cuNumeric

Wonchan Lee | Legion Retreat, 12/7/2022

cuNumeric

- Distributed and GPU-Accelerated NumPy-like array programming library
- Aims at both productivity and scalability

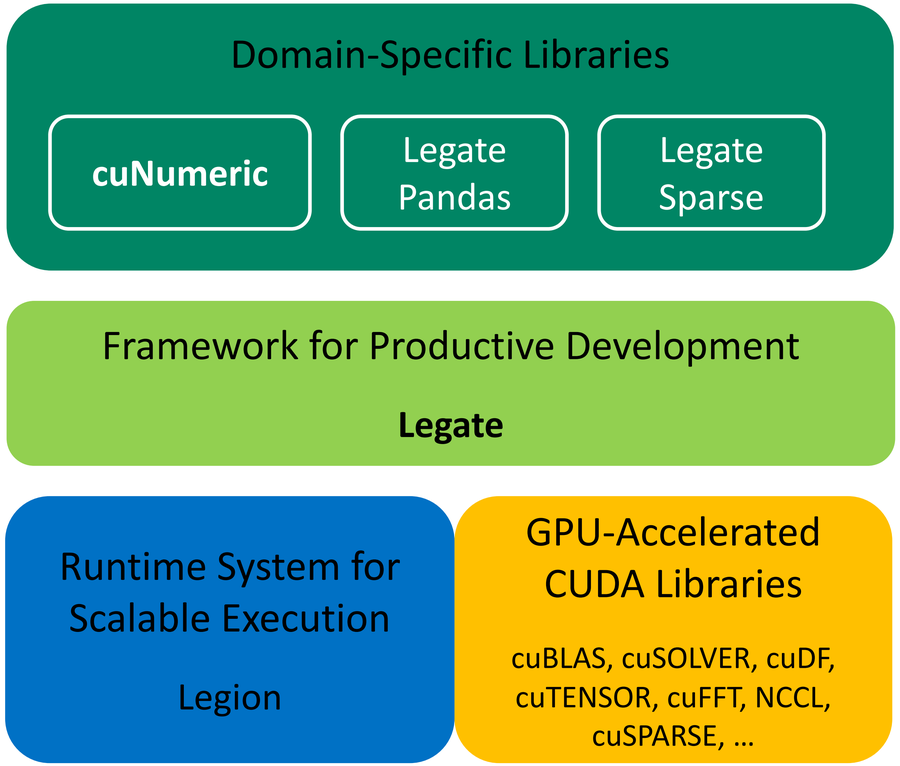
Write this

```
def cg_solve(A, b, tolerance):
    x = np.zeros_like(b)
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    max_iters = b.shape[0]

    for i in range(max_iters):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)

        if np.sqrt(rsnew) < tolerance:
          break

        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
```

