameters: the total execution time in milliseconds, the number of kernels per task, and the fraction of the total task time that requires the interpreter. We call this fraction the *GIL Hold Ratio*. The total task time is evenly split the kernels within each task. We evaluate a GIL Hold Ratio between 0-10% of the total time, a range of kernels per task (1-10), and a range of task sizes (0.5-64ms). All tasks within a DAG are given the same workload. This range of workloads was chosen to cover sizes where per-task overheads begin to dominate for all considered runtimes. Figure 6 shows CyRun's performance for kernels a GIL Hold Ratio of 0%. Although the task workload itself does not require the interpreter, it must be acquired on the worker thread to start and complete the task. Kernels in the range of 1-10s of milliseconds appear commonly in many NumPy and CuPy applications.

Task Dependencies (DAG). We consider a collection of DAGs adapted from the TaskBench benchmark [17]. Details are given in Table 3.

DAG	Description	Details
Independent	Independent tasks	1,000 tasks
Stencil	1D 3-point stencil	32 tasks wide, 1,000 tasks deep
Sweep	Sweep tasks	32 tasks wide, 1,000 tasks deep
FFT	FFT Butterfly graph	257 wide, 9 steps
Scatter-Reduction	Binary tree and its inverse	256 nodes at the widest point
Cholesky	Dependencies of a block Cholesky factorization	40×40 blocks
Map-Reduce	Fan-in/fan-out structure	$500 + 500m$ tasks for m workers

Table 3: Test Suite of DAGs

We scale to 15 threads, reserving a core for the scheduler thread itself. Parallel efficiency is measured relative to the ideal serial execution of the application. We highlight our main observations from this parameter study below.

GIL Contention. Figure 8 shows the average time each task waits for the GIL after it's kernels complete. As one expects, increasing task size decreases GIL contention. The runtime optimizations made in CyRun show a significant improvement relative to PyRun in alleviating the baseline contention rate. With task sizes of 16ms, each

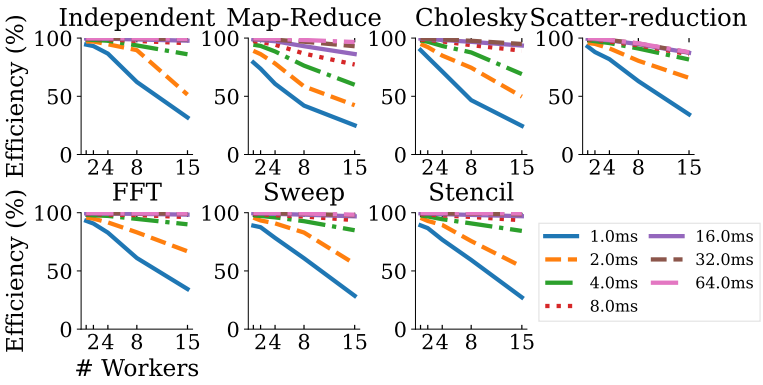

Fig. 6: **Scaling efficiency of CyRun for a range of task sizes (2-64ms). 1 Kernel/Task, GIL hold ratio = 0.**

task waits 4.2× less time in CyRun than in PyRun for the Map-Reduce DAG. This improved contention comes from (1) faster implementations of environment and dependency setup; and (2) removing runtime processing from the Python layer, decreasing the amount of time the interpreter is needed. GIL contention is a proxy measurement for all overheads at the Python layer. The Map-Reduce DAG is a worst-case scenario for end-to-end overhead. In this DAG, a task must wait for all tasks in the prior level to complete before launching. Delays to any task will be observed in the total graph execution time. Across 15 workers and 16ms tasks, we observe an average end-to-end overhead of 150μs per task. This is close to the overhead expected from GIL wait time alone (132μ per task). As internal runtime overheads in CyRun do not contribute to GIL wait time, the remaining dominant source of overhead is the necessary Python code to spawn tasks and configure task contexts.

