Spotify Visualizer

How to Run

Note: We defaulted the setup on this repository to validate each implementation against the serial one. Because of the need for *random* initialization in the k clusters, and the variable number of epochs and clusters, we run the serial and then execute whatever function you are using. For example, when running the single GPU implementation, we execute the serial implementation followed by the GPU implementation, so that the data can be validated.

Serial CPU

To run just the serial implementation:

```
g++ serial-only.cpp -o serial
./serial 25 6
```

To change the number of epochs and clusters, pass in different values as arguments to the command. `./serial <number of epochs> <number of clusters>`

Parallel CPU

```
g++ -fopenmp serial-to-parallel.cpp -o parallel
./parallel 25 6 8
```

To change the number of epochs, clusters and threads, pass in different values as arguments to the command. `./parallel <number of epochs> <number of clusters> <optional: number of threads>`

If you do not pass in a threadcount, it will default to 4 threads.

Distributed CPU

Running on CHPC first we need to load the module:

```
module load openmpi
```

When running this, you need to pass in two arguments: the number of epochs and the number of clusters. For example, to run 25 epochs with 6 clusters:

```
mpic++ serial-to-distributed-cpu.cpp -o distributed
mpirun -np 2 ./distributed 25 6
```

You need to specify the number of nodes you want to run on with the `-np` flag. This number must be less than or equal to the number of nodes you have access to.

To change the number of epochs and clusters, pass in different values as arguments to the command.

`mpirun -np <number of nodes> ./distributed <number of epochs> <number of clusters>`

Parallel GPU

Running on CHPC first we need to load the module:

```
module load cuda/12
```

Now we can compile and execute:

```
nvcc serial-to-single-gpu.cu -o gpu
./gpu 25 6
```

To change the number of epochs and clusters, pass in different values as arguments to the command.
`./gpu <number of epochs> <number of clusters>`

Distributed GPU

First we setup our CHPC environment to have N nodes and each node to have a GPU. Next, we load our modules:

```
module load cuda/12
module load openmpi
```

Now we can compile and execute, but we need to compile the files separately and then join them together:

Path to where CUDA is installed

```
echo $CUDA_PATH # gets the path to where cuda is
```

Your path could be different, but it will be similar to this on our CHPC system:

```
/uufs/chpc.utah.edu/sys/installdir/r8/cuda/12.2.0/
```

Then we append `lib64` to the end of that path:

```
/uufs/chpc.utah.edu/sys/installdir/r8/cuda/12.2.0/lib64
```

Now we can compile and execute:

```
nvcc -c distributedGPU.cu
mpicxx -o dist serial-to-distributed-gpu.cpp distributedGPU.o -
L/uufs/chpc.utah.edu/sys/installdir/r8/cuda/12.2.0/lib64 -lcudart -lcuda
mpirun -np 2 ./dist 25 6
```

To change the number of epochs and clusters, pass in different values as arguments to the command.

`mpirun -np <number of nodes> ./distributed <number of epochs> <number of clusters>`

Running the Python Visualization

First, edit the file `visualize.py` to point to the correct csv file you would like to visualize.

Now, run the following commands from the project root directory:

```
pip install -r requirements.txt
python3 visualize.py
```

Validation

In serial.hpp we wrote a function, `areFilesEqual`, to validate two csv files against eachother. It will return true if the are, false if not. We will check every file against the ground truth, defined by the serial implementation. If we pass in the optinal variable true after the thread names, it will print out the first 5 differences between the files and print the total number of differences.

To simplify grading and validation, we built into every implementation a function call to `performSerial` this allows the specific number of epochs and clusters to be run serially, and in the implementation. We also wrote the function `areFilesEqual` to compare the output of the serial implementation to the output of the implementation being tested. As a grader you only need to run the program and validation will be done for you.

Our Approaches

We chose to use 3,4, and 6 as our testing values for the clusters because we wanted to see how well the program would change going in doubling the cluster size (3 to 6), and also to see how only changing one (3 to 4) would affect our outcomes. We ran all of the programs several times to ensure accuracy in timing and outputs.

Serial Implementation

For the serial implementation we used the link provided by Dr. Petruzza here. We added onto this the z variable into both the Point3D value and how we update the clusters at the end of each epoch. We implemented the actual serial code in `serial.cpp` and the commonly used code across all implementations in `serial.hpp`. Fundamentally, this algorithm works by taking in X number of points and K randomized clusters. Every point is then assigned to a cluster based on how close it is. The clusters are then updated to reflect their nearest points, the average of the groupings, and then we run this entire process for a specified number of iterations, denoted as `epochs` in the code.

Single GPU Implementation

For the GPU implementation we took our serial implementation and changed as little as possible to preserve code modularity, but making the inner loop a GPU kernel. We created a kMeansClustering kernel that takes in the data, clusters, and epochs and performs the kMeansClustering algorithm. We also created an on device function `calculateDistance` to easily calculate the distance between two points.

