Accelerating NumPy for Data Science: A Summary on Legate

Cheng Peng[*]

January 6, 2022

Contents

[*]2020533068 pengcheng2@shanghaitech.edu.cn

1 Abstract

Python is a widely used general programming language that is known for the simple and clear syntax. With the support of huge standard library and third-part packages, a enormous Python eco-system has formed in the past two decade.

It is popular in the filed of scientific computation and data science, however Python is not designed for high performance computing. The object memory model and the interpreter design make it hard for Python programs to speed-up and/or scale-up. To overcome this barriers, plenty of research projects have arised. Some tries to address this issues by providing alternative runtime for Python, while others re-implement certain Python libraries with high performance code and link the external code with FFI.

In this paper review project, we will dive into Legate NumPy[1][1], a Python library aims at accelerating Python with distributed GPU cluster.

2 Background

Started as the successor of Numarray and Numeric, NumPy[2] quickly gain its popularity. The primary data structure in NumPy is a N-dimensional array (also called tensor) `numpy.ndarray`. It empowers the users to express bunch of mathematical expression with simple and clear syntax.
16 years after invention, a scientific research eco-system has formed on top of NumPy. The picture 1 provides

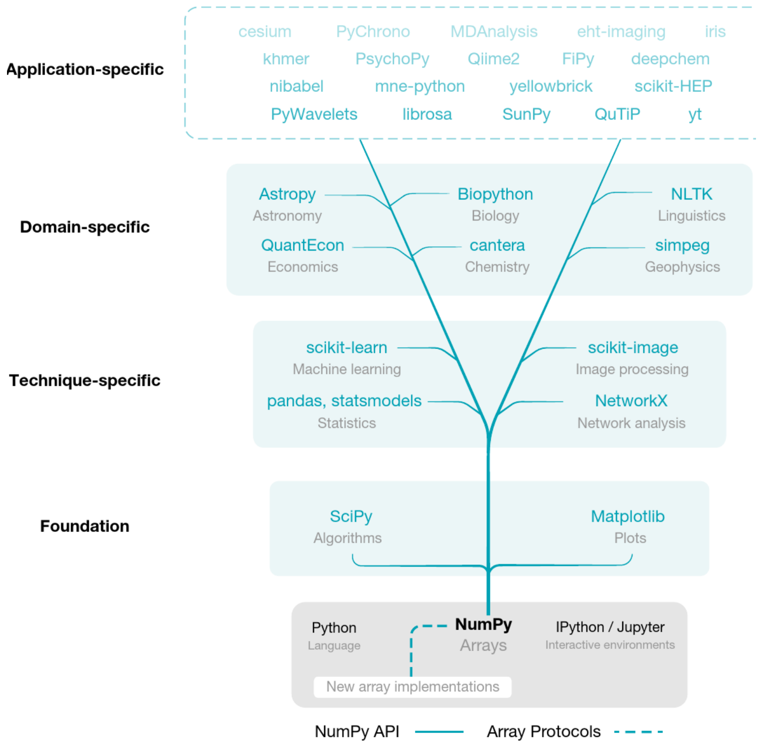


Figure 1: The scientific Python eco-system

a glance at the huge system.

The canonical implementation of NumPy stores the array in main memory and do computation on CPUs. As the problem size in research frontier continuse to grow, people find it hard to fit the data into single machine main memory and the power of CPUs are become insufficient. In deep-learning area, people are training networks with billions of paramters[3]. In seek for gravitational waves, Terabytes of data per data have to be examined[4]. To accomplish these challenging tasks, NumPy have to leverage modern specialized hardwares and explore parallism.

[1]published on SC '19

3 The Legate NumPy System

3.1 overview

Proposed and implemeneted by NVIDIA, Legate[1] is a programming system that strives to accelerate NumPy programs with the power of GPUs and enable distributed computing. Normal NumPy codes are translated into the Legion[5] programming model, which enhances the scalability and performance of programs. By offering interfaces mimics the NumPy array programming APIs and providing the equivalent semantics, Legate can achieve this while staying transparently to programmers.

