# Python Framework for Co-Design Applications in Exascale R&D

Ryan Marcus, Larry Cox
XCP-DO

## Introduction / Motivation

When developing implementations of "mini-apps" or "co-design applications" for Exascale R&D, it is important to demonstrate an algorithm's potential in several different forms, perhaps including:

1. Standard serial implementations
2. Standard parallel implementations (MPI, OpenMP) **(unimplemented)**
3. Emerging parallelism frameworks (CUDA, OpenCL)

Because of each of these forms traditionally requires a different codebase, demonstrating each of just these three forms would require writing three different implementations of an algorithm. This can become tiresome and costly, as implementation is not always the most important factor of co-design applications.

Thus, the goal of this framework is to provide an easy way to implement an Exascale algorithm once, and then run it, near-optimally, on multiple different forms of potential Exascale technologies.

## Supported Platforms

Initially, this framework should support:

1. Standard serial processing on standard 32-bit processors
2. MPI and OpenMP, parallelism libraries used on many super-computers **(unimplemented)**
3. OpenCL, a framework for writing programs that utilize a heterogeneous mixture of CPUs and GPUs.
4. CUDA, a proprietary framework from NVidia designed to provide a massively parallel environment on NVidia graphics cards.

## Primary methodology

In classical computer science, algorithms are written as a series of steps that can be executed by a machine in series. With new and emerging parallel technologies, steps are no longer being conducted in series, but in parallel. This has lead to a whole new class of algorithms designed to distribute data over multiple processes or threads.

Normally, it is fairly easy to take a parallel algorithm and run it in serial. Each step that would have been ran in parallel is simply executed in series, and then the results are merged together. However, it is much more difficult to run a serial algorithm in parallel, as identifying which parts of an algorithm can be ran in parallel can be challenging.

Because of this, this framework is designed around concepts that are compatible with both serial calculations and parallel operations. The two basic concepts utilized in this framework are:

1. **Array operations:** performing operations on each element of an array, or between two arrays
2. **Array reductions:** reducing an array of elements into a single element (e.g., summations, minimums, maximums)

Thus, this framework provides a way from algorithms written using these two constructions to run on a variety of platforms, using the same original source.

## Programming paradigm

The basic paradigm of this framework revolves around doing a series of identical operations to a series of non-identical inputs. Classically, if a problem needed to be solved for a set of different inputs, an algorithm designed to solve that problem would be applied to each input in series. Under this framework, an algorithm is applied to each input at the same time.

Consider an algorithm A(x), which takes in a data-point x and outputs an answer of interest. If the output of A(x) for a large number of inputs was desired, classical computation would calculate A(x) for each input. Under this framework, the algorithm A(x) can be applied to a large number of inputs simultaneously.

This framework is designed to make this transition (relatively) easy. This framework allows a user to treat a set of input data as a single piece of input data, and perform operations on a set of input data as if it were a single piece of input data.

### Example 1: Trivial Physics

In many physics models, gravity is considered a constant force pulling objects “downwards.” In some such models, it becomes necessary to calculate the “air time” or “hang time” of a particle with an initial upward velocity.

This calculation is fairly simple. The height of an object with initial upward velocity v, on a planet with the directed (in this case, negative) gravitational constant g, assuming the initial height off the ground is zero, is given by:

$$h = (0.5)gt^2 + vt$$

*h = (0.5)gt^2 + vt*

In order to calculate “hang time”, calculate the difference between the two roots of the equation. One of the roots will be *t = 0*. The other root will be twice the x-axis distance from 0 to the vertex of the parabola. Thus, “hang time” is given by:

$$t = \frac{-2v}{g}$$

*t = \frac{-2v}{g}*

Doing this calculation for even thousands of particles is trivial, but it still serves as a good example. In regular Python, one could easily make this calculation on a long list of particles:

```
particlesVelocity = [3, 2, 8, 5, … ]
hangTimes = [ ]
g = -9.8
for v in particlesVelocity:
        hangTimes.append((-2.0*v)/g)
```

While this calculation would likely be trivial for thousands of particles, this framework can still easily be applied:

```
import exascale
exascale.init()
p = exascale.get_imp()
v = p.createArray([3, 2, 8, 5, … ])
hangTimes = p.zeros_like(v)
p.s_mul(hangTimes, hangTimes,  -2.0)
p.s_div(hangTimes, hangTimes, 9.8)
```

The exascale.createArray function takes in a standard Python iterable and returns an array. Scalar operations are then applied to that array: first a multiplication of each element by -2.0, then a division of each element by g. Both of these two codes produce the same result, but depending on the capabilities of the system you are running in, the 2nd code could run on OpenCL, CUDA, OpenMP, or in serial using low-level C-packages.

## Example 2: Naïve Function Minimization

In mathematics, the problem of minimizing a function appears frequently. One naïve approach to estimating the minimum of a function is to test a large set of uniformly distributed pre-images and pick the one with the smallest image.

In traditional Python, this might look like:

```
mysteryFunction = lambda x: x**2
currentMin = -1000
for I in range(-1000, 1000):
        if mysteryFunction(currentMin) > mysteryFunction(i):
                currentMin = i
print “Smallest value: “ , (currentMin, mysteryFunction(currentMin))
```

This code is on the order of *n*, where *n* is the cardinality of the set of pre-images.