The most important thing to get right in this implementation was the memory management. We needed to allocate memory for the data, clusters, and distances. We also needed to copy the data and clusters to the device, and copy the clusters back to the host after each epoch. Keeping track of the the thread we were on within the kernel was also important, because we needed to know which cluster to update. The slowest part of this implementation was copying the data and clusters back and forth between the host and device, because we have to do that for each epoch in order to consolidate data and then fix upon completion.

Parallel CPU Implementation

For the parallel CPU implementation we took our serial implementation and changed it to use Open MP. As with the GPU paralleization, the main consideration we had was on the nested loops iterating over every point. This was the loop we parallelized, allowing each thread to use a chunk of the total number of points (1,240,425). In the case of 5 threads, each thread works on 1240425/5 = 248085 points. We let OpenMP handle the distribution of the points across each thread. After each thread is completed, we need to update the clusters, which is done by the master thread, then restart the loop for the next epoch, once again distributing points among the threads. We created one thread region outside of the epochs and then parallelized the algorithm, updated, and ran it `epoch` number of times.

Distributed CPU Implementation

For the distributed CPU implementation we wanted to be able to use as much code as possible from the serial implementation, but MPI required us to pass arrays around rather than points. This required rewriting the shared functions and creating our own distributed version called `calculateKMean` and `updateCentroidDataDistributed`.They take in arrays of clusters, denoted with k's and arrays of the data points. Each process has its own chunk of data it is working on, called recv_x recv_y and recv_z. Each process handles their subset of the points and compares it against the clusters. After each process is done, we need to update the clusters on the master node, restart the loop for the next epoch, and then broadcast the updated clusters to all processes.

To ensure the data could be distributed across any number of processors we used ScatterV to distribute the data and allowed each process to only work on its specified points. We had process 0 handle the serial implementation at the beginning and the comparison of points at the end to validate accuracy.

Distributed GPU Implementation

For the distributed GPU implementation, we took the code from our distributed CPU implementation and created the `distributedGPU.cu` file to run the GPU code. We wrote the `launchCalculateMean` function to launch the kernel and to also be accessible to outside files (needed for a distributed GPU). Everything else in the distributed GPU is the same as distributed CPU, besides that specific function call.

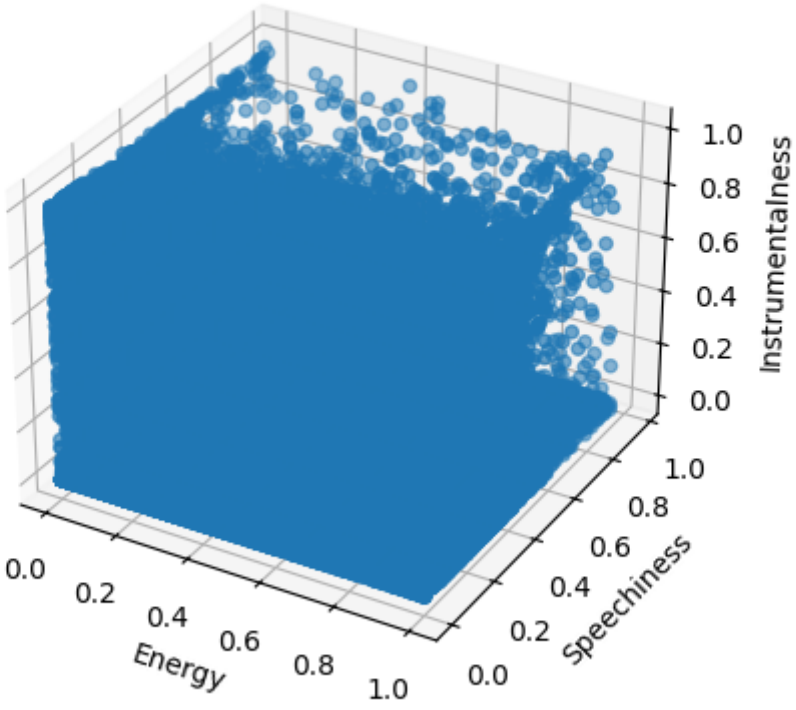
When we are running `launchCalculateMean` we allocate memory for the particular node's data points and the centroids. It runs the k-means clustering algorithm on the GPU, then copies the updated associations which are on teh reference from the GPU to the CPU. At this point we are back into the distributed system and use gatherV to bring these updated centroids and their associated points together. This whole process is repeated for each epoch.

We had trouble initially with the distributed GPU implementation because we were not allocating enough memory on the GPU to complete its required tasks. We diagnosed the issue using our `checkCUDAError` function and found that we were running out of memory on the GPU for our specific data points. We realized we had been using mismatched points, and after fixing that we were able to run the distributed GPU implementation.