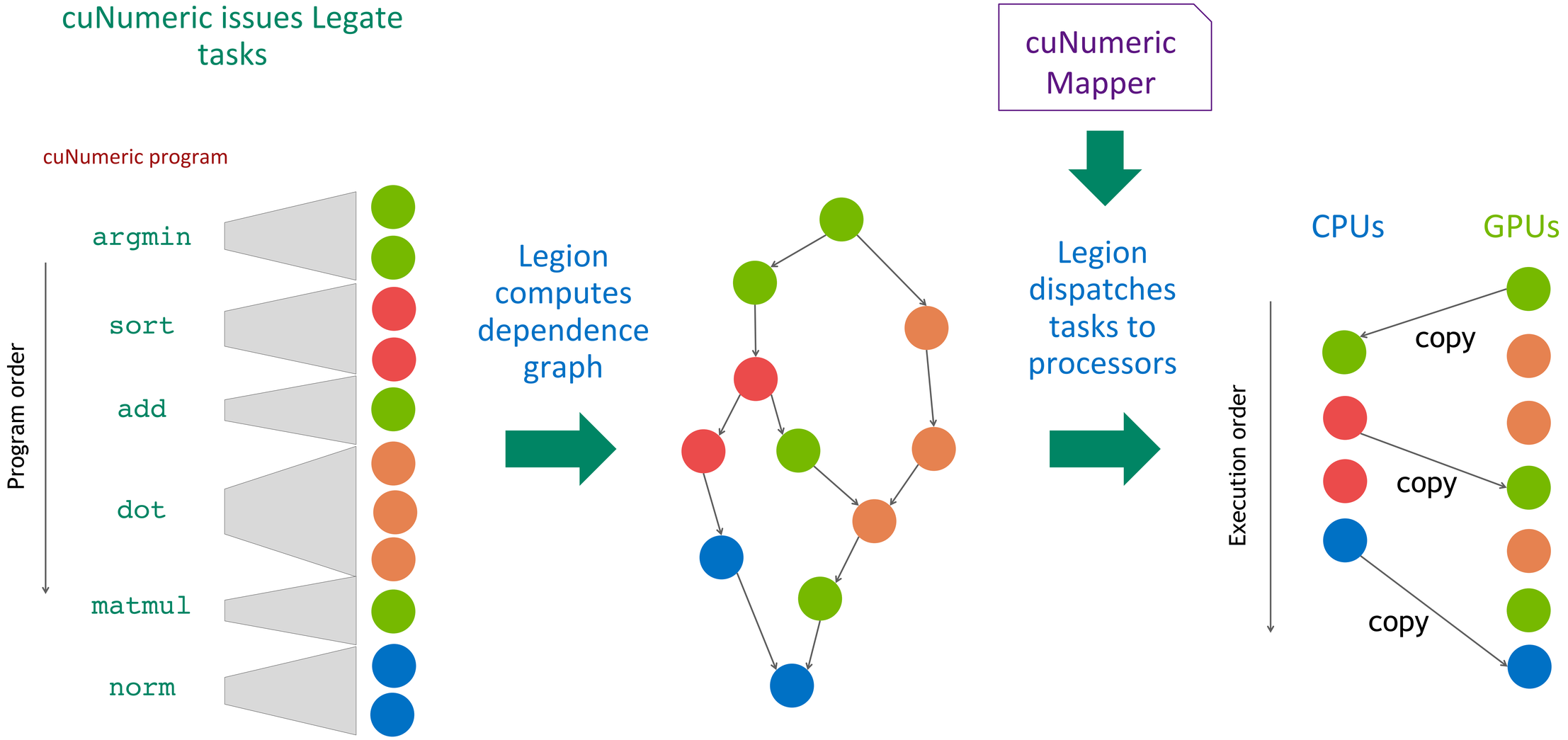
Run here

Strategy

- “White-box” approach: a from-scratch implementation of NumPy API
  - More labor-intensive than black-box approaches reusing existing NumPy implementations
  - But brings greater performance and scalability
- Build a framework that simplifies Legion programming
  - And facilitates composition and cross-library optimizations
- Reuse domain kernels in existing CUDA libraries



