Burrow Wheeler Transform with CUDA, Numba, and Python

(Team BWT)

Rajeev Thundiyil, Juan Castellon,Vincent Trejo

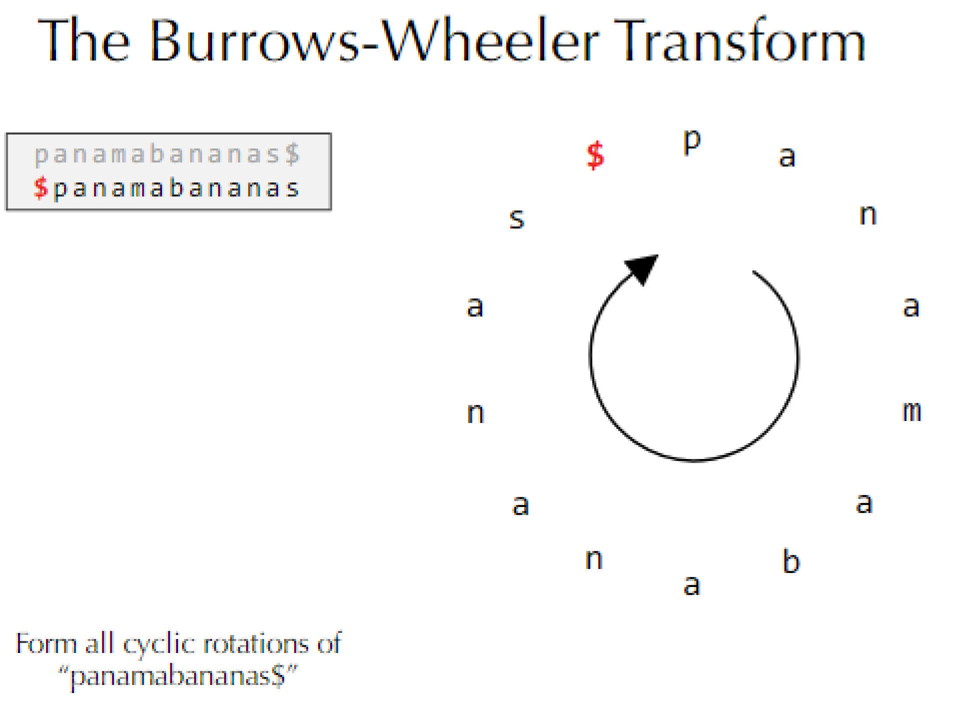
<u>Project Idea / Overview</u>

Burrows-Wheeler Transform is a lossless compression algorithm that shrinks down long strings into a much smaller string, useful in bioinformatics for storing large genomes (the inspiration and primary purpose of this project).

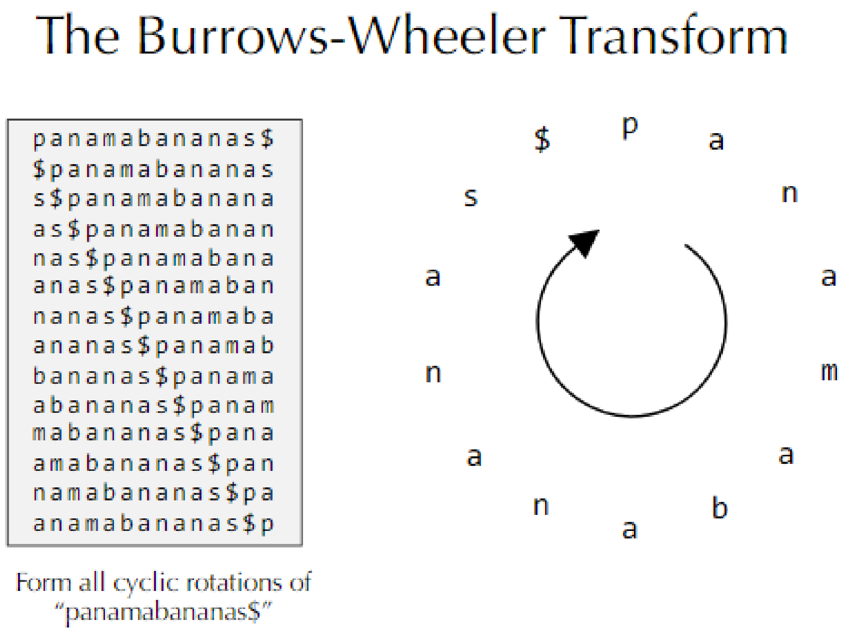
This project studies how the running time is affected by its implementation on the CPU using Python, CUDA, and Numba using the GPU.

