SDP Memo 50: The Accelerator Support of Execution Framework

Document number……SDP Memo 50
Document Type……MEMO
Revision……1.00
Author……Feng Wang, Shoulin Wei, Hui Deng and Ying Mei
Release Date……2018-09-18
Document Classification…… Unrestricted

Lead Author	Designation	Affiliation
Feng Wang		Guangzhou University/Kunming University of Sci & Tech
Signature & Date: Feng Wang 数字签名者：Feng Wang DN：cn=Feng Wang, o=Guangzhou University, ou, email=cnwangfeng@gmail.com, c=CN 日期：2018.09.18 21:35:52 +08'00'		

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Table of Contents

SDP MEMO DISCLAIMER **2**

TABLE OF CONTENTS **2**

LIST OF FIGURES **3**

LIST OF TABLES **3**

LIST OF ABBREVIATIONS **3**

INTRODUCTION **4**

BACKGROUND **4**

GPU TECHNOLOGY 4

PYTHON AND GPU SUPPORT 5

EXECUTION FRAMEWORK 6

DALIUGE AND GPU SUPPORT **6**

SPARK AND GPU SUPPORT **10**

CONCLUSION **13**

REFERENCES **13**

REFERENCE DOCUMENTS 13

List of Figures

Figure 1. Logical graph of the DALiuGE

Figure 2. The diagram of Numba work

List of Tables

Table 1. Answer of the Questions Investigated

List of Abbreviations

CPU .....Central Process Unit
GPU..... Graphical Process Unit
SDP..... Science Data Processor
SKA..... Square Kilometre Array

Introduction

Super-scale massive data processing is a significant issue in Square Kilometre Array (SKA) – Science Data Processor (SDP). To implement a data reduction system which could process the full data in real time, hardware accelerator technology is a choice of certainty. For the frameworks mentioned in SKA-SDP consortium, 5 questions are urgently demanded to make an investigation on how current mature Execution Frameworks (EF), i.e., DLiuGE and Spark support accelerators, in particular GPUs.

According to the requirements of the SKA-SDP, we need to answer the following questions.

- Q1: Do they recognize accelerator tasks as distinctly schedulable tasks from the CPU tasks, and are they are able to directly invoke GPU kernels?
- Q2: If there is no in-built support for directly invoking accelerator tasks can normal tasks adequately launch accelerator kernels (as, e.g., can be done in Spark/PySpark)
- Q3: Does the framework handle movement of data between the CPU memory and the accelerator memory?
- Q4: Is the framework able to interleave data movement and accelerator tasks?
- Q5: How much accelerator support is available through third-party libraries and how will these be supported in the long term?

In this document, the accelerator in the document is defined as the Graphics Process Unit (GPU) manufactured by NVIDIA Corp (http://www.nvidia.com). Meanwhile, considering the current programming languages we would use in astronomical data reduction, we focus on Python and C/C++ languages.

Background

GPU Technology

The graphics processing unit (GPU), as a specialized computer processor, addresses the demands of real-time high-resolution 3D graphics compute-intensive tasks. NVIDIA GPUs power millions of desktops, notebooks, workstations and supercomputers around the world, accelerating computationally-intensive tasks for consumers, professionals, scientists, and researchers.

The Compute Unified Device Architecture (CUDA) is accessible to software developers through CUDA-accelerated libraries, compiler directives such as OpenACC, and extensions to industry-standard programming languages including C, C++ and Fortran. C/C++ programmers use 'CUDA C/C++', compiled with *nvcc*, Nvidia's LLVM-based C/C++ compiler.

In addition to libraries, compiler directives, CUDA C/C++ and CUDA Fortran, the CUDA platform supports other computational interfaces, including the Khronos Group's OpenCL, Microsoft's DirectCompute, OpenGL Compute Shaders and C++ AMP. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Haskell, R, MATLAB, IDL, and native support in Mathematica.

Python and GPU Support

Python is one of the most popular programming languages today for science, engineering, data analytics and deep learning applications. However, as an interpreted language, it has been considered too slow for high-performance computing.

There are many third-party GPU development kits or packages supporting development for CUDA applicationsp under Python environment. C/C++ is the most commonly used language to extend the function of Python using *.so and ctypes. To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header "Python.h". For example, PyTorch is an effective machine learning package which could support GPU, which fully uses the CPython extension interface.

PyCUDA is one of the most commonly used for developing CUDA Application, which gives an easy, Pythonic access to Nvidia's CUDA parallel computation API. Meanwhile, PyCUDA has wrapped several CUDA APIs using ctypes. In a word, PyCUDA has the following features.