The benchmarks on machines with up to 1280 CPU cores and 256 GPUs show that Legate achieves weak-scalability with the state-of-the-art efficiency. Legate is capable for both constructing prototypes and scaling up pre-exists programs significantly.

We will demonstrate the Legion programming model and how Python programs are translated into Legion.

3.2 the Legion programming model

Designed as a programming model suitable for large scale high performance applications, Legion[5] attach great importance on the locality and independence of data and tasks. Detailed summary on the data model and the task model, as well as the scheduling strategies are to be shown in this section.

3.2.1 Legion data model: the logical region

Reponsible for organizing all the long-lived data, logical regions is the a concept in Legion system. It is a 2D tabular where the rows are named by multi-dimensional coordinates and the columns are indexed by named fileds. This flexible data model gives Legion the power to manipulate N-dimensional arrays and crate arbitary views readily. The figure 2 gives a brief example of the logical region.

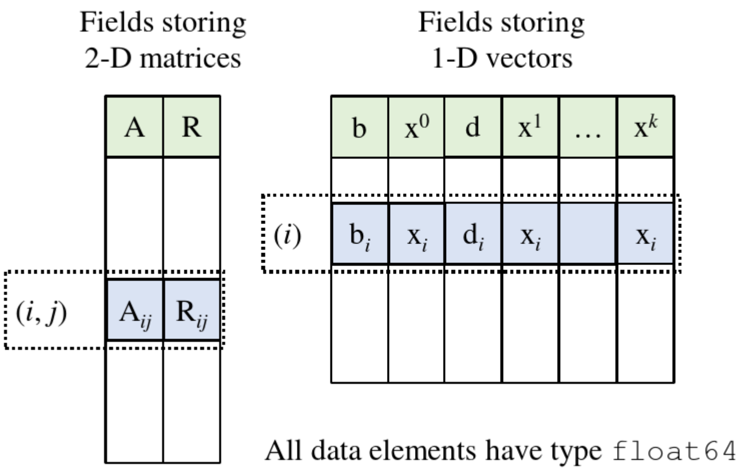
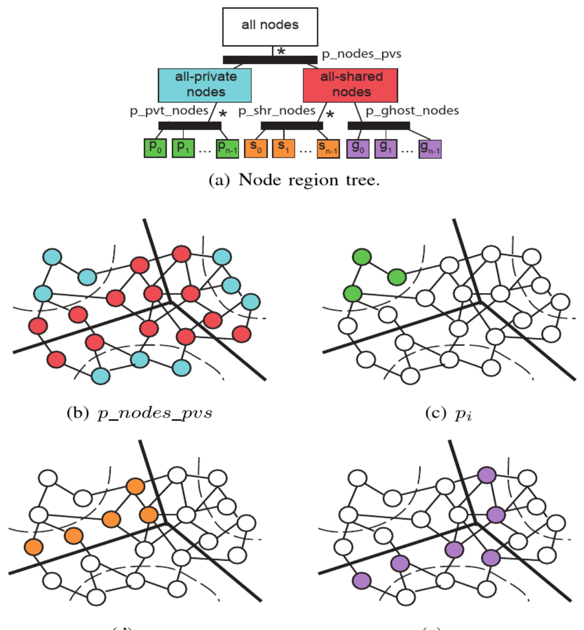

Fields storing 2-D matrices
Fields storing 1-D vectors
A
R
b
x^0
d
x^1
…
x^k
(i)
b_i
x_i
d_i
x_i
x_i
(i, j)
A_{ij}
R_{ij}
All data elements have type float64

Figure 2: A example illustrating the logical region data model[1]

