International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 06 | June 2020 www.irjet.net p-ISSN: 2395-0072

DataFrames on the GPU

Aditya Puranik[1]

[1]*Student, The National Institute of Engineering, Mysuru*

Abstract - *A DataFrame is a data structure which supports a variety of operations like map, reduce, filter, join, Etc on tabular data. It is a primitive non-relational and in-memory database system with a focus on speed and efficiency rather than features. They are used extensively in data science and machine learning pipelines. Graphical Processing Units(GPU's) are computation devices which are highly parallel in nature built for graphics applications. However, they are commonly used as accelerators in compute intensive and parallelizable applications. In this paper we explore the possibility of accelerating a number DataFrame operations on the GPU written in a high-level language (Julia). In particular we implement the three-fundamental big-data operations which are map, reduce and filter.*

Key Words: **GPU**, **DataFrame**, **Julia**, **CUDA**, **Map-Reduce**

1.INTRODUCTION

While an interactive environment is great for understanding data, associated challenges and writing code, with large datasets we require speed and interactivity is no longer a requirement once the solution code has been written. Often this has meant that the user must rewrite their program in a lower level language like C++ which is performant. This is known as the two-language problem wherein prototyping is done in a High-level language like Python or MATLAB and later the code is rewritten in a lower level language like C++ for performance reasons. With a large number of scientists and engineers having a non-CS background this is a very challenging problem for them.

Julia is a high-level language which aims to solve this "two-language" problem [1]. The idea is that despite having the syntax of a high-level language which is quite similar to that of Python its performance is quite similar to that of C or C++ often off by a factor of up to two which is great to save developer time. It has a number of high-level language features like pythonic syntax, interactive REPL environment, metaprogramming, Etc It also has a fantastic GPU stack which allows us to write both high-level
and low-level code to be executed on our GPU's. Tim Besard's paper on this explains the GPU stack of Julia very well [2].

GPU's are devices connected to a computer through the PCIe lane. They possess a large number(1000's) of less "smart" cores which are capable of processing independent of each other in parallel. However, being less smart they are extremely slow for serial tasks but blazingly fast for parallel tasks. GPU's are often described using the SIMT(Single instruction multiple thread) model which is analogous to the SIMD(Single instruction multiple data) model. The idea is that all of these "less smart" cores are performing the same instruction albeit on different data. There can be a level of thread divergence where two threads may be performing different operations concurrently but that generally has a significant performance penalty. GPU's are much more complicated than the simple description we have given above as that is beyond the scope of this paper. There are a number of performance considerations we must have while designing GPU applications. Factors like memory transfers (RAM <--> GPU), synchronization barriers and thread divergence can slow down GPU applications considerably making its performance orders of magnitude worse than the CPU. Fortunately, a number of operations concerned with DataFrames are "embarrassingly parallel" making GPU's an ideal porting target for DataFrame applications.

CUDA which stands for (Compute unified device architecture) refers to NVIDIA's GPU platform for their own GPU's. It is an extremely refined software stack upon which a number of bleeding machine learning and data stacks are built. It is due to this refinement and production quality software stack we have chosen CUDA. When we wish to use Julia with NVIDIA a number of packages like CUDAnative.jl, CUDAdrv.jl and CuArrays.jl allows us to write almost native Julia code which compiles for the NVIDIA GPU. Currently, work on the AMD stack is ongoing on the Julia GPU stack hence the hope is that eventually we can write the same code for both NVIDIA and AMD cards as Julia's Internal compiler infrastructure can take care of translations.

2. Operations

2.1. Map

Map is a fundamental operation which applies a higher-level function to a list(s).

```
julia> map(x -> x * 2, [1, 2, 3])
 3-element Array{Int64,1}:
  2
  4
  6
julia> map(+, [1, 2, 3], [10, 20, 30])
 3-element Array{Int64,1}:
  11
  22
  33
```

Listing-1: Map examples

Since each element can be computed independent of the other elements it's quite straight forward to implement a GPU version. One advantage of using Julia compared to Python is that we can directly write native Julia code for out mapping function and the compiler will generate an efficient version provided that the function's code is type stable and uses only static fundamental data types (int32, int64, bool, float32, float64,.etc) as GPU's cannot support complicated datatypes like Strings and BigInt's out of the box.

