Data Imputation with an Improved Robust and Sparse Fuzzy K-Means Algorithm

Connor Scully-Allison[†], Rui Wu[§], Sergiu M. Dascalu[†], Lee Barford[†‡], Frederick C Harris, Jr.[†]

[†]*Dept. of Computer Science and Engineering*
University of Nevada, Reno
Reno, NV

[§]*Dept. of Computer Science*
East Carolina University
Greenville, NC

[‡]*Keysight Laboratories*
Keysight Technologies
Reno, NV

cscully-allison@nevada.unr.edu, {dascalus,fred.harris}@cse.unr.edu, wur18@ecu.edu, lee.barford@ieee.org

Abstract—**Missing data may be one of the biggest problems hindering modern research science. It occurs frequently, for various reasons, and slows down crucial data analytics required to answer important questions related to global issues like climate change and water management. The modern answer to this problem of missing data is data imputation. Specifically, data imputation with advanced machine learning techniques. Unfortunately, an approach with demonstrable success for accurate imputation, Fuzzy K-Means Clustering, is famously slow compared to other algorithms. This paper aims to remedy this foible of such a promising method by proposing a Robust and Sparse Fuzzy K-Means algorithm that operates on multiple GPUs. We demonstrate the effectiveness of our implementation with multiple experiments, clustering real environmental sensor data. These experiments show that the our improved multi-GPU implementation is significantly faster than sequential implementations with 185 times speedup over 8 GPUs. Experiments also indicated greater than 300x increase in throughput with 8 GPUs and 95% efficiency with two GPUs compared to one.**

I. Introduction

With any dataset collected in large volumes and at high velocities, missing data will occur. For the Nevada Research Data Center(NRDC), a Nevada-based data management center, the occurrence of missing data can be frustrating.[1] Data holes hinder data analytics by reducing usable data. Oftentimes, entire rows of downloaded measurements are thrown out by data users to simplify pre-processing. Any loss or lack of usable data slows down research being done by various institutions on issues like water conservation and global warming. This, in turn, reduces the production of actionable intelligence which can be used to drive institutional policies on these matters.

Myriad phenomena can poke holes in otherwise valid datasets, and can result in long sequential strings of missing data. When a large series of data points goes missing, it makes data repair that much harder than if scattered holes were found. Specifically, sequences of missing data preclude the use of intuitive varieties of data imputation, like interpolation. With interpolation, a singular hole in time series data can be filled by finding the mean between the preceding and following data points and inserting that value. While not foolproof, methods like this can be simple and effective, however, they start to fail as the gaps between known data points get wider.

To surmount this problem more advanced imputation techniques are required. Many approaches exist to accurately fill data holes, however one of the most effective techniques comes from the domain unsupervised machine learning: clustering, specifically Fuzzy K-Means (FKM) clustering. [2] By using multivariate data from other sites and sensors, a missing value can be assigned into multiple clusters with similar values according to the similarity of other values, or “features”, collected at the missing timestamp. Even in the occurrence of large strings of missing data points, accurate clustering is still possible and provides sufficient contextual information for imputing data.

This method comes with a downside however. Fuzzy K Means clustering is a notoriously slow algorithm, especially when compared with other common imputation methods [2]. FKM often requires hundreds of iterations of tens of thousands of euclidean distance comparisons. If naively implemented, this algorithm can take hours to separate 40,000 vectors of floating point measurements into meaningful groups.

To address this, we propose to leverage the power of GPUs to enable the massive concurrent processing of embarrassingly parallel distance and optimization calculations used by FKM Clustering. Specifically, this paper proposes modifications to a specific Fuzzy K Means algorithm, called “Robust and Sparse Fuzzy K Means”. This algorithm ensures that robust and accurate clustering occurs with any number of GPUs. This paper shows that our proposed *improved Robust and Sparse Fuzzy K-Means* (iRSFKM) algorithm provides accuracy results sufficient for the imputation needs of the NRDC with up to 180x speedup over an optimized implementation of the original.