- Object cleanup tied to lifetime of objects. This idiom, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code. PyCUDA knows about dependencies, too, so (for example) it won't detach from a context before all memory allocated in it is also freed.
- Convenience. Abstractions like pycuda.compiler.SourceModule andpycuda.gpuarray. GPUArray make CUDA programming even more convenient than with Nvidia's C-based runtime.
- Completeness. PyCUDA puts the full power of CUDA's driver API at your disposal, if you wish.
- Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions.
- Speed. PyCUDA's base layer is written in C++, so all the niceties above are virtually free.

Execution Framework

Industrial data-intensive applications often use data parallel frameworks such as MapReduce, Dryad or Spark to handle parallel chunks of data independently. Such data parallelism virtually allows processing capabilities to scale indefinitely. While these data parallel frameworks provide horizontally scalable solutions, two issues arise when directly using them out of the box for modelling, executing, and managing (radio) astronomical data processing. One is how data parallelism works by partitioning a large/huge data set (or data stream) into smaller splits (or streams) and each of works is then processed in parallel. Another is how to time-critical, deadline-sensitive workloads.

The Data Activated Liu Graph Engine (DALiuGE) is an execution framework for processing large astronomical datasets at a scale required by the Square Kilometre Array Phase 1 (SKA1). It is a pure Python Application which includes an interface for expressing complex data reduction pipelines consisting of both data sets and algorithmic components and an implementation run-time to execute such pipelines on distributed resources. By mapping the logical view of a pipeline to its physical realisation, DALiuGE separates the concerns of multiple stakeholders, allowing them to collectively optimise large-scale data processing solutions in a coherent manner. The execution in DALiuGE is data-activated, where each individual data item autonomously triggers the processing on itself. Such decentralisation also makes the execution framework very scalable and flexible, supporting pipeline sizes ranging from less than ten tasks running on a laptop to tens of millions of concurrent tasks on the second fastest supercomputer in the world.

Apache Spark is an open-source distributed general-purpose cluster-computing framework which is implemented in JVM based languages such as Scala and CLojure. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.

DALiuGE and GPU Support

To further confirm the answers, we analyzed the relevant source codes and explained the reasons in detail. The newest version of the DALiuGE can be download from

https://github.com/ICRAR/daliuge

- Q1: Do they recognize accelerator tasks as distinctly schedulable tasks from the CPU tasks, and are they are able to directly invoke GPU kernels?

Ans: Not support but could be added in the near future. The DALiuGE scheduler currently has the capability to differentiate between CPU or GPU drops. For example, in the next version of the logical graph editor, the developer will be able to specify “Drop A has to run on GPUs”. During the translation between logical graphs to physical graphs, DALiuGE will allocate all instances of Drop A to a compute node that has GPUs. But DALiuGE is not concerned with allocating GPU resources to run these instances for reasons discussed below.

We discuss this question from two aspects.

1. DROP Types

Current design philosophy of DALiuGE is that the execution framework does not need to identify the type of DROP during run-time. The DALiuGE is a data driven execution framework. The DROP is the basic execution unit which could be written by C/C++ or Python, which is the implementation of one specified task. The source code of the DALiuGE shows that six types of DROPs (APP_DROP_TYPES = ['Component', 'BashShellApp', 'mpi', 'DynlibApp', 'docker', 'DynlibProcApp']) are supported. However, these types are only used during Logical Graph (LG) design, thus during the design time.

The real DROP and its definition are dependent on the Physical Graph (PG). After the program deployment, the LG could be converted into the PG which includes the information of computing nodes, data flow and relevant DROPs. A JSON file was used to store the PG, which describes the relationship between the DROP instance and the corresponding category. For example, the part of the following JSON files will create an instance using class “dfms.deploy.cnlab.loadMSlist.LoadMSlistApp”.

```
{"category":"Component", "appclass":"dfms.deploy.cnlab.loadMSlist.LoadMSlistApp",
```

The key idea is that the execution framework is responsible for “triggering” the execution and its lifecycle (starting, stopping, monitoring etc), but it is not directly involved in “executing code” for the drops.

2. GPU Task Invocation

The DALiuGE can support running GPU-based task but must use third-part packages. As we knew, the DALiuGE is a python-centric system which cannot support GPU natively. Therefore, the GPU support of the Python is heavily dependent on the third-party packages such as PyCUDA and so on.

The easiest way to run GPU DROP is to use BashShellDrop which can directly invoke a standalone program. This DROP should be written as follows.

```
import pycuda.driver as cuda

import pycuda.autoinit

from pycuda.compiler import SourceModule

…..
# Transfer data

# CUDA Programming

mod = SourceModule("""

 __global__ void doublify(float *a)

 {

  int idx = threadIdx.x + threadIdx.y*4;

  a[idx] *= 2;

 }

 """)

……

# Invoke kernel

func = mod.get_function("doublify")

func(a_gpu, block=(4,4,1))


# Get Back Data
```

The execution of the DROP is based on process mechanism. It means the number of DROP invocation is dependent on the machine's performance and resource consumption. The following codes included in bash_shell_app.py can prove that a new process would be created when the DAliuGE launches a DROP.

```
# Run and wait until it finishes

   process = subprocess.Popen(cmd,

                              close_fds=True,

                              stdin=stdin,

                              stdout=stdout,

                              stderr=subprocess.PIPE,
```

```
                    env=env)
```

- Q2: If there is no in-built support for directly invoking accelerator tasks, can normal tasks adequately launch accelerator kernels (as, e.g., can be done in Spark/PySpark)

Ans: Yes.