Legion requires programers to declare the data locality and dependencies using logical regions. Moreover, every task, which will be covered later, has to specify the logical regions that are accessed, the access privilages (e.g., read-only, read-write or reduce) as well as the coherence (e.g., exclusive access and atomic access).
Privileges provides the data dependence information of task, whichis used to extract potentially parallelism For example, if two tasks access the same region with read-only privilages the two tasks can potentially be run in parallel. The coherence configuration specifies the semantics of concurrent accesses. For example, if the program executes $f_1(r); f_2(r)$ where both tasks declare exclusive access to r. Then Legion has to guarantee that the output of execution will be as if the program runs sequentially. On the other hand, if the tasks access r with atomic coherence, then Legion guarantees that $f_1(r)$ runs entirely before $f_2(r)$ or $f_2(r)$ runs before $f_1(r)$.
Although data access and coherence information are required to be given, programmers never specify the physical data layout, the runtime system handles data partition and distribution automatically, staying transparent to the logical data model of a programs. How the data are actually partitioned and distributed are not determined solely by the logical region structure, the hardware conditions and runtime behaviors are all taken into account. Both natural decomposition and hierachical decomposition are applied to reduced the data transfer traffic and the increase locality of memory access pattern 3



(a) Node region tree.

(b) p_nodes_pvs (c) p_i

(d) s_i (e) g_i

Figure 3: A example of hierachical and natural decomposition[5]

3.2.2 Legion execution model: the task tree and deferred execution

Next, we will take a look at the execution model of Legion.

The computation unit of a Legion program is called task. A unique top-level task marks the start of the whole program. Every task is allowed to create arbitary number of subtasks, thus forming a rooted task tree. In previous section, we have mentioned that Legion requires data access permissions. The launched subtasks only have access to regions that can be accessed by its parent. Moreover, the privilages on a region should be compatible with its parent's.

A asynchronous execution model is employed in Legion programming language. A task launch call returns immediately to the caller. The return values of tasks are wrapped in future (also called promise in some areas). The caller can examine whether the computation of the previously launched task has finished through the future object or wait explicitly until the future is resolved. A noticable feature of future object is that it can be passed to other tasks, enabling easy compositon of tasks.
Based on the knowledge of which region a task will access, dynamic dependencies analysis at runtime determines the relationship between tasks. Tasks that are found to be independent can be re-ordered or executed in parallel while dependent data and tasks are handle with care to preserve the semantics of sequential execution order.

3.3 logical regions for NumPy arrays

To translate NumPy `numpy.ndarray` into logical region, the most obvious way is to create a logical region for every array object in the Python program. However, this is not the way Legate do. Legate to pack multiple arrays of the same shape into a single region, where each column corresponds to an array. In the example of logical region 2, we can see that the arrays $b, x^0, x^1 \dots x^k$ are organized in one region.
The motivation for packing several arrays in one region is to reduce the cost of dynamic partitioning. Suppose that we have two arrays of the same shape x, y, when perform element-wise operations e.g, $x + y$ adding two vectors or $x \cdot y$ computing dot-product, we would benefit from partitioning x, y in exactly the same way. Thus, when performing operation $x + y$, the cost of dynamic partitioning is avoided.

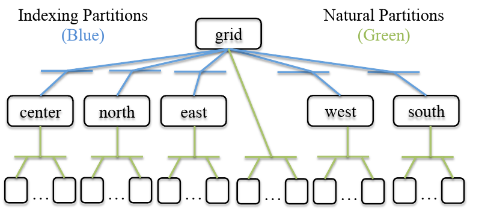
To create array slices for implementing the advanced indexing operations in NumPy, Legate calls Legion scatter and/or gather to copy from the original array. This reflect the semantics of array slicing: create a copy rather than an alias.

In contrast, for basic indexing, which create views/reference to the original array instead of a copy, Legate simply create a sub-region from the logical of the orignal array. In Legion Sub-region are aliased to the ancestor regions (including the direct parent region). Here we give an easy-to-read example of basic indexing ??.

```
# Given grid, an n×n array with n > 2,
# create multiple offset stencil views
center = grid[1:-1, 1:-1]
north  = grid[0:-2, 1:-1]
east   = grid[1:-1, 2:  ]
west   = grid[1:-1, 0:-2]
south  = grid[2:  , 1:-1]

for i in range(iters):
    total = center+north+east+west+south
    center[:] = 0.2*total
```

(a) 2D-stencil using basic indexing



(b) the region tree created by the stencil program

Figure 4: illustration of basic-indexing in Legate

3.4 legion tasks for NumPy APIs

Legate provides NumPy compatible APIs and implement them using Legion task, where each NumPy operation is decomposed into a few asynchronous tasks. When running the program, the tasks that implement N-dimensional array computations are created and submitted to the Legion runtime.
Recall: Legion runtime, guided by a mapper, will handle region partitioning and task schedule. Therefore, Legate can focus purely on domain-specific partitioning and mapping of each task. semantics being transparent to the low-level details of distributed execution.