Analysis

In the data set there are 1240425 points, and the amount of data processed is equal to epochs * numPoints. So for 100 epochs, we process 124042500 points. For 200 epochs, we process all of those points 200 times (248085000), etc.

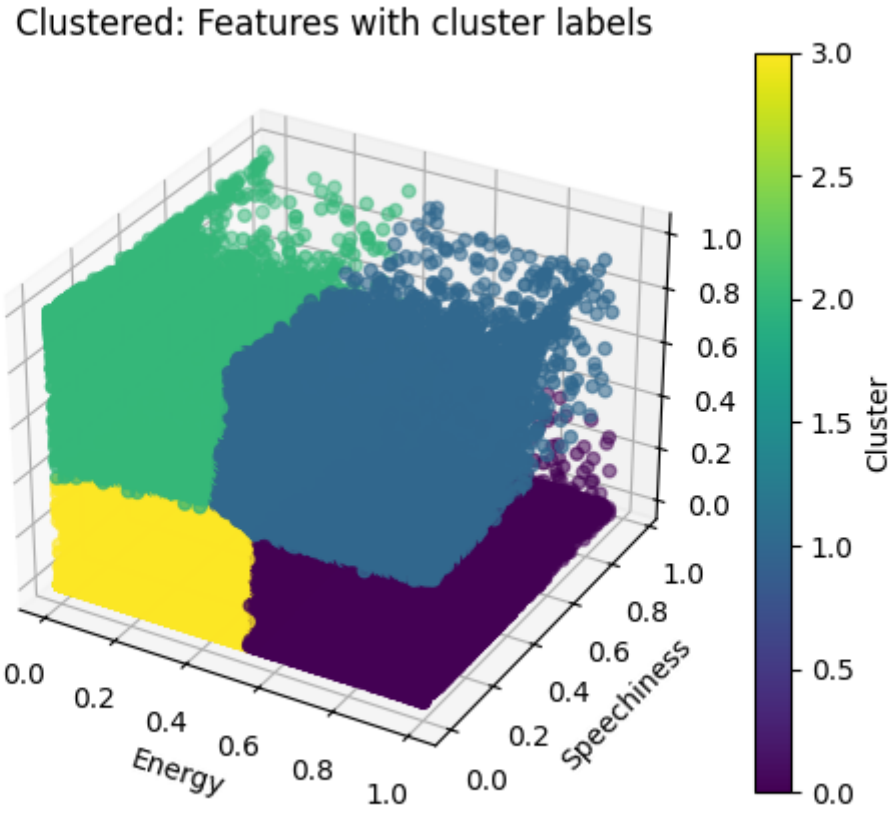
Unprocessed Data



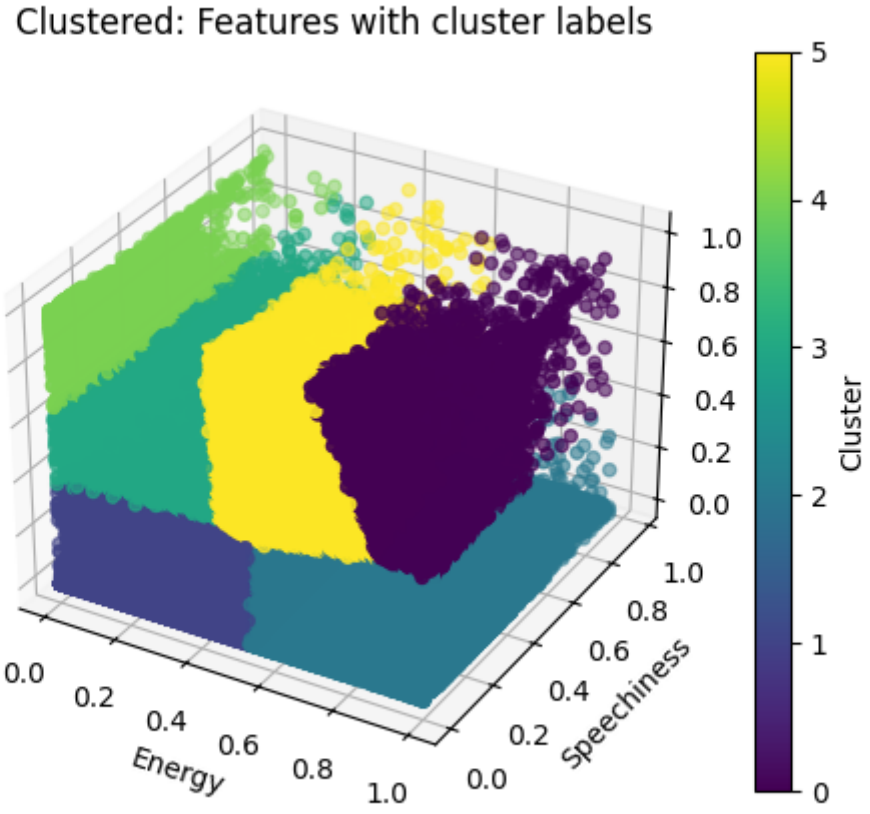
Serial Implementation

Time (s)	Epochs	Clusters
9.807173	50	3
12.552126	50	4
17.976485	50	6
19.570529	100	3
25.194927	100	4
35.975967	100	6
38.853329	200	3
50.216369	200	4
72.608509	200	6