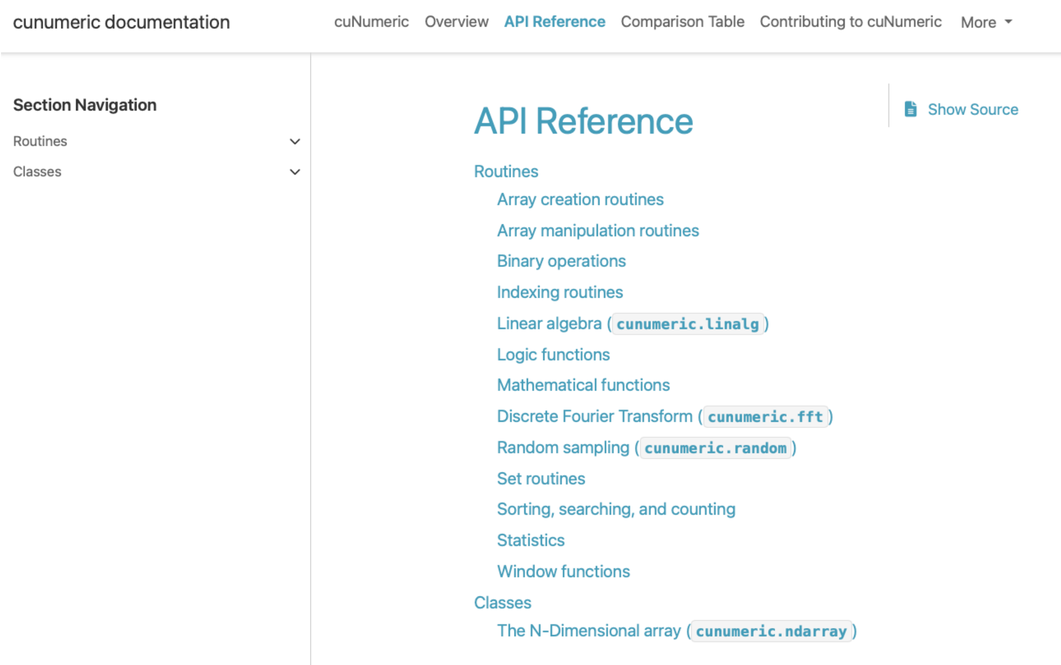
cuNumeric Architecture



What’s New

Productization

- Got a new name!
- Full-on production mode
  - Engineering for code quality
  - Rigorous development process with a team of full-time engineers
  - Full QA support with correctness & performance tests and code coverage
  - Documentations



What’s New

Improved API coverage

- Covers 60% NumPy API coverage
  - Advanced indexing
  - Tensor contraction
  - Multi-dimensional sorting
  - 96% of ufuncs
  - 80% of RNGs

Comparison Table

Here is a list of NumPy APIs and corresponding cuNumeric implementations.

A dot in the cunumeric column denotes that cuNumeric implementation is not provided yet. We welcome contributions for these functions.

NumPy vs cuNumeric APIs

Module-Level

NumPy	cunumeric	single-GPU/CPU	multi-GPU/CPU
`numpy.all`	`cunumeric.all`	✓	✓
`numpy.allclose`	`cunumeric.allclose`	✓	✓
`numpy.amax`	`cunumeric.amax`	✓	✓
`numpy.amin`	`cunumeric.amin`	✓	✓
`numpy.angle`	•		
`numpy.any`	`cunumeric.any`	✓	✓
`numpy.append`	`cunumeric.append`	✓	✓
`numpy.apply_along_axis`	•		
`numpy.apply_over_axes`	•		

Linear Algebra

NumPy	cunumeric	single-GPU/CPU	multi-GPU/CPU
`numpy.linalg.cholesky`	`cunumeric.linalg.cholesky`	✓	✓
`numpy.linalg.cond`	•		
`numpy.linalg.det`	•		
`numpy.linalg.eig`	•		
`numpy.linalg.eigh`	•		
`numpy.linalg.eigvals`	•		
`numpy.linalg.eigvalsh`	•		
`numpy.linalg.inv`	•		
`numpy.linalg.lstsq`	•		
`numpy.linalg.matrix_power`	`cunumeric.linalg.matrix_power`	✓	✓
`numpy.linalg.matrix_rank`	•		
`numpy.linalg.multi_dot`	`cunumeric.linalg.multi_dot`	✓	✓
`numpy.linalg.norm`	`cunumeric.linalg.norm`	✓	✓
`numpy.linalg.pinv`	•		
`numpy.linalg.qr`	•		
`numpy.linalg.slogdet`	•		
`numpy.linalg.solve`	`cunumeric.linalg.solve`	✓	✗

What's New

Ergonomics

- Improved NumPy interop/fallback
  - Seamless interoperation via array interface

```
a = numpy.zeros(10)
b = cunumeric.ones(10)
numpy.add(a, b) # dispatches to cuNumeric
```

  - Partially implemented operations gracefully fall back to NumPy
- API coverage tool & zero code-change patching

To use this tool, invoke it as shown below, with the name of the program to patch:

```
lgpatch <program> -patch numpy
```

For example, here is a small `test.py` program that imports and uses various `numpy` funtions:

```
# test.py

import numpy as np
input = np.eye(10, dtype=np.float32)
np.linalg.cholesky(input)
```