Legate have registerd three different implementation variants for every NumPy operation, one sequential version for a single CPU core, one OpenMP version for a multi-core CPU, and CUDA version for NVIDIA GPU.
To avoid code duplication and enable extensibility, Legate implement all the three task variants in one C++ template function. The generic template function can be instantiated using different executors. Executors, defined in the Agency library, are objects that specify the how the code can be executed. Agency library provides executor for CPUs, OpenMP and CUDA.
This design allows programmer to extend Legate for utilizing alternative accelerators, all you need to do is simply to create a executor for that hardware accelerator.

3.5 Legate mapper: domain-specified mapping and control replication

Legate translate Python NumPy operations into Legion programming model. Legion can handle the mapping and scheduling automatically, however it has no domain-specific heuristic. To achieve maximized performance, Legate implement a mapper to leverage its domain-specific knowledge of NumPy.

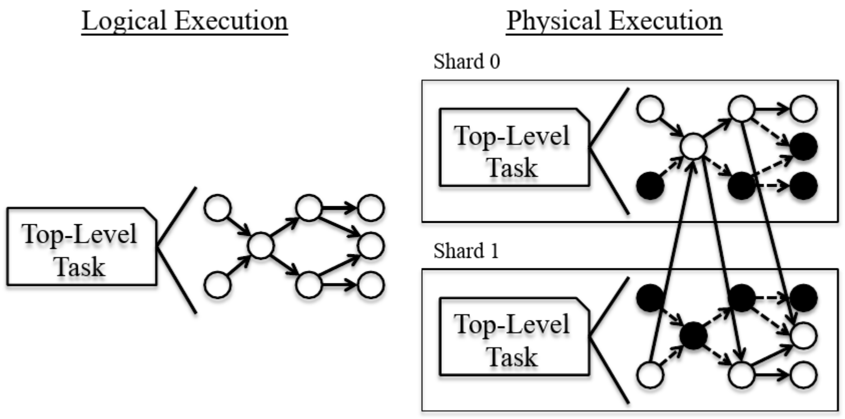


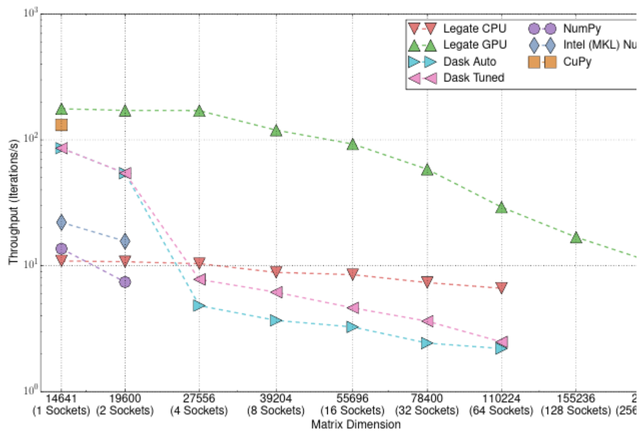
Figure 5: control replication

Recall that Legion programs start as a unique root task, the machine that initally runs this task can become a sequential bottleneck since it has to distribute large amount of sub-tasks to other nodes. This is quite common in systems that have a centralized controller which limit the scalability.
Control replication is a feature in Legion runtime system designed address this problem. Multiple copies of a task are spawned on different machines, they work collectively to reduce the time waiting for dragger.