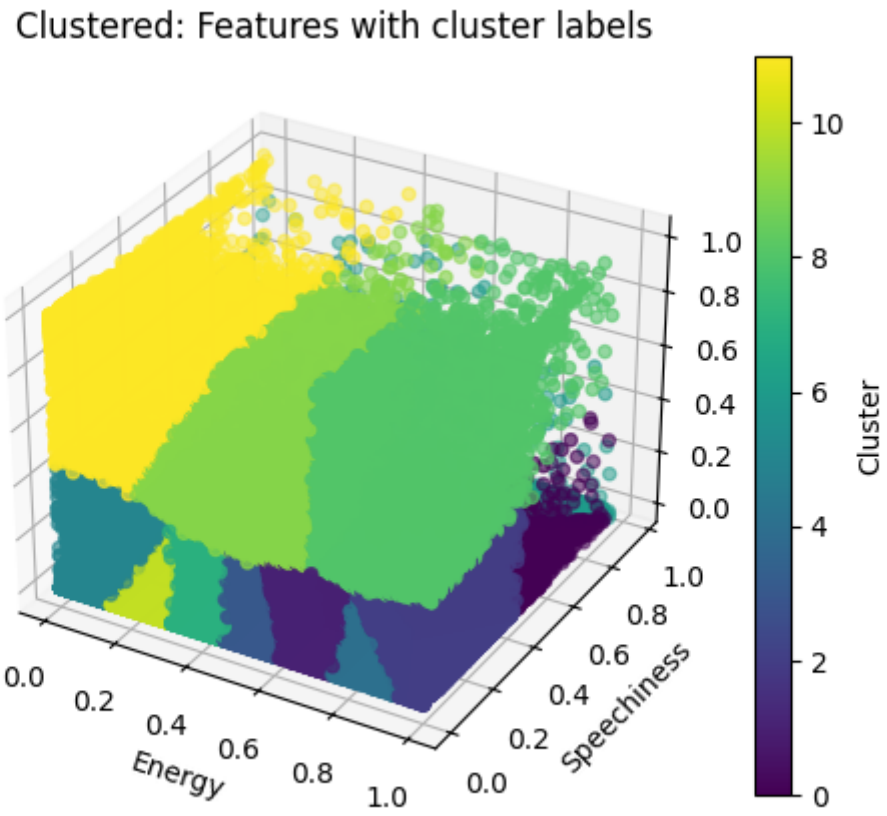
Output for 4 Clusters:



Output for 6 Clusters:



Output for 12 Clusters:



Parallel CPU Implementation

The following table represents the data for the parallel CPU implementation with 100 epochs. The number of clusters is variable, as is the number of threads. The speedup is calculated by dividing the time serial by the time parallel. The efficiency is calculated by dividing the speedup by the number of threads.

Threads	Time Parallel (s)	Time Serial (s)	Epochs	Clusters	Speedup	Efficiency
4	10.889288	19.551173	100	3	1.79545	0.44886
8	8.705353	19.522629	100	3	2.24260	0.28033
16	8.833985	19.614534	100	3	2.22035	0.13877
4	12.073964	25.194927	100	4	2.08671	0.52168
8	9.659430	25.156176	100	4	2.60431	0.32554
16	10.108751	25.094426	100	4	2.48245	0.15515
4	20.276805	36.017955	100	6	1.77631	0.44408
8	12.101930	35.999700	100	6	2.97471	0.37184
16	12.805487	36.019349	100	6	2.81281	0.17580

As you can see from the above table, The time it takes to run the parallel implementation is less than the time it takes to run the serial implementation. However, this is not strongly scalable, since the efficiency goes down as we increase the number of threads.