You can invoke `lgpatch` to run `test.py` using `cunumeric` functions instead, without any changes to the original source code. Any standard `cunumeric` runtime options (e.g. for coverage reporting) may also be added:

```
$ lgpatch test.py -patch numpy -cunumeric:report:coverage
cuNumeric API coverage: 4/4 (100.0%)
```

What’s New

Ergonomics

Conda packages

legate / packages / cunumeric 22.10.00

0

Drop-in Replacment for NumPy

Conda | Files | Labels | Badges

License: Apache-2.0
Home: https://github.com/nv-legate/cunumeric
Development: https://github.com/nv-legate/cunumeric
Documentation: https://github.com/nv-legate/cunumeric
661 total downloads
Last upload: 1 month and 17 days ago

Installers

Edit

Info: This package contains files in non-standard labels.

linux-64 v22.10.00

conda install

To install this package run one of the following:

```
conda install -c legate cunumeric
conda install -c "legate/label/archive" cunumeric
```

Jupyter support

jupyter Untitled1 Last Checkpoint: a few seconds ago (unsaved changes) Logout

File Edit View Insert Cell Kernel Help Trusted legate_gpu

Run Code

In [1]:
```
%load_ext legate.jupyter
%legate_info
```

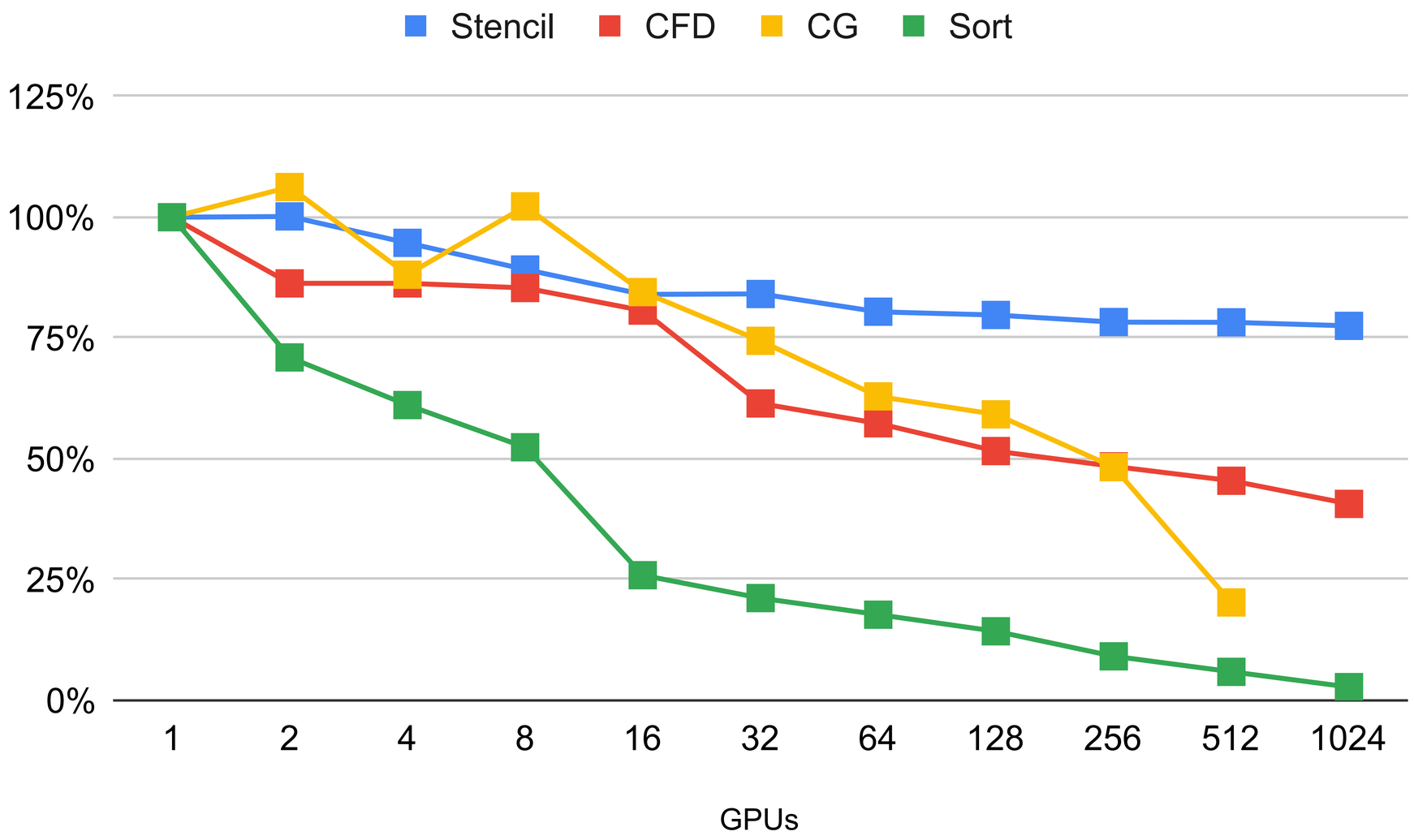
```
Kernel 'legate_gpu' configured for 1 node(s)

Cores:
  CPUs to use per rank : 4
  GPUs to use per rank : 1
  OpenMP groups to use per rank : 0
  Threads per OpenMP group : 4
  Utility processors per rank : 2

Memory:
  DRAM memory per rank (in MBs) : 4000
  DRAM memory per NUMA domain per rank (in MBs) : 0
  Framebuffer memory per GPU (in MBs) : 4000
  Zero-copy memory per rank (in MBs) : 32
  Registered CPU-side pinned memory per rank (in MBs) : 0
```