Sensitivity to Task Workload Parameters. Figure 7 shows the sensitivity of Python mulithreading to the number of kernels per task, GIL Hold Ratio, and task dependency structure. We highlight a case where task size is fixed to 8ms and efficiency is measured across 8 worker threads. Even when the task does not explicitly hold the GIL for any length of time, simply splitting a kernel into 5 separate calls can drop performance by 11%. Frequent releases of the GIL leads to delays before a thread can notify that a task has completed. DAGs with many global synchronization points, like Map-Reduce, can be an exception. If tasks are launching in lock-step with each other, interweaving small kernels can help hide this bottleneck by cycling through them more quickly. When the kernels become too small to launch or process the next task, e.g. 10 Kernels/Task, these benefits begin to disappear. In modern Python, this suggests it is not sufficient to minimize pure Python code and chain together many short GIL-releasing NumPy calls within a task. Effective tasks must call into larger fused kernels that release the GIL for their entire duration. Likewise, it can be better to avoid thrashing by holding the GIL through a sequence of short kernels. When these conditions are met, greater than 90% efficiency can often be obtained.

A Potential Future [with PEP 703 `nogil`]. The proposal to make the GIL optional in CPython presents an opportunity for robust multithreaded task-based parallelism

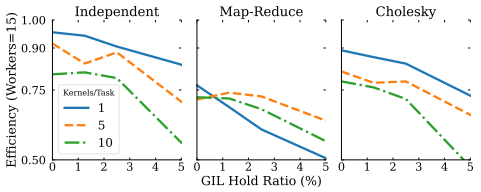

Fig. 7: **Effects of modifying GIL accesses patterns within 8ms tasks in CyRun. They hold the GIL for a percentage of execution, up to 400 μs at GIL Hold Ratio=5%. Time is split evenly between 1, 5, and 10 kernels in each task.**

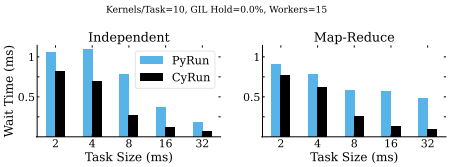

Fig. 8: **Per-task time spent waiting for acquiring the GIL after completing C-extension kernels.**

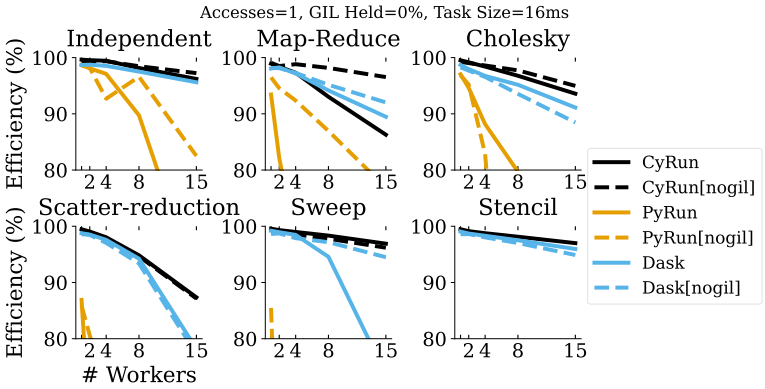

Fig. 9: **Scaling of 16ms tasks under CPython 3.9 with and without `nogil` (GIL Hold Ratio = 0%, Kernels/Task=1).**

that is not sensitive to workload design. Figure 9 and fig. 10 compare the performance of three runtimes: PyRun, CyRun, and Dask under PEP 703. Without the GIL, performance of contention dominated DAGs like Map-Reduce, Sweep, and Stencil is greatly improved. CyRun capitalizes on this and exhibits a substantial speedup of up 4x for the Map-Reduce workload. In contrast, Dask often shows less speedup. PEP 703 adds locks to many Python objects, like lists and dictionaries, in order to preserve their thread-safe behavior. Runtimes that use these objects extensively, like Dask, may actually experience degraded performance under this proposal. Similar effects are observed in PyRun especially for DAGs with long dependency lists, like Cholesky. Next, we shift our attention to multi-device tasking.

GPU Tasks. Parla GPU tasks require the GIL for a longer period of time to set up their GPU environment through the CuPy library. This leads to a higher overhead per task than their CPU counterparts. We study this with a Particle-in-Cell (PIC) [25] simulation, implemented in Parla, that uses 4 GPUs in a Map-Reduce pattern. Each task that evolves the particles runs on a separate GPU. These tasks internally launch 20 GPU kernels through Numba (totaling 48ms of work). At each fan-in barrier task, the kernels are synchronized to collect and broadcast particle statistics.