<u>How Does it Work ?</u>

The Burrows-Wheeler Transform is a compression algorithm that takes a string and creates a "matrix" of it's cyclic rotations, like so:


The Burrows-Wheeler Transform
panamabananas$
$panamabananas
$ p a n a m a b a n a n a s
Form all cyclic rotations of "panamabananas$"

In the end, the matrix will look like this:

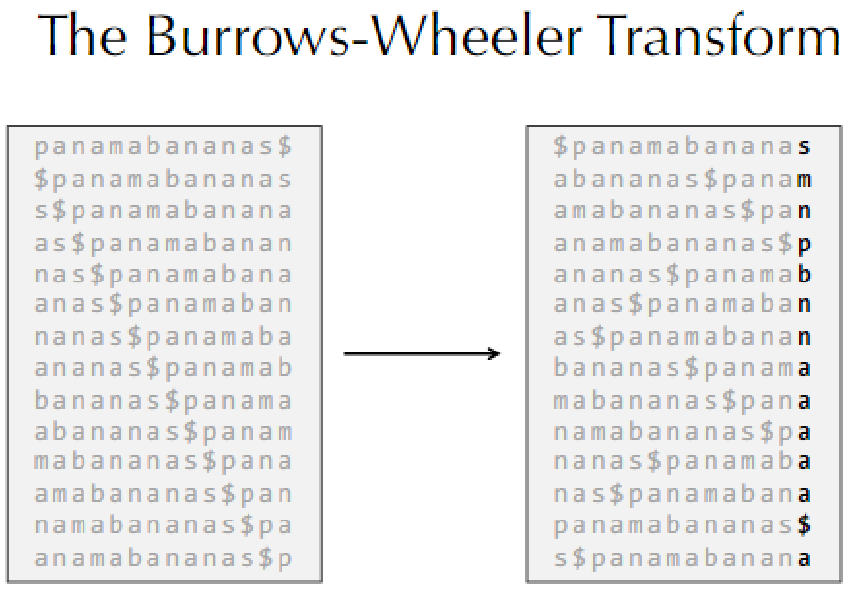


The Burrows-Wheeler Transform

```
panamabananas$
$panamabananas
s$panamabanana
as$panamabanan
nas$panamabana
anas$panamaban
nanas$panamaba
ananas$panamab
bananas$panama
abananas$panam
mabananas$pana
amabananas$pan
namabananas$pa
anamabananas$p
```

Form all cyclic rotations of "panamabananas$"

With this new list of all the cyclic rotations of our original string, we then sort it lexicographically and grab the last character of every string in the list and append them all together like this:

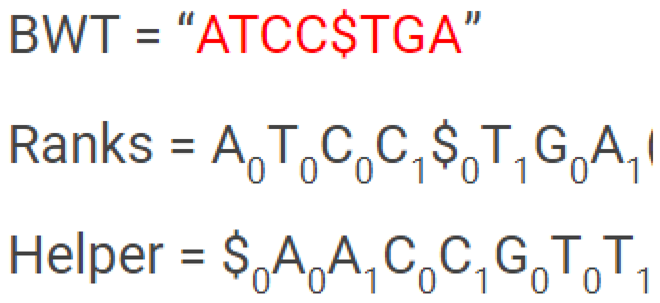


Form all cyclic rotations of *Text* = "panamabananas$"

BWT(*Text*) = last column of *M*(*text*)= "smnpbnnaaaaa$a".

