Data Visualitation Project : Ray Tracing with VTK

Aymeric MILLAN & Arthur VIENS

Lecture given by Julien Wintz

11th of March 2022

Abstract

This documents contains the presentation of our work on implementing a RayTracing algorithm using the VTK library and Python's bindings.

First, we are going to do a brief sate of the art.

Learning VTK's logic and syntax was a difficulty

1 Running the code

The source code is available on my GitHub branch, under the "Project" folder. Only the python files are the ones with the final code, not the jupyter notebook files.

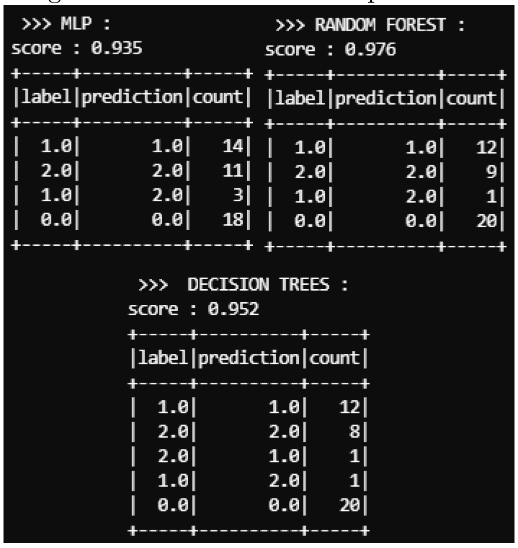
In order to run the code, you will need to have an Hadoop environment correctly set with the right python modules installed. Also, you might need to change the path of the data folder in order to run the ML codes.

2 Implementation logic

2.1 Spark ML : Iris Classification

In this part, we want to use the abstraction of Spark to compute machine learning algorithms on distributed files without having to manage the parallelism. I used three different calssifiers : Multilayer Perceptron, Random Forest and Decision Trees. The image following is the number of prediction for each model, and their scores.

Figure 1: Models' scores and predictions



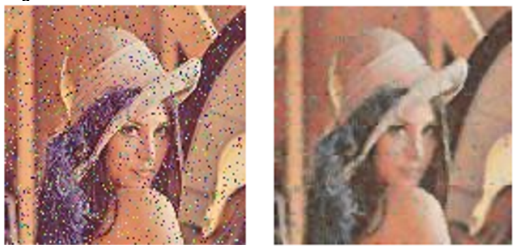
```
>>> MLP :                      >>> RANDOM FOREST :
score : 0.935                  score : 0.976
+-----+----------+-----+       +-----+----------+-----+
|label|prediction|count|       |label|prediction|count|
+-----+----------+-----+       +-----+----------+-----+
|  1.0|       1.0|   14|       |  1.0|       1.0|   12|
|  2.0|       2.0|   11|       |  2.0|       2.0|    9|
|  1.0|       2.0|    3|       |  1.0|       2.0|    1|
|  0.0|       0.0|   18|       |  0.0|       0.0|   20|
+-----+----------+-----+       +-----+----------+-----+

            >>> DECISION TREES :
            score : 0.952
            +-----+----------+-----+
            |label|prediction|count|
            +-----+----------+-----+
            |  1.0|       1.0|   12|
            |  2.0|       2.0|    8|
            |  2.0|       1.0|    1|
            |  1.0|       2.0|    1|
            |  0.0|       0.0|   20|
            +-----+----------+-----+
```

As we can see, the Random Forest Classifier seems to have the best score with this dataset. But it is probably due to overfitting since the dataset is quite small (150 samples).

2.2 Spark : parallelisation of the image processing algorithm "MedianFilter"

For this part, the main challenge was to truncate the image buffer in order to compute local median filter algorithms. To not reinvent the wheel, and to use more librairies to this project, I used the median_filter function from scipy's "ndimage" module. The true challenge of this part was to correctly slice and concatenate the image buffer. Moreover, I had to dive into the documentations of numpy, scipy and pyspark to know the methods I needed. It was a lot of internet search, but I finally made it in a easy and quickly understandable way.

Figure 2: Lena before and after MedianFilter

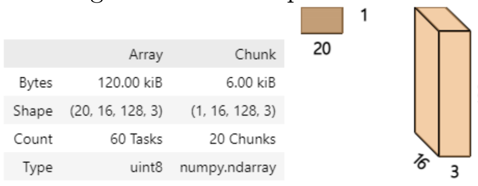


The algorithm splits the image in a certain number of blocks. Each process computes locally a median filter, and we concatenate the arrays we got from each process to rebuild the picture.

