errand

Release 0.2.7

Youngsung Kim

Nov 07, 2021

CONTENTS:

Welcome to Errand, the Pythonic GPU and Accelerator Interface.

errand is a Python module that enables an easy, scalable, and future-proof programming interface for accelerator hardwares such as GPUs.

NOTE : Errand is under development, not for production-use.

CHAPTER

ONE

GETTING STARTED

errand is a Python module that enables users to easily use accelerator hardware such as GPU.

errand takes responsibility of data movements between CPU and accelerators like GPU, and creates multiple threads on the accelerators. Therefore, user can focus on computation in well-known programmiong frameworks, such as Cuda, Hip, OpenAcc, and conventional pthread (more programming frameworks are coming.) **errand** takes advantages of the functionality and convinience of numpy ndarray.

errand makes use of conventional programming tools that you may be already familar with. For example, **errand** uses Nvidia CUDA compiler or AMD HIP compiler if needed. As of this writing, **errand** supports CUDA, HIP, OpenAcc(C++), and Pthread(C++).

1.1 Installation

The easiest way to install errand is to use the pip python package manager.

```
>>> pip install errand
```

You can install errand from github code repository if you want to try the latest version.

```
>>> git clone https://github.com/grnydawn/errand.git
>>> cd errand
>>> python setup.py install
```

1.2 NumPy array example in CUDA(Nvidia) or HIP(AMD)

To run the example, create two source files in a folder as shown below, and run the Python script as usual. The example assumes that at least one of the following compilers is usable: CUDA (nvcc), HIP(hipcc), C++ OpenAcc(GNU >=10), and Pthread C++ compiler(GNU)).

```
>>> python main.py
```

The following Python code demonstrates how to compute numpy arrays using multiple programming frameworks including Cuda, Hip, OpenAcc(GNU), or Pthread(GNU). Errand automatically checks and uses one of frameworks available on the system.

Python code (main.py)

```
# This example shows how to add numpy arrays using Errand.

import numpy as np
from errand import Errand

N1 = 10
N2 = 20

a = np.ones((N1, N2))
b = np.ones((N1, N2))
c = np.zeros((N1, N2))

# creates an errand context with an "order"
with Errand("order.ord") as erd:

        # build workshop with input(a, b) and output(c)
        workshop = erd.workshop(a, b, "->", c)

        # call N1 teams of N2 gofers
        gofers = erd.gofers(N2, N1)

        # let gofers do their work at the workshop
        gofers.run(workshop)

        # do your work below while gofers are doing their work

# check the result when the errand is completed
if np.array_equal(c, a+b):
        print("SUCCESS!")

else:
        print("FAILURE!")
```

While Errand handles data movements and thread generation, user needs to specify computation in an order file. For example, the order file below defines an element-wise addition of 2 dimensional array in multiple programming frameworks including Cuda, Hip, OpenAcc-C++, and PThread.

For convinience, Errand provides user with a Numpy ndarray-like interface to the input and output arguments as demonstrated below. For example, an array can be accessed through indices and the shape array is informed with shape member method.

Order code (order.ord)

```
[cuda, hip]

        // N1 teams are interpreted to Cuda/Hip blocks
        // N2 gofers of a team are interpreted to Cuda/Hip threads

        int row = blockIdx.x;
        int col = threadIdx.x;

        // the input and output variables keep the convinience of numpy

        if (row < x.shape(0) && col < x.shape(1))
```

(continues on next page)

(continued from previous page)

```
                c(row, col) = a(row, col) + b(row, col);

[openacc-c++]

        #pragma acc loop gang
        for (int row = 0; row < a.shape(0); row++) {

                #pragma acc loop worker
                for (int col = 0; col < a.shape(1); col++) {
                        c(row, col) = a(row, col) + b(row, col);
                }
        }

[pthread]

        int row = a.unravel_index(ERRAND_GOFER_ID, 0);
        int col = a.unravel_index(ERRAND_GOFER_ID, 1);

        if (row < a.shape(0) && col < a.shape(1) )
                c(row, col) = a(row, col) + b(row, col);
```

CHAPTER

TWO

ERRAND BASICS

Writing an **errand** has two parts. The first part is to define errand in Python code. More specifically, user defines a workshop(analogous to h/w such as GPU) and gofers(analogous to threads). The second part is to define an order that the gofers run in the workshop.