In [2]:
```
from legate.timing import time
import cunumeric as np

def cholesky(n, dtype):
    input = np.eye(n, dtype=dtype)
    print(type(input))

    start = time()
    np.linalg.cholesky(input)
    stop = time()
    flops = (n**3) / 3 + 2 * n / 3
    total = (stop - start) / 1000.0
    print(f"Elapsed Time: {total} ms")
    print(f"{flops / total} GOP/s")

cholesky(20, "float32")
```

```
<class 'cunumeric.array.ndarray'>
Elapsed Time: 0.568 ms
4718.30985915493 GOP/s
```

Performance

Weak scaling efficiency for different styles of workload



Operational differences from NumPy

- Deferred exceptions

```
a = np.zeros(10)
b = np.array([10])
try:
  a[b] = 1
except IndexError:
  pass # May not hit this in release mode
```

- Non-determinism due to parallel updates

```
a = np.zeros(10)
b = np.array([0, 0])
a[b] = [1, 2]
assert a[0] == 2 # May fail if the first
                   writer won
```

- Some operators that return views in NumPy can make copies

```
a = np.random.zeros(5, 5)
b = b.reshape(25)
b[0] = 1
assert a[0, 0] == 1 # Fail in cuNumeric
```

Plan

Grinding towards a beta-release

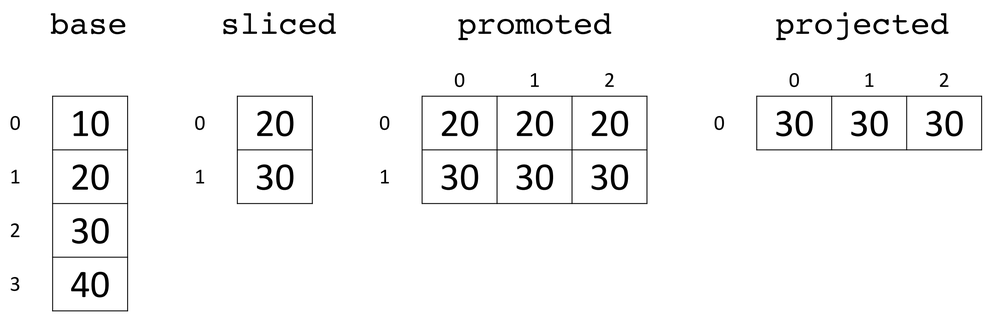
- More API coverage
  - FFT and linear algebra
  - IO functions
- Performance improvements
  - Better mapping heuristics
  - Reduce overhead by porting Legate to C++
- Usability
  - Automatic machine configuration
  - Compatibility with the canonical Python interpreter
  - Multi-node capable conda packages

cuNumeric Internals
(A Brief Overview of Legate)