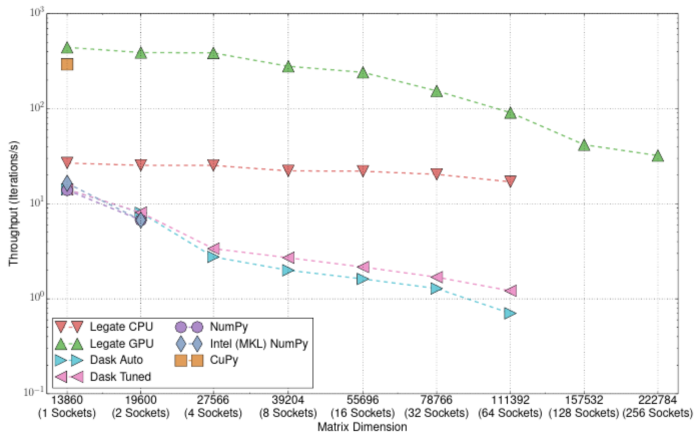
The figure 5 gives an example on how control replication helps to scale-up a program.

3.6 benchmarks

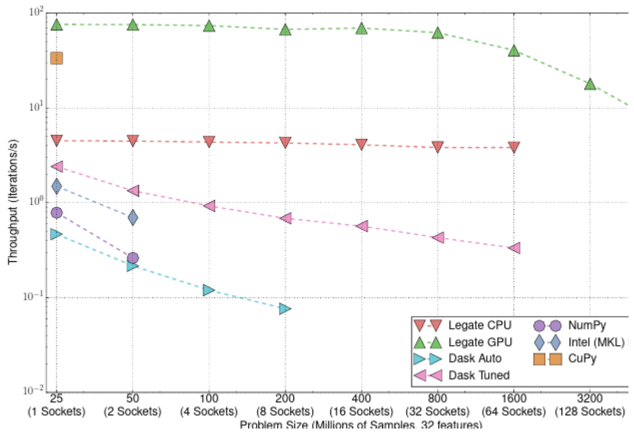
To make a empirical analysis on the scalability of Legate, they made a few representative benchmarks. The benchmarks are done on a cluster constists of 32 NVIDIA DGX-1V node. Each node is equipped with 8 Tesla V100 GPUs connected by a hybrid mesh cube topology using NVLink and PCI-E. The CPUs on the DGX-1V ndoe are two 20-core Intel Xeon E5-2698 CPUs with hyperthreading enabled and 256 GB of DDR4 memory. The Nodes in the cluster are connected by an Infiniband EDR switch with each node having 4 Infiniband 100 Gbps NICs.
It turns out that Legate has obtained near linear efficiency weak-scaling with up to 1280 CPUs and 256 GPUs.



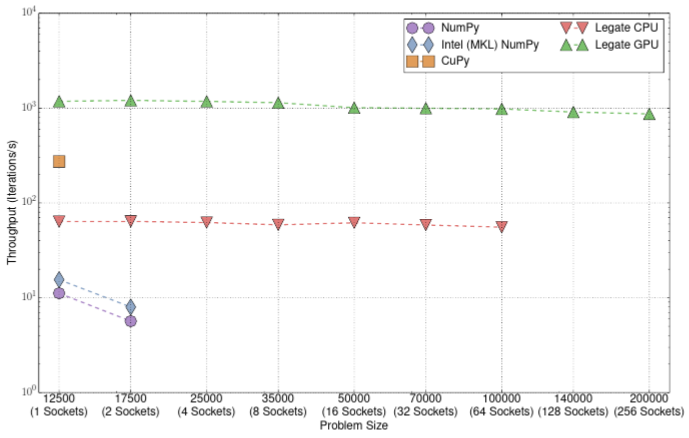
(a) Weak-scaling of a preconditioned CG solver



(b) Weak-scaling throughput of a Jacobi solver



(c) Weak-scaling of a logistic regression



(d) Weak-scaling throughput of a 2D stencil

Figure 6: the benchmarks result[1]

4 Related Works

As we have mentioned in the introductory section, people have been trying to accelerate Python ever since the rise of data science. In this section, we are to cover a few more Python libraries/frameworks/runtimes/compilers for speeding up large scale data science applications.