With this framework, this might look like:

```
import exascale
exascale.init()
```

```
p = exascale.get_imp()
a = p.create_array(range(-1000, 1000))
p.powf(a, a, 2)  # apply the function – can't use lambdas
minVal = p.min_reduction(a)
print "Smallest value: ", (minVal, minVal**2)
```

Notice that instead of iterating over each number between -1000 and 999 (inclusive), all of the images are evaluated at the same time. Then, a minimum-finding reduction is applied to the array, which runs in In order *log n.*

## Using the Framework

### Run levels

This framework can run code in multiple different ways, each of which is called a "run level." Normally, this framework picks the highest priority run level to execute operations. The run level that the framework is using can be queried using the get_current_run_level() function.

The run level priority is:

1. CUDA, if available
2. If not, OpenCL, if available
3. If not, MPI, if available
4. If not, OpenMP, if available
5. If not, standard serial operations.

### Software Requirements

The software required to use this framework depends on the desired "run-level." For standard, serial operation, only numpy is required. However, for CUDA operations, both numpy and pyCuda are required.

| **Run level** | **Required software** |
|---|---|
| Serial / Sequential Processing | Numpy |
| OpenMP **(unimplemented)** | Numpy + Python OpenMP implementation |
| MPI **(unimplemented)** | Numpy + Mpi4py |
| OpenCL | Numpy + pyOpenCL |
| CUDA | Numpy + pyCUDA |

### Initialization

Before calling any functions from this framework, the framework needs to be initialized. In order to initialize the framework, simply call the init() function. The init() function will automatically pick the highest available run level and setup the framework.

To force the framework to use a certain run level, call init(force_level=RUN_LEVEL_), using whichever run level constant is needed.

After calling the init() function, the run level picked can be queried with the get_current_run_level() function. This function returns an integer which will equal one of the following available constants:

| **Run level** | **Constant** |
|---|---|
| Serial / Sequential Processing | RUN_LEVEL_SERIAL |
| OpenMP **(unimplemented)** | RUN_LEVEL_OPENMP |
| MPI **(unimplemented)** | RUN_LEVEL_MPI |
| OpenCL | RUN_LEVEL_OPENCL |
| CUDA | RUN_LEVEL_CUDA |

## Getting an exascale implementation

Once the framework has been initialized, one can acquire an implementation object. This object contains the same set of functions regardless of the run level used to initialize the framework. This implementation object contains all of this framework's computational functions.

One can obtain an implementation object using exascale.get_impl().

## Creating arrays

Currently, two functions exist to create arrays: create_array() and zeros_like(). Both of these functions take in a standard numpy array or Python iterable and convert them to an array compatible with the current run level.

The create_array() function takes in an array-like object and copies each entry into an array compatible with the current run level. This is the standard function that could replace a "copy array to device" style function.

The zeros_like() function takes in an array-like object and creates a new array, compatible with the current run level, filled completely with zeros. This is a replacement for the numpy function by the same name.

An array created by the create_array() function should be considered *on the device*. In other words, the actual object that is returned by create_array() should be considered a *pointer* to an array, not an array itself.

Slicing an array or reading a specific index of an array is not possible because the data inside of the array is actually being stored on a device through whatever run level is currently being used. In order to get this data back, use the get_array() function to retrieve data from the device. The get_array() function takes in a pointer from create_array() and returns a standard numpy array with the data from the device.

Note that a pointer from create_array() and a numpy array from get_array() are not linked. If one creates an array with create_array(), performs operations on it, and then uses get_array() to retrieve the data back, modifying the returned data will not modify the data on the device.

## Working with arrays

All array operations are documented in the developer's documentation for this framework.

## Array Transpose

Within the framework, there exists a function to take the transpose of a two-dimensional array. Note that the transpose of an array is defined as:

$$AT_{ij} = A_{ji}$$

where AT is the transpose of the input array, A.

Because of the nature of exascale computing devices (highly parallel), a non-traditional transpose algorithm is used. All two-dimensional arrays are actually stored (when "on the device") as flattened one-dimensional arrays. Formally, when H is an array "on the host" and D is an array "on the device":

$$(H_{ij} = D_k) \to (k = (i * width) + j)$$

Note that, given the width of an array, it is possible to calculate the indices of any flattened-index:

$$k = iw + j$$

$$i = \frac{k - (k \% w)}{w}$$

$$j = (k \% w)$$

where k is the flattened index, i is the first index of the two-dimensional array, j is the second index of the two-dimensional array, and % is the division-remainder/modulus operator. From this point forward, i will represent  the index of the flattened array.

From this, constructing the transpose of an array is a matter of mapping indices in the input array to the indices in the result array. Consider a function that maps the set of indices in the input array to the set of indices in the result array. Formally:

$$f(x) = (i \% w)w + \frac{(i - (i \% w))}{w}$$

where w is the width of the input array, % is the division-remainder/modulus operator, and i is the index. Using some algebra, this function simplifies to:

$$f(x) = \frac{(w^2 - 1)(i \% w) + i}{w}$$

Thus, when the array transpose function is called, data from the result array at index i is copied into the result array at index f(i). Formally:

$$\forall i(I_i = R_{f(i)})$$

where i is the input array and R is the result array. In arrays with less elements than there are processors on the exascale device (when the cardinality of the input array is lower than the number of available threads), this algorithm is effectively *O*(1). Otherwise, the algorithm is effectively *O*(n).