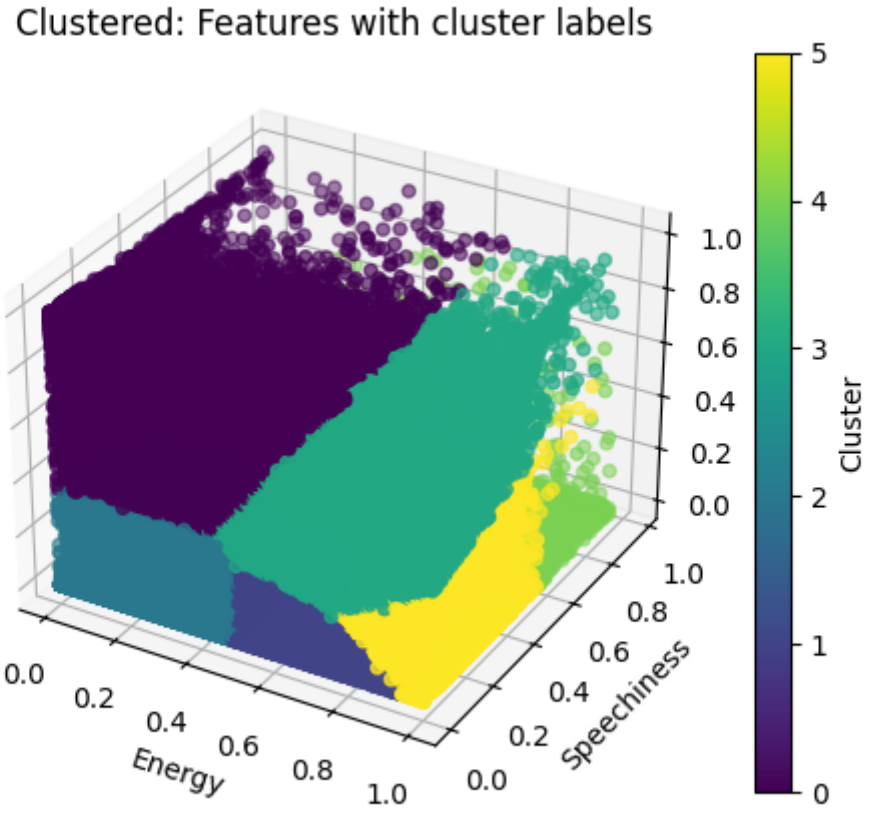
The following table represents the data for the parallel CPU implementation with the number of epochs increasing at a proportional rate to the number of threads. Data is collected for 3, 4, and 6 clusters. Again, the speedup is calculated by dividing the time serial by the time parallel. The efficiency is calculated by dividing the speedup by the number of threads.

Threads	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup	Efficiency
4	6.825731	9.807173	50	3	1.43679	0.35920
8	8.923136	19.570529	100	3	2.19323	0.27415
16	17.886764	38.853329	200	3	2.17218	0.13576
4	7.926284	12.552126	50	4	1.58361	0.39590
8	9.594114	25.115965	100	4	2.61785	0.32723
16	19.675230	50.216369	200	4	2.55226	0.15952
4	10.002409	17.976485	50	6	1.79722	0.44930
8	11.812873	35.975967	100	6	3.04549	0.38069
16	25.851216	72.608509	200	6	2.80871	0.17554

The above table again shows that the parallel implementation is faster than the serial implementation. However, it is not weakly scalable, since the efficiency goes down, even as we increase the number of epochs along with the number of threads.

Unfortunatly, both tables show that the parallel implementation is not scalable. However, the parallel implementation is still 1.5-2.5x faster than the serial implementation.

Parallel CPU Implementation Visualized with 6 Clusters:



Single GPU Implementation

Device Details from Cuda Query

We are looking at the following GPU:

```
Device 0: "NVIDIA GeForce RTX 3090"
  Major revision number:                         8
  Minor revision number:                         6
  Total amount of global memory:                 3963289600 bytes
  Number of multiprocessors:                     82
  Number of cores:                               656
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x
65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.70 GHz
  Concurrent copy and execution:                 Yes
```

Becuase it can have 1024 threads per block, we will use that as our baseline. It has 82 multiprocesors, and ideally we hit our maximum number of threads and use all of the multiprocessors.

Experimental Results

The following table represents the data for the single GPU implementation with 50 epochs. The number of clusters is variable, as is the number of threads per block. The speedup is calculated by dividing the time serial by the time parallel.

Threads per Block	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
256	1.726700	9.633755	50	3	5.57929
512	1.678286	9.381626	50	3	5.59000
1024	1.838031	9.616937	50	3	5.23220
256	1.731712	12.514634	50	4	7.22674
512	1.718942	12.332335	50	4	7.17438
1024	1.652933	12.598727	50	4	7.62204
256	1.838919	18.889932	50	6	10.2723
512	1.670835	19.385700	50	6	11.6024
1024	1.717685	18.611738	50	6	10.8354

As you can see from the above table, the time it takes to run the parallel implementation is less than the time it takes to run the serial implementation. It is interesting to note, that while increasing the number of threads to 512 marginally increases the speedup, increasing the number of threads to 1024 actually decreases the speedup in each case.