Another benefit of using Julia is that it will automatically replace functions with the best implementation available. Using a complicated expression like

x -> x^2 + sin(y) + 1.9sqrt(|cos(z)|) (1)

Where x, y and z are different columns of the DataFrame. The compiler will automatically replace the call to sin(x) with NVIDIA's sin(x) however if a function isn't implemented by NVIDIA, Julia can use its internal definition to generate code that will work. There is absolutely no guarantee that this code will be optimal. It is most likely that it will be non-performant as internal definitions were written for the CPU which generally has a lot of branches which is really bad for the GPU performance wise. It is the user's responsibility to profile the code and/or check the output assembly to check for badly generated code for functions that aren't fundamental or defined by them in a way which is good for the GPU.

Given p parallel processors and n elements, map can be computed in O(n/p)

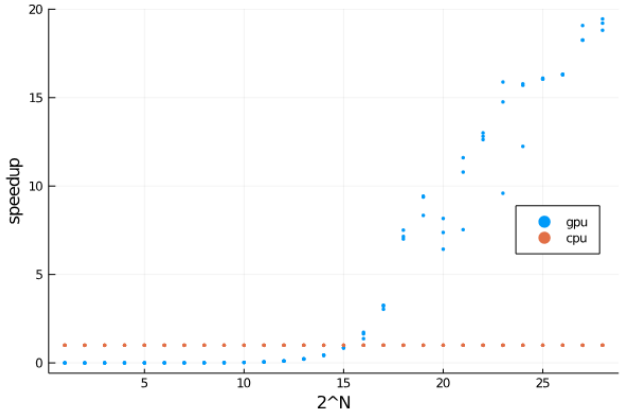

Figure-1: GPU speedup over CPU for map operation

The above benchmark shows us the speedup of the CUDA GPU version relative to the CPU version on the simple function $f(x) = x^2$. For small array sizes the CPU is significantly faster with the GPU overtaking it at around $2^{17} \sim 10^5$ elements with the GPU getting a 20x speedup for 2^{28} $2.6*10^8$ elements. We must also note that Julia has auto-vectorized the CPU version. As the input function f(x) gets more complicated the GPU speedup is expected to be higher. Also our map is only using a single input vector, adding more input vectors like in equation(1) will perform better on the GPU.

2.2. Reduce

Reduce applies a binary operator to each element of a given list.

```
julia> reduce(*, [2; 3; 4])
 24
julia> reduce(xor, [7; 8; 9])
6
#0111 xor 1000 xor 1001 = 0110
```

Listing-2: Reduce examples

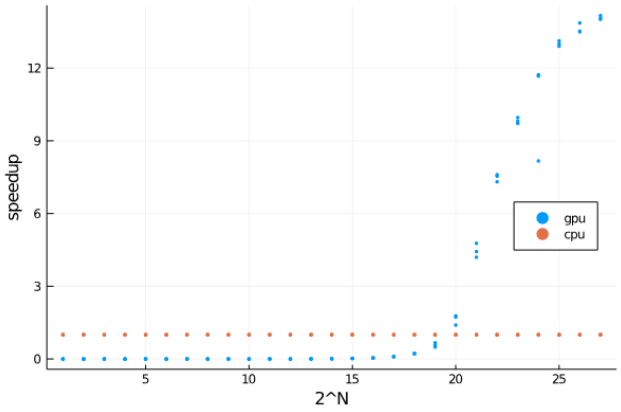

Figure-2: GPU speedup over CPU for reduce operation

Unlike the map example both the final speedup for large arrays is less along with the point where the GPU overtakes the CPU 2^{20} here vs 2^{17} earlier. This is understandable as unlike the last example there is a level of dependency among all elements and the GPU kernel has points of synchronization which hits the throughput of this operation.