Unlike CPU tasks, GPU kernels can be enqueued asynchronously to device streams. The work launched in each task can be synchronized with the host machine at the end of it's own task body, which we call *self-synchronization*, or at a later point in time.

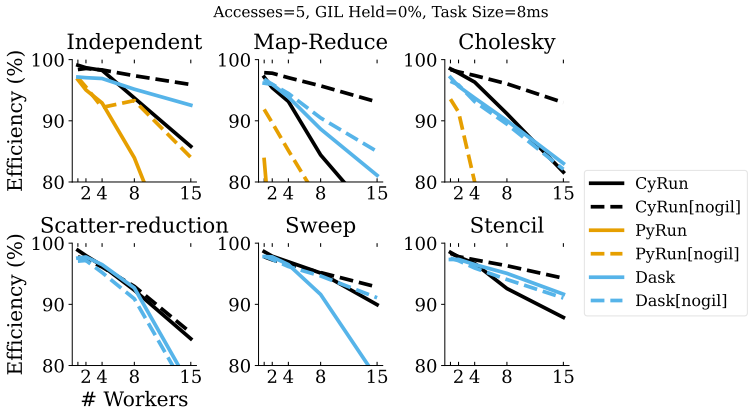

Fig. 10: **Scaling of 8ms tasks under CPython 3.9 with and without `nogil` (GIL Hold Ratio = 0%, Kernels/Task=5).**

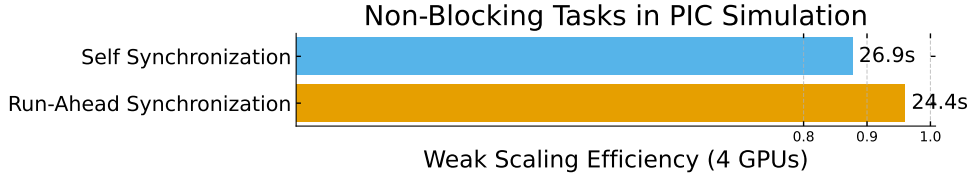

Fig. 11: **Asynchronous run-ahead execution between tasks helps hide overheads from GIL contention during GPU kernel launching. 500 timesteps of a Particle-in-Cell application, a map-reduce pattern with 20 asynchronous GPU Numba kernels per task. Total time per task is approximately 48ms.**

We refer to synchronizing the chain of tasks with the host as late as possible, e.g. when the result is needed on the CPU, as *run-ahead synchronization*. Precedence constraints between tasks are managed by cross-stream events. This allows task launching overhead, and costs of performing Python (GIL-holding) work in each task, to be better hidden by the workload. In Figure 11, we show how run-ahead synchronization can greatly improve the performance of Python GPU tasks.

5.3 Multi-Device Tasking

We validate our proposed multi-device runtime system by performing benchmarks against a runtime that supports similar tasks (Ray). As Ray is a multi-process library, users don't need to worry about the GIL, but tasks have higher overheads and must communicate between processes. Unlike CyRun, Ray does not provide data movement or variants for their multi-gpu tasks. We then evaluate the throughput of our proposed Multi-Device Queue launching mechanisms and the efficiency of CrossPy communication through microbenchmarks. Tying together these performance evaluations, we demonstrate orchestration of multi-GPU libraries with cuFFTmg, and benchmark a mini-app written for multi-device tasks using CrossPy.

Overheads of Multi-Device Tasks. Table 4 measures average per-task overhead as a function of the number of devices per task. These averages are collected from serial chains of 128 multi-GPU tasks with uniform size and the same number of devices per task. Measuring serial execution avoids measuring differences in scheduling policy and packing efficiency. CyRun has at least an 8× smaller overhead than Ray. However, in

Table 4: Per-task Multi-GPU Overhead. Time to initialize all GPU environments is shown in brackets.

GPUs per Task	1	2	4
CyRun Time (s)	1.036	1.043	1.052
Overhead (μs)	91 [55]	148 [88]	219 [140]
Ray Time (s)	1.249	1.252	1.254
Overhead (μs)	1759	1778	1797

Table 5: Multi-Device Runtime Comparison. Parallel efficiency in brackets.