This string is now the Burrows-Wheeler Transform of our original string. For this small example, the size reduction may look negligible, but in genome sizes with tens of millions of base pairs, the reduction is many times smaller than the original size with the irreplaceable benefit of being lossless (which makes it so suited for bioinformatics).

In order to go backwards in order to get the original string, we implemented a method using a “rank” and a “helper” list. The helper list is a list of all the characters in the Burrows-Wheeler Transform of the string and the number of times they’ve appeared in the string thus far. The rank list is derived by sorting the characters of the Burrows-Wheeler Transform of the string and then adding them to the list along with the number of times they’ve appeared in the string thus far:



BWT = “ATCC$TGA”

Ranks = $A_0T_0C_0C_1\$_0T_1G_0A_1$

Helper = $\$_0A_0A_1C_0C_1G_0T_0T_1$

We then start at position 0 of the helper and find the index at which ($,0) appears in the ranks list. We then take that index and go to that place in the helper list and take the letter of that pair (index = 5, current output is C), append that to our text. Loop this logic over the length of the text and you will traverse all elements in the lists and will you have your final answer (CCATGTA$).

How is the GPU used to accelerate the application?

The problem at the core of the Burrows-Wheeler-Transform is the sort required to get the lexicographically sorted list of rotations. Parallelizing an array of strings was difficult to get a handle on, but we found some resources that pointed towards using a Bitonic Sort. This sort uses

Implementation details

Documentation on how to run your code

Instruction steps:

- C++ Implementation steps
    - Open on bender environment
    - Run **singularity shell --nv /singularity/cs217/cs217.sif**
    - Run **g++ bwt.cpp**
    - Run **a.out** executable

Explanations on what gets shown

- C++
    - The C++ implementation shows the rotations step by step of the process done through the BWT algorithm. Once that is done, a final string is outputted to show what it looks like after executing the functions. At the very bottom is a line showing the runtime of the algorithm with the string it uses.

- Python/Numba Implementation steps
    - Open on bender environment
    - Run **singularity shell --nv /singularity/cs217/cs217.sif**
    - Run **python bwt.py**

Explanations on what gets shown

- Python/Numba
    - The algorithm outputs the original text, the Burrows-Wheeler-Transform of the text, and the reverse of the transform.

Evaluation/Results

C++ implementation times

The implementation tests had tests of a 100 character string, 1000 character string, and a string at its max size with 2971 characters. Using #include <chrono> , a clock was implemented into the code to time each time the function was called.

Character #	Time (microseconds)
100	44,279 microseconds
1000	2,078,724 microseconds
2971 (MAX SIZE)	5,998,840 microseconds

Problems faced

We knew C++ would be harder to implement compared to Python since Python would be easier to code with lists instead of C arrays/vectors. After some struggles in trying to find out how to implement this for C++, we managed to come up with the code that can run this algorithm properly without issues. However, we had issues with parallelizing through CUDA. However, we did know what we had to do in order to do it.

Our data needed to be sorted lexicographically (from the matrix of rotations). With some research, we found that a bitonic sort is what we needed to parallelize our data to use with CUDA. This sorting algorithm has a runtime of O($nlog^2(n)$) with its comparisons. It only works if the amount of elements is at a total of 2^n.

On the last page, include a table with a list of tasks for the project, and a percentage breakdown of contribution of each team member for the tasks. You can choose the granularity of task breakdown here.

Tasks	C++	Numba	Python	Cuda	Report	Video
Rajeev	33	33	33	33	33	33
Vincent	33	33	33	33	33	33
Juan	33	33	33	33	33	33

Everyone was involved in calls and in attempts to code the implementations.

Link for the video:
https://ucr.yuja.com/V/Video?v=5067233&node=17616392&a=1635452213&autoplay=1