3 Dask

We are now going to do similar works that we have done with Spark, but using the **Dask** framework. This framework is younger than Spark. Which means, it is less mature, so we won't have as much as functionnalities and robustness, but it is much more dynamic and user-friendly. For example, this is the output when reading an image with dask.

Figure 3: Dask output interface



	Array	Chunk
Bytes	120.00 kiB	6.00 kiB
Shape	(20, 16, 128, 3)	(1, 16, 128, 3)
Count	60 Tasks	20 Chunks
Type	uint8	numpy.ndarray

The framework gives us, in a pretty way, the dimension of our array, the size of it, its type, and other informations. To achieve this same thing in Spark, it takes more than a single line of code. Here we can see that our image is 128×128, with a 3rd dimension that corresponds to the RGB values since we are working in colors.

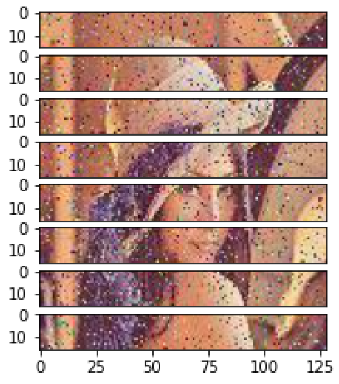
3.1 Dask ML : flower classification

In this sections, we used Dask's framework to predict the class of flowers, just as we did with Spark in the previous part. Dask was really easier to approach, since doing machine learning with Dask is just doing the ML you know (sklearn, xgboost, etc..) and plugging it to the dask cluster. The syntax stays the same. For this part, all three models have the same predict score at the end (93 %), this is probably because of the size of the dataset, which is too small to be able to see difference. The code is available in the python file.

3.2 Dask : parallelisation of the image processing algorithm "MedianFilter"

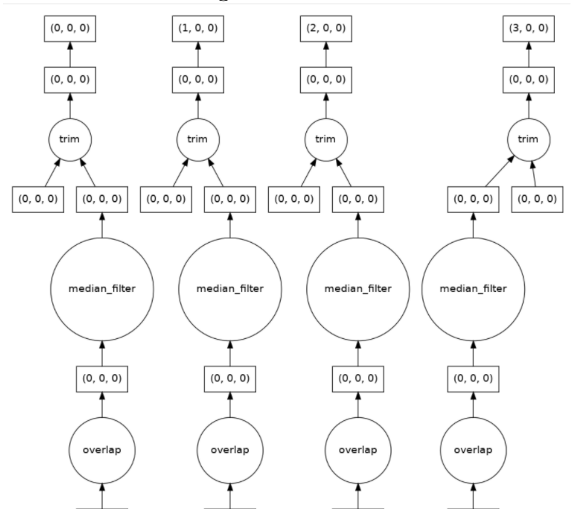
In order to compute the parallelized median filter, we first have to truncate ou picture in the number of partition that we want. Dask doesn't work exactly like Spark. How I did is I first splited the Lena picture in the number of partition.

Figure 4: Tiled Lena



Then, I distributed the calculus between all these separated images, wich are all in different files. Here we have 8 partitions and our image is 128×128, so each partition will be of size 128×16, times 3 for the 3 RGB canals. This means that we won't have to compute the local buffer **inside** the function, as we did with Spark.

Figure 5: DAG



The figure above shows part of the task graph that Dask has generated. Once again, there was an implementation of the median filter function in the framework. The really difficult part here was to embrace the framework and get use to Dasks DataFrames and parallelisation methods.

But once I understood the concepts, I found the framework much more intuitive and user-friendly than Spark. In just a few lines of code, we could achieve the same results as Spark.

Conclusions

In this project, I tried to minimize **my** code. By this, I mean that when I could find a useful function that does what I need (e.g. the median filter), I decided to spend more time on reading the documentation and integrating a new framework to my code, rather than trying to reinvent the wheel by myself. This was intended, to focus more on learning how to learn new framrworks than learning a particular one. Hadoop, Spark, and Dask give a level of abstraction that makes one more productive while working on large data sets. We do not have to manage the distribution of the files between the nodes, nor the distribution of the calculus between the CPUs. But these tools still let us adapt as much as we want our system to be working on one special environment.

Doing the work of setting up the environment, installing the right modules or setting the right system variables made me realize how much every technologies are connected together, like Java, Python and C/C++ for this project. But it also made me realize the hard work of developpers who make a high abstraction level so people can work without having to think of hardware details as we have to do when working in a C/C++ environment. Of course, even when using such frameworks and high-level implementation, one must be aware of what happends inside the blackbox, to better use it.