System	DAG	2 GPUs per Task	4 GPUs per Task
Ray	Independent	8.58s [0.93]	17.14s [0.93]
CyRun	Independent	8.23s [0.97]	16.42s [0.97]
Ray	Map-Reduce	3.63s [0.87]	7.22s [0.88]
CyRun	Map-Reduce	3.40s [0.94]	6.55s [0.98]

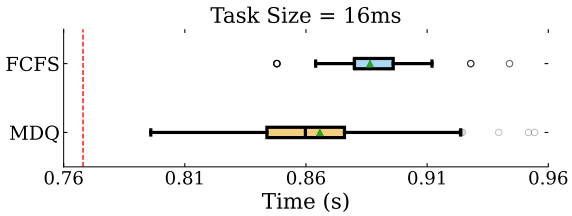

Fig. 12: Execution times for CyRun's multi-device queues (MDQ) vs. expected times of a strict FCFS schedule given their spawn order, for 100 random sequences of 128 16ms multi-device tasks, requiring different number of GPUs. The optimal wall-clock time for this load under optimal packing is 768ms and shown by the dashed line.

practice, Ray's overheads are often partially hidden by overlap in parallel execution. In contrast, overheads in CyRun may be serialized across threads due to the GIL. For CyRun, a large fraction of this overhead is the cost to serially set up the GPU environments for each device in the task ($\approx$30μs more per device). Comparisons with Ray on DAGs execution are shown in Table 5. All tasks in the DAGs are the same size (16ms) with a fixed number of devices. As the system has four GPUs, at 2-GPUs per task, two tasks are able to run in parallel. Due to lower overheads when launching each task, CyRun is able to exhibit better performance. Efficiency is calculated w.r.t the ideal time to process the workload without overheads on a system with 4 GPUs.

Microbenchmark of Multi-Device Queues The proposed MDQ algorithm aims for high-throughput even with sub-optimal priority in each device queue. We verify our system's performance by submitting 128 16ms tasks (64 1-GPU, 32 2-GPU, and 16 4-GPU tasks) in 100 random spawn orders on a four-GPU node. Each device queue uses insertion order as priority. Figure 12 compares the distribution of the observed execution times to the expected global FCFS time (without tasking overhead) for the same task orders. CyRun demonstrates performant scheduling, we next evaluate the proposed data communication interface and applications that use it.

Microbenchmark of CrossPy Communication. CrossPy provides a natural and efficient interface for programming multi-device tasks, but sometimes Pythonic interfaces can hinder performance. Table 6 demonstrates the benefit of using the cached *alltoall* to scatter data between arrays (Listing 6) over the traditional slicing assignment. Python assignment operators must fully resolve the right side before the left. For index assignment, this means it must first generate a temporary object for the right-hand-side and then assign it to the left-hand-side. To avoid this intermediate

Table 6: Speedup of Reusable *alltoall* vs. Python Assignment. Performance of first call with initialization shown in brackets.

#GPUs of src (rows)/dst (cols)	To 2 devices	To 4 devices
From 2 devices	25x [0.93x]	19.7x [0.91x]

allocation when reusing communication patterns, CrossPy can store the permutation between the source and target indices. This leads to a significant 20x speedup compared to the direct indexing. The initial setup for this requires additional work and allocation, resulting in a slight slow down on the first call (shown in brackets).

Multi-GPU Library Orchestration. The motivating use-case for multi-device tasks is the orchestration of multi-device library calls. To demonstrate that CyRun presents an effective solution, we submit seven independent multi-gpu FFT tasks of different sizes, 4 1-GPU, 2 2-GPU, and 1 4-GPU tasks, using the NVIDIA multi-GPU library cuFFTmg. Within each task, `clone_here` is used to move and evenly partition a new 20k-by-20k CrossPy array from the CPU onto the task's device set. cuFFTmg is used to process this partitioned data. Over all spawn orders, we observe an average speedup of 1.93× and a best case of 2.36× over a serial execution of the same workload. Serially, this workload takes 16.5 seconds, with 5.7 seconds of computation and 10.6 seconds of data communication.

Multi-Device Tasks using CrossPy. Multi-device applications can also be developed without external multi-device libraries. Using CrossPy, we implemented k-means inside of a multi-device task (Listing 5). Table 7 shows the weak scaling results of this implementation. As the input size grows, the computation dominates the running time relative to overheads from task scheduling and kernel dispatching. CrossPy does not compromise scalability for programming convenience.