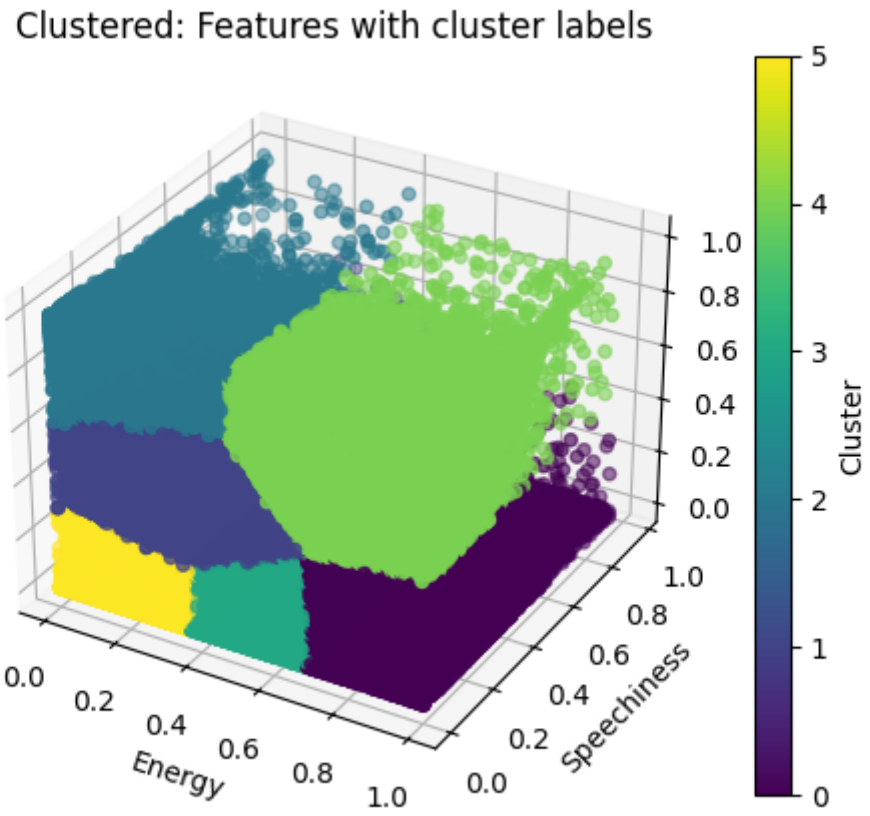
The following table represents the data for the single GPU implementation with the number of epochs increasing at a proportional rate to the number of threads per block. Data is collected for 3, 4, and 6 clusters. Again, the speedup is calculated by dividing the time serial by the time parallel.

Threads per Block	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
256	1.726700	9.633755	50	3	5.57929
512	3.224774	19.231200	100	3	5.96358
1024	6.292316	38.413889	200	3	6.10489
256	1.731712	12.514634	50	4	7.22674
512	3.190473	24.911140	100	4	7.80798
1024	6.146267	50.652925	200	4	8.24125
256	1.838919	18.889932	50	6	10.2723
512	3.311074	37.237659	100	6	11.2464

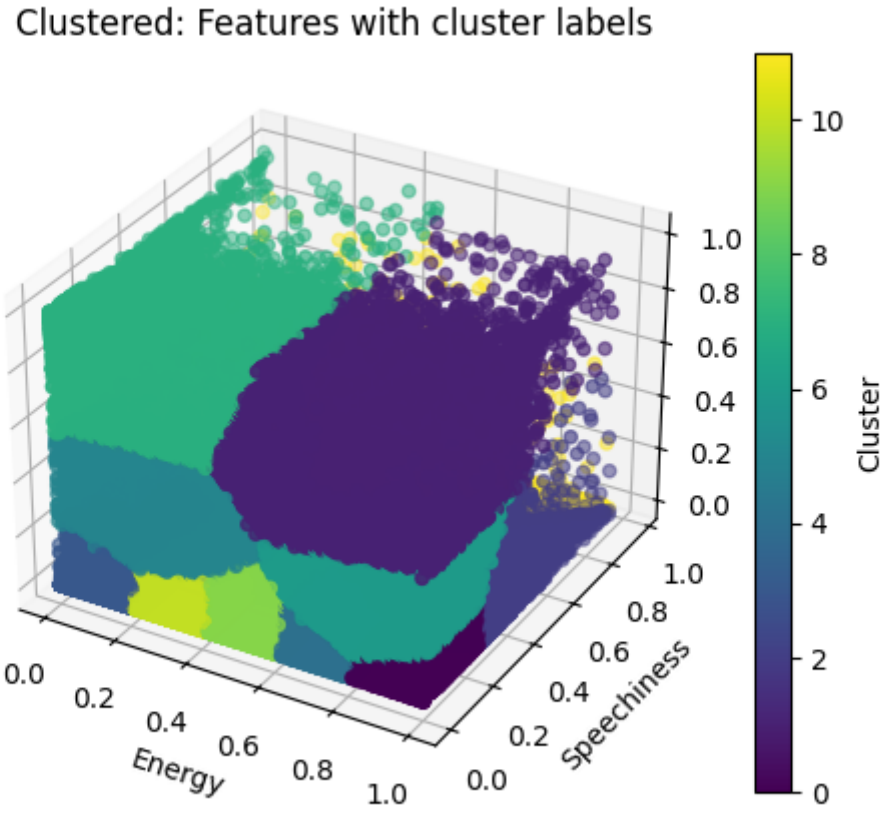
Threads per Block	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
1024	6.135140	75.330369	200	6	12.2785