Given `p` processors and array length of `n` the complexity of parallel reduce is O(n/p + log(N)).

It should also be noted that IEEE Floats are not associative which means that for two floating points `x` and `y` there is no guarantee that x + y = y + x. Hence for a lot of reduction operations expecting the answer to be exactly same cannot be guaranteed due to the way floating point math works which has a lot of repercussions on GPU code since the result depends upon the exact order of events which cannot be guaranteed in a parallel environment. A lot of nuance of floating-point math can be found in Goldberg's paper [3].

As a final example lets return to equation (1) where we do the map and the reduction too which is exposed via the mapreduce function which combines both the calls.

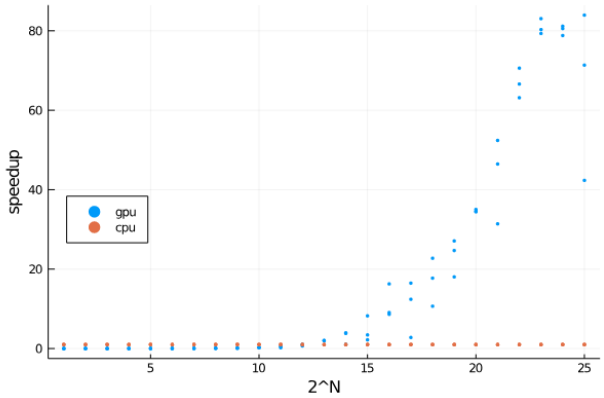

Figure-3: GPU speedup over CPU for map-reduce operation

The benchmark provides sufficient empirical evidence of how GPU is truly dominant over the CPU, even giving around an 80x speedup as the array length grows.

2.1.3. Filter

Filter is a method which only selects those elements from a container which satisfy a predicate equation.

There are two types of filter operations: stable and unstable. Stable filter method means to maintain the original ordering of the elements and unstable filter means that the ordering may be changed in the output container. On a GPU unstable filter is much faster due to no need for synchronizations. Stable filtering requires a level of synchronization hence, we tried to benchmark this worst case against the CPU. We made our test predicate even worse by testing for odd integers which causes massive warp-divergence resulting in 50% efficiency.

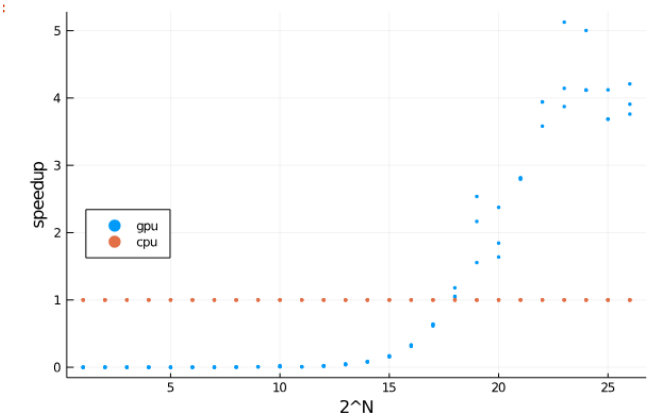

Figure-3: GPU speedup over CPU for filter operation

3. CONCLUSIONS

We have gone through the three fundamental operations of big-data analytics which are map, reduce and filter. The experiments show very convincingly the acceleration achieved with the help of a general-purpose GPU.

REFERENCES

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B Shah, ``Julia: A Fast Dynamic Language for Technical Computing,'' SIAM review, Volume 59, Page(s): 65 – 98

[2] Tim Besard, Christophe Foket, Bjorn De Sutter, ``Extensible Programming: Unleashing Julia on GPUs,'' IEEE Transactions on Parallel and Distributed Systems, Volume: 30 , Issue: 4 , Page(s): 827 - 841

[3] D Goldberg, ``What every computer scientist should know about floating-point arithmetic,'' ACM Computing Surveys (CSUR), 1991