Table 7: Weak Scaling Efficiency for *k*-means++

#Points per device ($\times 10^6$)	40	80	160
2-GPU Efficiency	.99	.91	.99
4-GPU Efficiency	.86	.88	.97

6 Related work

For an overview of Python-based runtimes for task parallelism, see Section 2. Multi-device tasking is common in multi-resource job scheduling in workflow and cluster management systems. In these systems, tasks are whole applications that may require different collections of nodes and distributed hardware resources [26–29]. Marble [30] and Teresis [31] provide multi-user job scheduling on systems with multi-GPU jobs. Teresis provides data prefetching onto GPUs. We focus on the finer-grained problem of orchestrating multi-device library calls within an application on a single node. Multi-process workflow scheduling systems, like Dask [32] and Parsl [33], support the static configuration of workers that map onto user-defined groups of specific devices. Ray [14] and PyCOMPSs [15] support tasks that specify different numbers of CPU cores and GPU devices. Unlike CyRun, all are process-based and NVIDIA GPU resources are

defined to each task through CUDA_VISIBLE_DEVICES. They support persistent GPU memory between tasks and the preloading of Python modules to reduce the overhead of initializing each worker's interpreter. Similar to CyRun, PyCOMPSs allows different implementations of a task to be defined for different hardware and environment resource constraints. However, they do not support GPU data prefetching, streams, or specialized function variants. Tensorflow [34] allows partitioning of work across user-defined device sets through a domain-specific interface for deep learning. Beyond Python, Chapel [35] provides scheduling for tasks with multiple GPUs per locale. HPX

CrossPy provides a shared indexing space across heterogeneous devices, which is akin to Global Array [36] in heterogeneous settings. CrossPy wraps lower-level NumPy, CuPy, etc. objects to provide a shared array abstraction on heterogeneous architectures. Several other Python systems do offer similar solutions but with some limitations. Ray's [14] shared datasets and Kokkos Remote Views [37] can only be distributed across homogeneous devices; Dask's [32] tasks require input arrays to be NumPy arrays thus limiting optimizations using manual placement. Legion [9] via its Python binding Pygion [38] or its Legate [39] NumPy extension, supports heterogeneous logically-shared arrays but are not interoperable with other runtimes or CuPy.

7 Conclusions

We discussed two challenges in developing task-parallel HPC apps in Python. First, a major bottleneck in performant parallel Python applications is the GIL. We evaluated complex interactions between the GIL and proposed different optimizations to mitigate these overheads. We provide a forward-looking solution that is able to scale especially effectively under the modern proposal for a "`nogil`" Python. With these optimizations, tasking in Python can be quite efficient. Second, we address the orchestration of multi-device libraries. Our solution in CyRun via multi-device tasks provides a set of unique features to program and effectively execute them. Our evaluation shows that CyRun can achieve parallel speedup across several applications and is competitive with more mature solutions in this space.

There are several opportunities for improvement. In the long run, we would like to adapt more sophisticated mapping and scheduling policies into our runtime, potentially ones that are aware of performance differences between function specializations over different device sets. For these policies, memoizing and learning from repeated DAGs may be critical to gather information and hide overheads.

References

[1] van der Walt, S., et al.: The NumPy array: A structure for efficient numerical computation. Computing in Science Engineering (2011)

[2] Virtanen, P., *et al.*: SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature methods **17**(3), 261–272 (2020)

[3] Okuta, R., et al.: CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations (2017)

[4] Anaconda: Numba (2018). https://numba.pydata.org/

[5] Al Awar, N., *et al.*: A Performance Portability Framework for Python. In: Proceedings of the ACM International Conference on Supercomputing (2021)