As the number of epochs is increased with the number of threads per block, the speedup increases in each case, including the increase from 512 to 1024 threads per block. It is also worth noting, that the speedup increases as the number of clusters increases. The speedup for these values are much higher than the parallel CPU speedups (2.5x versus GPU's 5-10x)

Single GPU Implementation Visualized with 6 Clusters:



Single GPU Implementation Visualized with 12 Clusters:



Distributed CPU Implementation

On 3 clusters:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
2	0.766036	4.811508	25	3	6.28104
3	0.621373	4.827385	25	3	7.76890
4	0.581396	4.842869	25	3	8.32972

On 4 clusters:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
2	0.947897	6.203727	25	4	6.54472
3	0.739652	6.202104	25	4	8.38516
4	0.670900	6.228862	25	4	9.28433

On 6 Clusters:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
2	1.245606	8.818218	25	6	7.07946

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
3	0.942043	8.863819	25	6	9.40914
4	0.819143	8.859402	25	6	10.8154

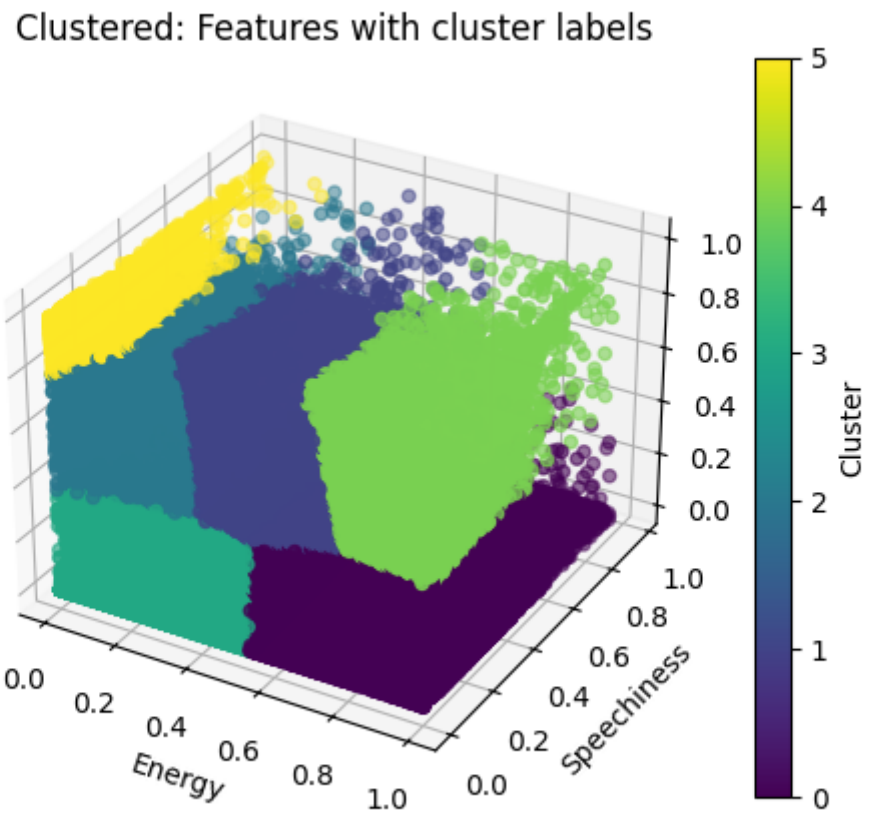
Notice how the parallel time is going down as we increase the number of nodes in the above tables. Increasing the number of nodes breaks up the amount of data to process per node and allows us to process the data faster. It is also worth noting, that increasing the number of clusters increases the speedup. This is because the amount of data per cluster is decreased, and thus the amount of data per node is decreased.

Increasing the number of epochs also increases the speedup, as shown in the following table:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Speedup
4	0.820076	8.816901	25	6	10.7513
4	1.595893	17.698479	50	6	11.0900
4	3.121529	35.421519	100	6	11.3474
4	6.180246	70.824262	200	6	11.4597

The above table shows that as we increase the number of epochs, the time increases proportionally. This is to be expected. The speedup also increases as the amount of data increases, because there is less overhead and each node gets more data to work on.