2.1 Writing a workshop and gofers in Python code

Following code shows typical way to define a workshop and gofers.

```
with Errand("order.ord") as erd:

        workshop = erd.workshop(a, b, "->", c)

        gofers = erd.gofers(NUM_TEAMS, NUM_GOFERS_PER_TEAM)

        gofers.run(workshop)

        # do your work below while gofers are doing their work

# The output from the errand is available after the end of the Errand context.
```

Errand uses Python context manager to wrap all activities under "errand". In the first line an Errand object is created with a file path to "order" file.

Workshop is created through Errand context handler. Errand accepts numpy ndarray or Python objects that can be converted to ndarray. Arguments is splited by an arrow argument, "->". Left arguments of the arrow argument are input arguments, and right arguments are output arguments. In the example, "a" and "b" variables are inputs and "c" is an output variable.

The second line calls gofers with the number of teams and members in each team. Errand provides user with flexible ways in calling gofers. For example, if there is only one integer argument, Errand consider it as the number of members in a team and there is only one team. Detail information on calling gofers will be explained in other pages in this site.

Next, gofers runs the order on the workshop, where technically Errand generates source code based on "order.ord" and compiles it to an shared library, and load & run the shared library.

Errand runs the "errand" asynchronously. Therefore, Errand natually supports overlapping computations between host machine and target machine.

Errand collect output from target machine and populate it to output variable.

2.2 Writing an order in various programming frameworks

As of this writing, Errand supports Cuda, Hip, OpenAcc(GNU C++), and PThread(GNU C++). More programming frameworks and compilers will be supported.

Order file is composed of one implicit Python section and multiple programming framework sections.

"order.ord"

```
# implicit Python section

enable_pthread = False # control if pthread section is enabled.

[cuda, hip]

        // first section is for cuda as well as hip

[openacc-c++: -O3]

        // second section for openacc-c++

[pthread@enable=enable_pthread]

        // third section for pthread
```

From the beginning of the file to the right before of the first named section is the implicit Python section. You can put any Python code that you can write in a Python script file. The code will be executed when the order file is loaded and the variables created will be used to control the following sections. For example, "enable_pthread" is set to False and that disables the third pthread section from execution candidates. You can create multiple versions of the same section and activates only one of them depending on system, compiler, test cases, and so on

The first named section has two target frameworks, "cuda" and "hip". In many cases, it is possible that a code can be compiled by both compilers as they share many features in common. Incompatibilities between cuda and hip in programming perspective exist on mainly data movement between CPU and GPU, and Errand takes care of the differnce behind.

User can control many aspects of compilation. In the example for openacc-c++, user added -O3 compiler options.

For the purpose of explanation, the last pthread section is disabled as explaned in the first implicit Python section.

2.3 Numpy inputs to Errand

Errand expects input argument to Errand workshop are Numpy ndarray or Python objects that can be converted to ndarray. The ndarray provides various information that Errand pulls from and populates to the Errand genrated source file.

```
import numpy as np
from errand import Errand

N1 = 10
N2 = 20

a = np.ones((N1, N2))
b = np.ones((N1, N2))
```

(continues on next page)

(continued from previous page)

```
c = np.zeros((N1, N2))


with Errand("order.ord") as erd:

        ...


        workshop = erd.workshop(a, b, "->", c)
```

In the example, “a”, “b” are input ndarray variables and “c” is output ndarray variable. “->” is a visual sign that seperates aruments between inputs and outputs.

You can use Python list instead of ndarray. However, the list is converted into ndarray Errand interanlly.

```
NROW = 2
NCOL = 3

a = [[1,1,1], [1,1,1]]
b = [[1,1,1], [1,1,1]]
c = [[0,0,0], [0,0,0]]

with Errand("order.ord") as erd:

        workshop = erd.workshop(a, b, "->", c)

        ...
```

CHAPTER

THREE

WRITING ERRAND CONTEXT

T.B.D.

CHAPTER

FOUR

WRITING ERRAND ORDER

4.1 Writing an order in Cuda

T.B.D.

4.2 Writing an order in Hip

T.B.D.

4.3 Writing an order in OpenAcc-C++

T.B.D.

4.4 Writing an order in Pthread

T.B.D.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search