We can refer to answer of the previous question. According to the tutorial and manual of the PyCUDA, PyCUDA supports either asynchronous kernel launch or synchronous kernel launch. Meanwhile, this is an example provided by PyCUDA to show the feature of Kernel Concurrency. In a word, as long as the ability of tasks launch for PyCUDA is enough, the DALiuGE is capable of supporting high performance kernel launch as well.

- Q3: Does the framework handle movement of data between the CPU memory and the accelerator memory?

Ans: Not support in current version. IS IT NECESSARY?

Current DALiuGE cannot support. Actually, the DALiuGE is a task scheduler rather than a task module, which aims to provide a stable, flexible and static task mapping and execution. The DROP can be divided into 2 types. One is the DATA DROP and another is the APP DROP. For example, the “FITS File” and “Images” are DATA DROP. The “Load FITS File” and “Imaging” are APP DROP.

Figure 1. Logical graph of the DALiuGE

The data transfer between two DROPs is a big problem for design GPU-based distributed system. According to the features of the DALiuGE, the data transportation between two adjacent DROPs is unavoidable. As mentioned before, the DALiuGE is a data driven execution framework. A new process or a thread (for some PythonApp Drops) would be created when one DROP has been launched. Therefore, The APPDROP of “Imaging” must obtain the data from the DATADROP of “FITS FILE” and further transfer the data from HOST to GPU memory by APPDROP itself. In other words, the DALiuGE is in charge of the data transfer between HOST memory but the DROP has to transfer to GPU memory by itself.

- Q4: Is the framework able to interleave data movement and accelerator tasks?

Ans: No.

Under the DALiuGE framework, the data transfer and task are totally scheduled by the DALiuGE. As a data-driven framework, the DALiuGE must guarantee that the subsequent DROP would be invoked after the prior DROP finished. The event mechanism is used to guarantee the status change of each DROP respectively. This implementation limited the ability of interleaving data movement and task invocation.

Furthermore, PyCUDA cannot support some new features of CUDA. To improve the data transfer performance, some techniques, such as Peer-to-Peer Transfers between GPUs, Peer-to-Peer memory access, RDMA and GPU Direct for Video are developed and supported in current CUDA SDK. These new techniques could significantly promote the computing performance and decrease the programming difficulties. However, all the DROPs for GPU are developed using PyCUDA, which constraints the performance optimization.

- Q5: How much accelerator support is available through third-party libraries and how will these be supported in the long term?

Ans: Sufficient

Due to the continuous and quick development, Python language is becoming the world's most popular coding language. Thus, many packages or third-party SDKs are developed to extend the Python function and support more features.

Since the first version was released in 2009, PyCUDA has been developing for about 10 years. New features are continuously integrated into the package. It can be expected that the PyCUDA has a stable developing roadmap.

Meanwhile, more packages began to support GPU computing. For example, Numba, Theano, Keras, PyTorch and so on. We realize that more open source communities would develop new gpu-based packages for Python.

Spark and GPU Support

Spark has emerged as the infrastructure of choice for developing in-memory distributed analytics workloads. It provides high-level abstractions in multiple languages (e.g., Java, Scala, and Python) that hide the underlying data and work distribution operations such as data transfer to and from the Hadoop Distributed File System (HDFS) or that maintain resiliency in

the presence of system failures. Spark also provides libraries for relational Online Analytical Processing (OLAP) using SQL, machine learning, graph analytics, and streaming workloads. These features enable developers to build complex analytics workflows quickly to support different data sources in various operating environments.

Although many documents introduced how to integrated GPU tasks into Spark, few can really work due to the wrong package version and many unknow errors. Many open source codes from Github are semi-finished product or even a concept, which cannot be successfully deployed and tested. Even in the very few systems that can successfully installed and deployed, Python language cannot be used for programming.

The technical documents titled “GPU Computing with Apache Spark and Python" is one of the valuable documents which provides the reasonable technical solution for Spark Under GPU. We tested the example codes introduced in this document and tried to analyze the low-level technical implementation.

- Q1: Do they recognize accelerator tasks as distinctly schedulable tasks from the CPU tasks, and are they are able to directly invoke GPU kernels?

Ans: No.