[6] Bradbury, J., et al.: Jax: Autograd and xla. Astrophysics Source Code Library (2021)
[7] Augonnet, Cédric and others: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience (2011)
[8] Bosilca, G., *et al.*: PaRSEC: Exploiting Heterogeneity to Enhance Scalability. Computing in Science Engineering **15**(6), 36–45 (2013) https://doi.org/10.1109/MCSE.2013.98 . Conference Name: Computing in Science Engineering
[9] Bauer, M., *et al.*: Legion: Expressing locality and independence with logical regions. In: SC '12: International Conference on High Performance Computing, Networking, Storage and Analysis (2012). https://doi.org/10.1109/SC.2012.71
[10] .NET Foundation: IronPython (2009). https://ironpython.net
[11] Gross, S.: PEP703: Making the Global Interpreter Lock Optional in CPython
[12] NVIDIA: cuBLAS (2023). https://developer.nvidia.com/cublas
[13] NVIDIA: cuFFT (2023). https://developer.nvidia.com/cuFFT
[14] Moritz, P., *et al.*: Ray: A Distributed Framework for Emerging AI Applications. In: Operating Systems Design and Implementation (2018)
[15] Tejedor, E., et al.: PyCOMPSs: Parallel Computational Workflows in Python. International Journal of High Performance Computing Applications (2017)
[16] Lee, H., *et al.*: Parla: A Python Orchestration System for Heterogeneous Architectures. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (2022)
[17] Slaughter, E., *et al.*: Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (2020)
[18] DeVito, Z.: Torchscript: Optimized execution of pytorch programs. Retrieved January (2022)
[19] Snow, E.: PEP 703 – A Per-Interpreter GIL (2022)
[20] Gonthier, M., *et al.*: Memory-aware scheduling of tasks sharing data on multiple gpus with dynamic runtime systems. In: IPDPS (2022). https://doi.org/10.1109/IPDPS53621.2022.00073
[21] StarPU Handbook - Language Bindings (2023)
[22] Beazley, D.: Understanding the Python GIL. In: PyCON Python Conference. Atlanta, Georgia (2010)
[23] Rossum, N.C.: PEP 343: The "with" Statement (2005)
[24] Texas Advanced Computing Center (TACC), The University of Texas at Austin (2018). https://www.tacc.utexas.edu/
[25] Almgren-Bell, J., Awar, N.A., Geethakrishnan, D., Grigoric, M., Biros, G.: A Multi-GPU Python solver for low-temperature non-equilibrium plasmas. In: IEEE 34th International Symposium on Computer Architecture and High Performance Computing SBAC-PAD 2022, p. 11 (2022). https://doi.org/10.1109/SBAC-PAD55451.2022.00025
[26] Fan, Y., *et al.*: Scheduling beyond CPUs for HPC. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (2019)

[27] Gog, I., Schwarzkopf, M., Gleave, A., Watson, R.N., Hand, S.: Firmament: Fast, centralized cluster scheduling at scale. In: OSDI 16 (2016)
[28] Reuther, A., *et al.*: Scalable system scheduling for hpc and big data. Journal of Parallel and Distributed Computing **111**, 76–92 (2018)
[29] Schwiegeishohn, U., Yahyapour, R.: Improving first-come-first-serve job scheduling by gang scheduling. In: Workshop on Job Scheduling Strategies for Parallel Processing (1998)
[30] Han, J., *et al.*: Marble: A multi-gpu aware job scheduler for deep learning on hpc systems. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pp. 272–281 (2020)
[31] Chen, C., *et al.*: Tereis: A package-based scheduling in deep learning systems. In: 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS), pp. 867–874 (2023). IEEE
[32] Dask Development Team: Dask: Library for Dynamic Task Scheduling. (2016)
[33] Babuji, Y., *et al.*: Parsl: Pervasive Parallel Programming in Python. HPDC '19 (2019). https://doi.org/10.1145/3307681.3325400
[34] Abadi, M., et al.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems (2015)
[35] Chamberlain, B.L.: Chapel (Cray Inc. HPCS Language), (2011)
[36] Nieplocha, J., *et al.*: Advances, applications and performance of the global arrays shared memory programming toolkit. The International Journal of High Performance Computing Applications **20**(2), 203–231 (2006)
[37] Mishler, D., *et al.*: Performance Insights into Device-initiated RMA Using Kokkos Remote Spaces. In: 2023 IEEE International Conference on Cluster Computing Workshops (2023)
[38] Slaughter, E., *et al.*: Pygion: Flexible, Scalable Task-based Parallelism with Python. In: Parallel Applications Workshop, Alternatives To MPI (2019)
[39] Bauer, M., Garland, M.: Legate NumPy: Accelerated and distributed array computing. In: SC19: International Conference for High Performance Computing, Networking, Storage and Analysis (2019)