A visualized example of the distributed CPU implementation with 4 nodes and 50 epochs:



Distributed GPU Implementation

On 3 Clusters:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Threads per Block	Speedup
2	0.707328	4.711447	25	3	256	6.66090
3	0.647141	4.714513	25	3	256	7.28514
4	0.799928	4.704620	25	3	256	5.88130

On 4 Clusters:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Threads per Block	Speedup
2	0.634951	6.065612	25	4	256	9.55288
3	0.814948	6.070404	25	4	256	7.44882
4	0.804700	6.096105	25	4	256	7.57562

On 6 Clusters:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Threads per Block	Speedup
2	0.778927	8.686541	25	6	256	11.1519

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Threads per Block	Speedup
3	0.657219	8.690562	25	6	256	13.2232
4	0.821848	8.679386	25	6	256	10.5608

It is interesting to note that while the speedup usually increases as we increase the number of nodes from 2 to 3, it decreases when we increase the number of nodes from 3 to 4. This is likely due to the fact that we are using a distributed GPU, and the overhead of copying data to and from the GPU is slowing down the process.

However, increasing the number of epochs increases the speedup in a similar behavior to the distributed CPU implementation, as shown in the following table:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Threads per Block	Speedup
4	0.821734	8.690958	25	6	256	10.5763
4	1.296684	17.392156	50	6	256	13.4127
4	2.257343	34.808612	100	6	256	15.4201
4	4.277128	69.621106	200	6	256	16.2775

On this table, we see that the increasing epochs does increase the time, and doubling the amount of data very nearly doubles the time. This is to be expected. The speedup also increases as the amount of data increases, similar to the distributed CPU implementation.

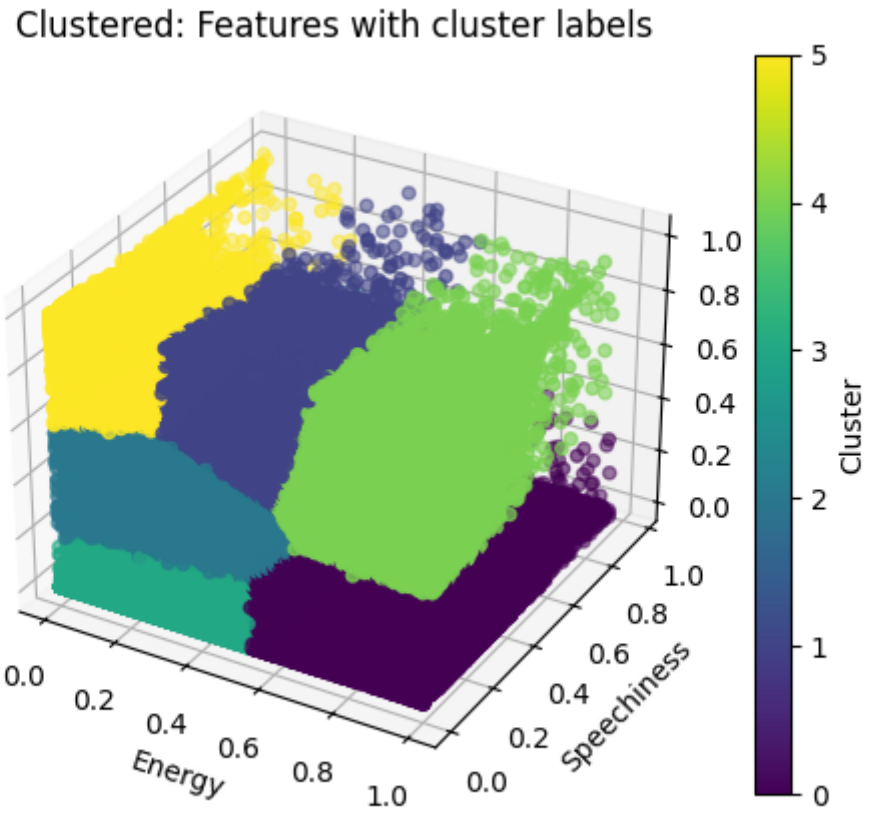
As expected, the distributed GPU implementation is generally faster than the distributed CPU implementation, however, it depends more on an ideal configuration of nodes and threads per block. The following table shows changes in the speedup as we change the number of threads per block:

Nodes	Time (s)	Time Serial (s)	Epochs	Clusters	Threads per Block	Speedup
4	1.400168	17.441074	50	6	64	12.4564
4	1.296684	17.392156	50	6	256	13.4127
4	1.410010	17.333747	50	6	1024	12.2933

As you can see, the speedup is highest when we use 256 threads per block, and decreases when we use 64 or 1024 threads per block. Thus, the ideal configuration for the distributed GPU implementation is 256 threads per block.

The distributed GPU implementation is faster than the distributed CPU implementation, but it is not scalable. The speedup is not consistent as we increase the number of nodes, and the speedup is not consistent as we increase the number of threads per block. For 200 epochs on 4 nodes the GPU implementation is 1.5x faster than the CPU implementation. The GPUs are slowest when they need to copy all of the data from the CPU to the GPU, and then back to the CPU. Using tiling techniques would be reduce that memory bandwith copy time and allow the GPU to be more efficient.

Figure of the distributed GPU implementation with 6 nodes and 100 epochs:



References

- Initial Setup
- K-Means Clustering
- Paper on K-Means
- Distributed GPUs