pyamgx Documentation

Release 0.1

Ashwin Srinath

Aug 28, 2018

Contents

pyamgx is a Python interface to the NVIDIA AMGX library. pyamgx can be used to construct complex solvers and preconditioners to solve sparse sparse linear systems on the GPU.

CHAPTER 1

Features

- Provides a Pythonic interface to all AMGX C-API functions for solving linear systems on a single GPU
- Allows directly uploading matrix and vector data from SciPy sparse CSR matrices, NumPy arrays and Numba DeviceArrays, among others
- Solver settings can be provided in JSON files or as `dict` objects
- Error checking and handling: AMGX errors are automatically converted into Python exceptions

CHAPTER 2

Contents

Note: This guide provides an overview of the pyamgx library, its classes and functions. It does not contain information about AMGX algorithms (solvers and preconditioners), and their configuration. For that, please refer to the AMGX Reference Manual

2.1 Installation

pyamgx has been tested only on Linux, though it should be possible to install on Windows as well.

2.1.1 Requirements

Before installing pyamgx, you should ensure the following software packages are installed:

1. The AMGX library. The distributed (MPI) version of AMGX is not required.
2. Python 2 >= 2.7 or Python 3 >= 3.5. If you are using Python 2 < 2.7.9, you will need to install pip.
3. Python libraries SciPy and Cython. It is highly recommended to use pip to install these packages:

   ```
   $ pip install scipy cython
   ```

   If you are using the Anaconda distribution, these packages should already be installed.

2.1.2 Building and installing pyamgx

Get the source code

Download the pyamgx source either by visiting https://github.com/shwina/pyamgx and clicking "Clone or Download", or if you have Git, running the following command:

```
$ git clone https://github.com/shwina/pyamgx
```

Set environment variables

Before installing pyamgx, you should export the following environment variables:

1. `AMGX_DIR`: Path to the AMGX project root directory
2. `AMGX_BUILD_DIR`: If AMGX was built in a directory other than `$AMGX_DIR/build`, set `AMGX_BUILD_DIR` to that directory. Otherwise, you don't need to set this variable

On bash, the commands to set the above environment variables are:

```
$ export AMGX_DIR=/path/to/.../AMGX
$ export AMGX_BUILD_DIR=/path/to/.../build
```

Install pyamgx

```
$ cd pyamgx
$ pip install .
```

2.2 Demo

To give you an idea of how pyamgx is used, here is a simple demo program that sets up and solves a linear system using pyamgx, and compares the result with `scipy.sparse.linalg.spsolve()`.

```
import numpy as np
import scipy.sparse as sparse
import scipy.sparse.linalg as splinalg

import pyamgx

pyamgx.initialize()

# Initialize config and resources:
cfg = pyamgx.Config().create_from_dict({
   "config_version": 2,
        "determinism_flag": 1,
        "exception_handling" : 1,
        "solver": {
            "monitor_residual": 1,
            "solver": "BICGSTAB",
            "convergence": "RELATIVE_INI_CORE",
            "preconditioner": {
                "solver": "NOSOLVER"
        }
    }
})

rsc = pyamgx.Resources().create_simple(cfg)

# Create matrices and vectors:
```

(continues on next page)

(continued from previous page)

```
A = pyamgx.Matrix().create(rsc)
b = pyamgx.Vector().create(rsc)
x = pyamgx.Vector().create(rsc)

# Create solver:
solver = pyamgx.Solver().create(rsc, cfg)

# Upload system:

M = sparse.csr_matrix(np.random.rand(5, 5))
rhs = np.random.rand(5)
sol = np.zeros(5, dtype=np.float64)

A.upload_CSR(M)
b.upload(rhs)
x.upload(sol)

# Setup and solve system:
solver.setup(A)
solver.solve(b, x)

# Download solution
x.download(sol)
print("pyamgx solution: ", sol)
print("scipy solution: ", splinalg.spsolve(M, rhs))

# Clean up:
A.destroy()
x.destroy()
b.destroy()
solver.destroy()
rsc.destroy()
cfg.destroy()

pyamgx.finalize()
```

Output:

```
AMGX version 2.0.0.130-opensource
Built on Jul  6 2018, 12:08:15
Compiled with CUDA Runtime 8.0, using CUDA driver 9.2
pyamgx solution:  [-0.90571145  0.85909259  0.54397665  2.02579923 -0.94139638]
scipy solution:  [-0.90571145  0.85909259  0.54397665  2.02579923 -0.94139638]
```

2.3 Basic Usage

2.3.1 Initializing and finalizing pyamgx

The `initialize()` and `finalize()` functions **must** be called to initialize and finalize the library respectively.

```
import pyamgx
pyamgx.initialize()
```

(continues on next page)

(continued from previous page)

```
# use pyamgx

pyamgx.finalize()
```

2.3.2 Config objects

Config objects are used to store configuration settings for the linear solver used, including algorithm, preconditioner(s), smoother(s) and associated parameters.

Config objects can be constructed from JSON files or dict objects.