Only general purpose open source project are covered. In fact, a variety of domain-specified NumPy implementations do exists and a considerable number of enterprises have developed their own closed-source replacement of NumPy.

4.1 Theano / Aesara

Aiming at accelerating mathematical expressions, Theano[6], consisting of a set of NumPy-like APIs and a compiler. Before the Python code is executed, Theano compiler transform it into optimized high performance C++ code and link it as a dynamically loaded Python modules.

Theano is the pioneer of differentiable programming paradigm, the auto symbolic differentiation features allow programmers to implement numerical optimization algorithm such as the gradient descent without manually evaluating the derivatives, which is painful and error-prone. This key feature not only alleviates manual symbolic calculation, but also opens a door for advanced code generation. Having full access to the computational graph, Theano applies local graph transformations to eliminate unnecessary, slow or numerically unstable expression patterns.

Theano is orginally a research project from Google MILA groups, however MILA stopped the development and maintance in 2017. It is being continued as aesara, a community fork of Theano.

4.2 Numba

Numba[7] is similar to Theano, both libraries try to compile Python code into high performance. When comparing them, Numba is more powerful and flexible.

To overcome the performance degrade of Python interpreter and take the advantage of parallel execution, the only modification needed is simply marking functions with `@jit,@cuda.jit,@cfunc` decorator. The code then get transformed into optimized high performance code and can even utilize common hardware accelerators.

Numba implement a JIT (just in time) compiler to transform normal Python code into LLVM IR. They do optimization on the LLVM IR level and generate machine code from it. The benefit of using LLVM IR is that Numba can support for targeting different hardwares such as multi-core CPUs, GPGPUs and Google TPUs.

4.3 CuPy: array programming on a single NVIDIA GPU

Prior to the birth of Legate, the researchers at NVIDIA developed a drop-in replacement for NumPy called CuPy[8]. It is highly compatible with the N-dimensional array APIs provided by NumPy. Typically, to turn a CPU code into CUDA program can be achieve by replacing `import NumPy as np` with `import cupy as np`.

CuPy does not simple move the data onto GPU main memory and implement array operations with CUDA kernels. It is linked with NVIDIA's CUDA toolkit. cuBLAS, cuSOLVE, cuRAND, cuFFT are integrated into CuPy. CuPy also include support low-level CUDA codes. User-defined CUDA kernels written in C++ are wrapped and loaded readily.

4.4 Dask

The previous three libraries focus on single node performance boost, while Dask[9] is targeted fully utilizing the power of multi-node cluster. Dask is a task-based runtime system for parallel and distributed computation.

`dask.array`, `dask.bag` and `dask.dataframe` are the three main data structure provided, programmer use the then to compose programs. Dask analysis the program and divides the computation into tasks. The tasks are organized in a DAG called `dask graph`. The edges and vertices represent dependencies and computation tasks respectively. Dask explore the graph and schedule execution of tasks dynamically until completion.

Dask provides richer data structures than Legate, however the performance is lower than Legate. In the Legate paper, researchers at NVIDIA compared the performance and scalability of Dask and Legate. They concluded that Legate outperforms Dask in the sense of absolute speedup and weak-scalability.

4.5 Tensorflow, PyTorch and MXNet

The last five years have witnessed the tide of deep learning. As the typical size of neural networks grows, training and inference require more computation power and memory foodprint.

Tensorflow, PyTorch and MXNet are three popular deep learning frameworks, each has implemented tensor (which is a fancier name for ndarray) facilities on GPU.

5 Conclusion

In this paper review project, we give an anatomy of the Legate system, a NVIDIA research project provides accelerated NumPy-compatible N-dimensional array operations on distributed GPU cluster.

In the beginning we give a overview of the scientific Python ecosystem, where NumPy plays a central role. We then explored the Legion runtime upon which the Legate is built. Shortly after that, we presented a detailed explanation on how Legate maps NumPy data structures and computational task into Legion data and tasks.