Data Abstraction

- cuNumeric maps ndarrays to *Legate Stores*
- Legate Store is a multi-dimensional array of fixed-type elements
  - Backed by a region field or a future if it's singleton (i.e., of shapes (), (1,), (1,1),(1,1,1), …)
  - Views to a store can be created using *store transformations*

```
sliced = base.slice(tile_shape=(2,),offsets=(1,))
promoted = sliced.promote(1, 3)
projected = store.project(0, 1)
```



  - Views are mapped to subregions and/or accessor transformations
- Legate manages free region fields from GC'd Legate stores for reuse
  - Stores can reuse the same region field if they have the same shape and element size

Tasks

- Each cuNumeric operator uses one or more Legate tasks to implement the algorithm

- Legate task is a thin wrapper around Legion task
  - Legate tasks take stores and by-value scalars
  - Each store can be associated with one or more access modes: read, write, and reduction
  - Stores with multiple access modes are mapped to read-write region requirements

- Each Legate task is mapped to **one or more Legion tasks**
  - Legate automatically parallelizes tasks by partitioning their store arguments
  - Partitioning constraints attached to a task control how stores are partitioned

- Two useful optimizations in Legate-to-Legion conversion
  - Region requirements for the same region and privilege are coalesced into one requirement
  - When a store in reduction mode is partitioned disjointly, the privilege is promoted to read-write

Legate in Action

Binary Operator Example

Task launch (Python)

```
lhs = self.base
rhs1 = src1._broadcast(lhs.shape)
rhs2 = src2._broadcast(lhs.shape)

task = self.context.create_auto_task(TaskID.BINARY_OP)
task.add_output(lhs)
task.add_input(rhs1)
task.add_input(rhs2)
task.add_scalar_arg(op_code, ty.int32)

task.add_alignment(lhs, rhs1)
task.add_alignment(lhs, rhs2)

task.execute()
```

Match shapes of stores using projections and promotions

Populate Legate task with stores and scalars

Request inputs and output to be partitioned in the same way

Task (C++)

```
auto rect = lhs.shape<DIM>();

auto lhs_acc = lhs.write_accessor<LHS, DIM>(rect);
auto rhs1_acc = rhs1.read_accessor<RHS1, DIM>(rect);
auto rhs2_acc = rhs2.read_accessor<RHS2, DIM>(rect);

BinaryOp<OP_CODE, TYPE_CODE> func{};
for (PointInRectIterator<DIM> p(rect); p(); p++)
  lhs_acc[*p] = func(rhs1_acc[*p], rhs2_acc[*p]);
```

Covers **all possible combinations of inputs and outputs**
(region x region, region x future, future x future, inplace operation)

Unbound Stores

- Stores whose shapes are determined by producer tasks
  - Mapped to output requirements in Legion

```
outputs = tuple(
    self.context.create_store(np.int64,shape=None,ndim=1)
    for _ in range(self.ndim)
)
task = self.context.create_auto_task(TaskID.NONZERO)
task.add_input(self.store)
for output in outputs:
    task.add_output(output)
...
```

self.store

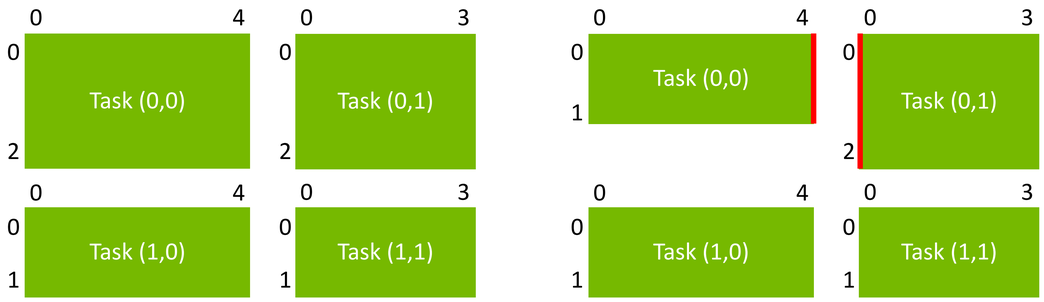
	0	1	2
0	0	20	20
1	30	0	0

outputs

0	0
1	0
2	1

0	1
1	2
2	0

- Unbound stores can be multi-dimensional
  - Sub-stores must be aligned with their neighbors