As an example, the Config object below represents the configuration for a BICGSTAB solver without preconditioning, and is constructed using the create_from_dict() method:

```
cfg = pyamgx.Config()
cfg.create_from_dict({
     "config_version": 2,
     "determinism_flag": 1,
     "exception_handling" : 1,
     "solver": {
         "monitor_residual": 1,
         "solver": "BICGSTAB",
         "convergence": "RELATIVE_INI_CORE",
         "preconditioner": {
             "solver": "NOSOLVER"
         }
     }
 })
```

Examples of more complex configurations can be found here, and a description of all configuration settings can be found in the AMGX Reference Guide.

The create_from_file() method can be used to read configuration settings from a JSON file instead:

```
cfg = pyamgx.Config()
cfg.create_from_file('/path/to/GMRES.json')
```

After use, Config objects **must** be destroyed using the destroy() method.

```
cfg.destroy()
```

2.3.3 Resources objects

Resources objects are used to specify the resources (GPUs, MPI ranks) used by Vector, Matrix and Solver objects. Currently, pyamgx only supports "simple" Resources objects for single threaded, single GPU applications. created using the create_simple() method:

```
resources = pyamgx.Resources()
resources.create_simple(cfg)
```

After use, Resources objects **must** be destroyed using the destroy() method.

```
resources.destroy()
```

Important: A `Resources` object should be destroyed only **after** all `Vector`, `Matrix` and `Solver` objects constructed from it are destroyed.

2.3.4 Vectors

`Vector` objects store vectors on either the host (CPU memory) or device (GPU memory).

The value of the optional *mode* argument to the `create()` method specifies whether the data resides on the host or device. If it is `'dDDI'` (default), the data resides on the device. If it is `'hDDI'`, the data resides on the host.

```
vec = pyamgx.Vector()
vec.create(resources, mode='dDDI')
```

Values of `Vector` objects can be populated in the following ways:

1. From an array using the `upload()` method

   ```
   vec.upload(np.array([1, 2, 3], dtype=np.float64))
   ```

2. Using the `set_zero()` method

   ```
   vec.set_zero(5) # implicitly allocates storage for the vector
   ```

3. From a raw pointer using the `upload_raw()` method. This allows uploading values from arrays already on the GPU, for instance from `numba.cuda.device_array` objects.

   ```
   import numba.cuda

   a = np.array([1, 2, 3], dtype=np.float64)
   d_a = numba.cuda.to_device(a, dtype=np.float64))

   vec.upload_raw(d_a.device_ctypes_pointer.value, 3) # copies directly from GPU
   ```

After use, `Vector` objects **must** be destroyed using the `destroy()` method.

2.3.5 Matrices

`Matrix` objects store sparse matrices on either the host (CPU memory) or device (GPU memory).

The value of the optional *mode* argument to the `create()` method specifies whether the data resides on the host or device. If it is `'dDDI'` (default), the data resides on the device. If it is `'hDDI'`, the data resides on the host.

```
mat = pyamgx.Matrix()
mat.create(resources, mode='dDDI')
```

`Matrix` objects store matrices in the CSR sparse format.

Matrix data can be copied into the `Matrix` object in the following ways:

1. From the arrays *row_ptrs*, *col_indices* and *data* that define the CSR matrix, using the `upload()` method:

   ```
   mat.upload(
       row_ptrs=np.array([0, 2, 4], dtype=np.int32),
       col_indices=np.array([0, 1, 0, 1], dtype=np.int32),
       data=np.array([1., 2., 3., 4.], dtype=np.float64))
   ```

2. From a `scipy.sparse.csr_matrix`, using the `upload_CSR()` method:

```
import scipy.sparse
M = scipy.sparse.csr_matrix(np.random.rand(5, 5))

mat.upload_CSR(M)
```

After use, `Matrix` objects **must** be destroyed using the `destroy()` method.

2.3.6 Solvers

A `Solver` encapsulates the linear solver specified in the `Config` object.

The `setup()` method, must be called prior to solving a linear system; it sets the coefficient matrix of the linear system:

```
solver = pyamgx.Solver()
solver.create(resources, cfg)

solver.setup(mat)
```

The `solve()` method solves the linear system. The two required parameters to `solve()` the right hand side `Vector` *b* and the solution vector `Vector` *x* respectively. The optional argument *zero_initial_guess* can be set to `True` to specify that an initial guess of zero is to be used for the solution, regardless of the values in *x*.

```
b = pyamgx.Vector().create(resources)
x = pyamgx.Vector().create(resources)
b.upload(np.random.rand(5))

solver.solve(b, x, zero_initial_guess=True)
```

After use, `Solver` objects **must** be destroyed using the `destroy()` method.

Typically, the `pyamgx.Solver.solve()` method is called multiple times (e.g., in a time-stepping simulation loop). For the case in which the coefficient matrix remains fixed, the `pyamgx.Solver.setup()` method should only be called once (prior to iteration).

If the coefficient matrix changes at each iteration (e.g., in a non-linear solver), the `pyamgx.Solver.setup()` method should be called every iteration. In this case, the `pyamgx.Matrix.replace_coefficients()` method can be used to update the values of the coefficient matrix, as long as the location of non-zeros in the matrix remains the same.

2.4 pyamgx Reference

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

P