In the last part, we covered several related works. We give a short summary on their mechanism and compared them with Legate.

References

[1] M. Bauer and M. Garland, "Legate numpy: Accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290. DOI: 10.1145/3295500.3356175.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020. DOI: 10.1038/s41586-020-2649-2.

[3] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, no. 4, pp. 681–694, 2020.

[4] W. M. Farr, S. Stevenson, M. C. Miller, I. Mandel, B. Farr, and A. Vecchio, "Distinguishing spin-aligned and isotropic black hole populations with gravitational waves," *Nature*, vol. 548, no. 7668, pp. 426–429, 2017.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, pp. 1–11. DOI: http://dx.doi.org/10.1109/SC.2012.71.

[6] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th python in science conf*, vol. 1, 2010, pp. 3–10. DOI: http://dx.doi.org/10.25080/Majora-92bf1922-003.

[7] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6. DOI: https://doi.org/10.1145/2833157.2833162.

[8] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[9] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130–136. DOI: http://dx.doi.org/10.25080/Majora-7b98e3ed-013.

A Sample code

A.1 NumPy and Legate

It is convenient to express matrix operations in NumPy API. Here we illustrate how to implement Jacobi iteration with NumPy.

```
try: import legate.numpy as np
except: import numpy as np

n = 1000
# Generate a random n x n linear system
A = np.random.rand(n, n)
b = np.random.rand(n)
```

```
# initalize solution
x = np.zeros(b.shape)
# extrat diagnal elements
d = np.diag(A)
# non-diagnal elements
R = A-np.diag(d)

# Jacobi iteration x_{i+1} \gets (b-R x_{i}) D^{-1}
for _ in range(n):
    x = (b - np.dot(R, x)) / d
```

A.2 Theano/Aesara

Here we demonstrate how to implement logistic regression in aesara.

```
import numpy
import aesara
import aesara.tensor as aet
rng = numpy.random
N, feats, training_steps = 400, 784, 10000

# generate a dataset: D = (input_values, target_class)
D = (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))

# Declare Aesara symbolic variables
x, y = aet.dmatrix("x"), aet.dvector("y")
# shared variables are kept between training iterations
# initialize the weight vector w randomly
w = aesara.shared(rng.randn(feats), name="w")
# initialize the bias term
b = aesara.shared(0., name="b")

# Construct Aesara expression graph
# Probability that target = 1
p_1 = 1 / (1 + aet.exp(-T.dot(x, w) - b))
# The prediction thresholded
prediction = p_1 > 0.5
# Cross-entropy loss function
xent = -y * aet.log(p_1) - (1-y) * aet.log(1-p_1)
# The cost to minimize
cost = xent.mean() + 0.01 * (w ** 2).sum()
# Compute the gradient of the cost w.r.t weight vector w and bias term b
gw, gb = aet.grad(cost, [w, b])

# Compile
train = aesara.function(inputs=[x, y], outputs=[prediction, xent],
                        updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = aesara.function(inputs=[x], outputs=prediction)

# Train
for i in range(training_steps):
    pred, err = train(D[0], D[1])
```

A.3 Numba

The following code gives a simple example on numba's JIT decorator.

```
from timeit import default_timer as timer
from matplotlib.pylab import imshow, show
import numpy as np
from numba import jit
```

```
@jit(nopython=True)
def mandel(x, y, max_iters):
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z * z + c
        if (z.real * z.real + z.imag * z.imag) >= 4:
            return i

    return 255


@jit(nopython=True)
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image


image = np.zeros((500 * 2, 750 * 2), dtype=np.uint8)
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print(e - s)
imshow(image)
show()
```

A.4 CuPy

With CuPy, one can speedup common scientific computation task with NVIDIA CUDA toolkit. We demonstrate how to use cuFFT.

```
import cupy as cp
import cupyx.scipy.fft as cufft
import scipy.fft
scipy.fft.set_global_backend(cufft)

a = cp.random.random(100).astype(cp.complex64)
b = scipy.fft.fft(a)  # equivalent to cufft.fft(a)
```