Obviously, the Spark cannot support GPU directly unless it invokes third-party packages which written by C/C++ language. According to the [R1], Numba is an open-source, type-specializing compiler for Python functions. Meanwhile, which can translate Python syntax into machine code if all type information can be deduced when the function is called.

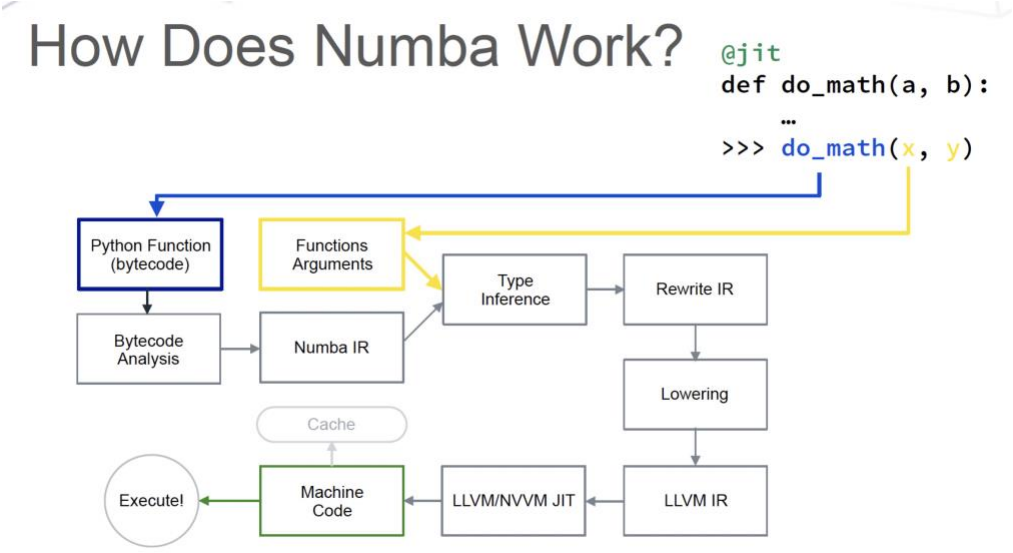

Figure 2. The diagram of Numba work

- Q2: If there is no in-built support for directly invoking accelerator tasks, can normal tasks adequately launch accelerator kernels (as, e.g., can be done in Spark/PySpark)

Ans: Yes.

- Q3: Does the framework handle movement of data between the CPU memory and the accelerator memory?

Ans: No

The Spark has thorough mechanism to control the data flow. Especially the RDD technique could provide high performance distributed data access. With the support of Numba, Python can support handling device memory directly.

```
In [6]: %timeit zero_suppression_gpu[threadsperblock, blockspergrid](x, 50, out)

        The slowest run took 7.42 times longer than the fastest. This could mean
        that an intermediate result is being cached.
        1000 loops, best of 3: 927 µs per loop

In [7]: gpu_x = numba.cuda.to_device(x)
        gpu_out = numba.cuda.to_device(out)

        %timeit zero_suppression_gpu[threadsperblock, blockspergrid](gpu_x, 50, gpu_out)

        1000 loops, best of 3: 198 µs per loop
```

- Q4: Is the framework able to interleave data movement and accelerator tasks?

Ans: No.

It is obvious from the following codes that we have to manually transfer the data from host to device using codes like cuda.to_device(im0) and cuda.to_device(im1).

```
def cross_power_spectrum(im0, im1):
    f0 = as_complex64(cuda.to_device(im0))
    f1 = as_complex64(cuda.to_device(im1))
    cufft.fft_inplace(f0)
    cufft.fft_inplace(f1)
    # cps == cross-power spectrum of im0 and im2
    d_cps = elemwise_mult_conjugate(f0, f1)
    cufft.ifft_inplace(d_cps)
    cps = complex_abs(d_cps).copy_to_host()
    return cps
```

However, Spark RDD mechanism would improve the data transfer performance in CPU memory or between computing nodes.

- Q5: How much accelerator support is available through third-party libraries and how will these be supported in the long term?

Ans: A few.

It is very hard to search the relevant documents from the Internet. Meanwhile, only a few open source codes can be found.

Conclusion

In a word, we can gather all the answers and fill into Table 1. Obviously, in current stage, neither DALiuge nor Spark can support GPU very well. The functions of third-party SDKs or libraries are limited, which cannot provide new features of current GPU technology.

Table 1. Answer of the Questions Investigated

No.	Name	Q1	Q2	Q3	Q4	Q5
1	DALiuGE	N	Y	N	N	Sufficient
3	Spark	N	N	N	N	A few

References

Reference Documents

eference Number	Reference
R1	DALiuGE, https://www.sciencedirect.com/science/article/pii/S2213133716301214
R2	Stan Seibert and Siu Kwan Lam, GPU Computing with Apache Spark and Python, https://www.slideshare.net/continuumio/gpu-computing-with-apache-spark-and-python