POSTER
P4233
CONTACT NAME
Aditya Atluri: abcd@gwu.edu

Urutu:

A Python based Parallel Programming Library for GPUs

https://pypi.python.org/pypi/Urutu

What is Urutu?

Urutu is a Python based Parallel Programming Library for GPUs.

Writing a parallel code which supports all the programming models (like, CUDA, OpenCL etc.), programming environments (like, Windows, Linux) and different GPU Hardware (like, NVIDIA, AMD, Intel) is difficult. For Example, the CUDA code cannot be run on GPUs which does not support CUDA. The main issue faced by GPU programmers is portability of the code. There is no single "*library*" or "*programming language*" which supports all the features at the same time i.e., just single code snippet running on all platforms and GPUs.

Urutu is a solution to this problem. It translates the code to machine and OS specific code. It uses Python to build the GPU code. It focus more on portability.

As Python is beautiful and elegant language it brings it's grace to GPU programming. It makes easier to write GPU code rather than writing in "C/C++".

Features of Urutu:

- **Platform independent:** Urutu works on all platforms which supports Python and GPU.
- **GPU independent:** It works on all GPUs and CPUs which support CUDA and OpenCL.
- **Thread level Parallelism:** Urutu supports usage of hardware threads in the code which makes it more powerful.
- **Algorithm design:** Urutu supports "CUDA" and "OpenCL" primitives, which makes it powerful. And, flexible to create new algorithms rather than using pre-built libraries.
- **Numpy Arrays:** Urutu uses Numpy arrays rather than Python "Lists" which makes it faster to compute on GPU.
- **Support for CUBLAS API:** Urutu supports CUBLAS by using the Numpy package scikits.cuda.
- **Automated Code Generation:** Urutu produces device code at runtime depending on the GPU used.
- **Automated "data-type" detection:** Urutu detects the data-type of the array and allocates it on GPU.

An Example

The decorators are used to run the Python code on GPUs. By specifying the **'CL'** and **'CU'** flags to the decorator function, respective API is invoked.

The variables, **Tx, Ty, Tz** represents the number of 3-D threads need to be invoked and **Bx, By, Bz** represents the number of 3-D blocks needs to be created for the task.

```
from Urutu import *          # import Urutu
import numpy as np           # import numpy

@Urutu("CL")                 # Run using OpenCL
def div(a, b, c):            # Passing arguments
   Tx, Ty, Tz = 100, 1, 1 # Create Threads
   Bx, By, Bz = 1, 1, 1   # Create Blocks
   c[tx] = a[tx] / b[tx]  # Math Operation
   return c                  # Returns the result

@Urutu("CU")                 # Run using CUDA
def add(a, b, d):            # Passing arguments
   Tx, Ty, Tz = 100, 1, 1 # Create Threads
   Bx, By, Bz = 1, 1, 1   # Create Blocks
   d[tx] = a[tx] + b[tx]  # Math operation
   return d                  # Returns the result

# Declaring variables
a = np.random.randint(10, size = 100)
b = np.random.randint(10, size = 100)
c = np.array(a, dtype = 'f')
d = np.empty_like(a)

# Calling the functions
print "The Array A is: \n", a
print "The Array B is: \n", b
print "Running on OpenCL..\n", div(a, b, c)
print "Running on CUDA..\n", add(a, b, d)
```

Why do you need Urutu?

With Python's portability to various platforms and Urutu compiler's translation to various GPUs, it makes your code more portable.

- High Performance is maintained.
- Simple syntax.
- No C/C++ snippets in the code.

How it works?

- Decorator ***@Urutu*** is added to the function which is to be executed on GPU. The decorator takes the flags **'CL'** and **'CU'** as it's arguments.
- The Python interpreter checks for any syntax errors. If there are any, it is prompted to the user. Urutu doesn't come into play here. The arguments of the decorator and the valid Python code are passed to the Urutu compiler.
- The compiler checks for number of Threads and Blocks declared in the function and invokes them.
- The arguments passed to the function are taken, GPU memory spaces are allocated and the data is transferred to them from CPU.
- The compiler translates the Python code into valid OpenCL or CUDA code. The OpenCL and CUDA code are executed.
- The results are transferred back to CPU after complete execution of the code. These are returned to the Python interpreter as the return arguments.

Here, the variables "tx", "ty", "tz", "bx", "by", "bz" represent each thread and block for respective dimension. These are hardcoded variables, they cannot be re-assigned. Whereas, the variables, "Tx", "Ty", "Tz", "Bx", "By", "Bz" represent the total number of threads and blocks that are needed to run the kernel code.

All the variables used inside the function must be passed as arguments i.e., the input and output "Numpy" arrays should be passed as arguments.

The return types can be any number of variables passed to the function. New variables created cannot be returned to the interpreter.

Supports data types supported by "Numpy"

Future Work

- Support for more Python keywords.
- Automated Device detection.
- Automated variable declaration.
- Faster compiler.
- Integration of CUDA and OpenCL APIs.
- Dynamic Parallelism support for CUDA GPUs.