aligned

misaligned

Mapping Interface

- Legate comes with the base policy engine with default heuristics for
  - Task slicing and sharding
  - Task and store mappings
  - Colocation of stores and their slices in physical instances

- Mapping policies can be overridden via a high-level mapping interface

```
struct InstanceMappingPolicy {
  StoreTarget target{StoreTarget::SYSMEM};
  AllocPolicy allocation{AllocPolicy::MAY_ALLOC};
  InstLayout layout{InstLayout::SOA};
  DimOrdering ordering{};
  bool exact{false};
}

struct StoreMapping {
  std::vector<Store> stores{};
  InstanceMappingPolicy policy;
}

struct LegateMapper {
  virtual TaskTarget task_target(Task& task, std::vector<TaskTarget>& options) = 0;
  virtual std::vector<StoreMapping> store_mappings(Task& task, std::vector<StoreTarget>& options) = 0;
};
```

Resource Scoping (Work in Progress)

- Allows users to assign different subsets of resources to difference sections of code
  - Legate automatically adjusts sharding/mapping/etc. for each resource scope
  - Useful when the program has independent workloads

```
gpus = legate.core.get_machine().only("gpu")
num_gpus = gpus.count("gpu")

subset1 = gpus[              :num_gpus   //3]
subset2 = gpus[num_gpus   //3:num_gpus*2//3]
subset3 = gpus[num_gpus*2//3:]

...

for _ in range(ITERS):
    with subset1:
        cunumeric.matmul(arr1, arr2)
    with subset2:
        cunumeric.einsum("mlk,lkn->mn", arr3, arr4)
    with subset3:
        cunumeric.sort(arr5)
```

Summary

- Legate is packed with many productivity features for Legion library development
  - Features uncovered in this talk: Legate data interface, profiling support, testing framework, manual parallelization primitives, etc.
  - More features to come soon!

- **Please consider Legate for your next Legion development!**
  - Find more about Legate at https://github.com/nv-legate
  - Reach out to us on legate@nvidia.com for any questions or comments
  - Or start hacking today!

Questions?

Cross-Language Exceptions

- Some exceptional cases can only be caught in C++ tasks
  - E.g., out-of-bound accesses
- Legate provides a mechanism to capture C++ exceptions and re-throw them in Python

Task launch (Python)

```
task = self.context.create_auto_task(TaskID.ZIP)
task.throws_exception(IndexError)
```

Task (C++)

```
auto new_index = index < 0 ? index + extent : index;
if (new_index < 0 || new_index >= extent)
  throw legate::TaskException("index is out of bounds in index array");
```

  - Deduplicates exceptions from multiple point tasks
- Exceptions from C++ tasks are deferred
  - By default, Legate checks and re-raises pending exceptions only occasionally
    ➔ A client Python program cannot and should not rely on those exceptions in the control flow
  - Checks can optionally be made eager

Communicators

- Algorithms requiring collective communication are often more efficient to implement using explicit communication APIs
- Legate supports per-task communicator uses
  - With concurrent progress guarantee from Legion's concurrent index launch
  - Task launchers must declare communicator types at launch time (leaky abstraction)
  - Legate caches communicators based on participant counts

Task launch (Python)

```
task = context.create_task(TaskID.SORT)
...
if runtime.has_gpus:
    task.add_nccl_communicator()
else:
    task.add_cpu_communicator()
```

Task (C++)

```
void SortTask::gpu_variant(TaskContext& context) {
  ...
  auto comm = context.communicators()[0].get<ncclComm_t*>();
  ...
}
```