The remainder of this paper is organized as follows: Section II describes the background of data imputation, FKM clustering, and the use of GPUs in these domains. Section II also describes in more significant detail the background required to understand the RSFKM algorithm which this paper modifies. Section III describes the details of implementing and adapting the RSFKM algorithm to one GPU and multiple GPUs. Section IV outlines the experiments run and results which validate the iRSFKM implementation on one GPU and multiple GPUs. Section V concludes the paper and outlines avenues and opportunities for future research based on this work.

II. Background Related Works

"Missing data imputation" describes the process of filling in holes that occur in sufficiently large data sets [3]. To ensure valid statistical analyses with a dataset these holes must be filled with accurate estimations of "ground truth" values. Although many simple and effective statistical imputation methods exist, like K Nearest Neighbors which takes a simple mean of "closely related" data points [4], not every dataset is as sympathetic to these approaches as others. Accordingly, when choosing approaches to data imputation, multiple factors must be considered. The randomness of missing data and the structure of the dataset being imputed are a few examples of this.

The work of Schmitt et. al. explores these factors in detail with a comparative analysis of six common methods of data imputation [2]. In this paper, the authors compare the imputation methods of a simple Mean, KNN, fuzzy K-means, singular value decomposition (SVD), Bayesian principal component analysis (bPCA) and multiple imputations by chained equations (MICE). Using a quantitative analysis gauging the accuracy of imputed data on multiple benchmark data sets, the authors largely concluded that – for both large and small datasets – FKM provided accurate results at the cost of a very poor execution time. This trade-off indicates a clear opportunity for optimization with the aid of modern GPU programming techniques.

Fuzzy K-Means clustering algorithms are modifications of traditional K-Means clustering. These algorithms exploit fuzzy set theory to define membership as a percentage allowing for a data point to have membership in multiple clusters [5]. This approach to clustering has proven to be very effective for applications in the domains of computer vision and pattern recognition [6], [7]. Data imputation as an application of FKM has been explored by many authors [8], [9], but in the recent past, "pure" FKM approached to Data Imputation fallen out of vogue, yielding to hybrid approaches with other machine learning techniques [10], [11]. Accordingly, there currently exists a research gap in the area of utilizing a standalone FKM algorithm for Data Imputation.

Finally, the concept of leveraging GPUs to accelerate the time consuming process of clustering is not a new one. A handful of publications have been produced exploring how best GPUs can be leveraged for clustering. However, many of these publications are now outdated [12], [13], having been released very shortly after the introduction of NVIDIA's GPU processing framework, CUDA. Accordingly, they do not reflect significant changes to GPU hardware and software. Changes which certainly affect the structure of proposed algorithms and implementations. Additionally, more prominent contemporary articles do not consider applications of data imputation and, more importantly, do not create Multi-GPU implementations of their algorithms [14]. We assert that the inclusion of Multiple GPUs is a substantial contribution of this paper over prior work.

A. Fuzzy K Means

The traditional approach to Fuzzy K-Means, as introduced by Dunn[5], is relatively simple. A set **X** of *n* objects can be grouped into *c* clusters with membership coefficients **U** defined by centroids in matrix **V** with the following algorithm:

```
while Not Converge do
    Compute centroids V via (1)
    Compute coefficients of memberships for U via (2)
end while
```

The equations referenced in the above algorithm follow here with a brief explination:

$$v_k = \frac{\sum_x w_k(x)^m x}{\sum_x w_k(x)^m} \tag{1}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{c} u_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{\|x_i - v_j\|}{\|x_i - v_k\|} \right)^{\frac{2}{m-1}}} \tag{2}$$

u_{ij} in the above equation describes a specific cell of membership matrix **U** which defines the membership of object x_i in centroid v_j.

B. Robust and Sparse Fuzzy K-Means

The fundamental algorithm modified and used in this work is derived from Xu et. al. [15]. This algorithm, called "Robust and Sparse Fuzzy K-Means Clustering," makes several adjustments to traditional Fuzzy K Means Algorithms to enforce "robustness" and "sparsity" of the clustering. In this context, "robustness" refers to a characteristic of centroids which mitigates outlier influence in updating their mean value. "Sparsity" refers to a cluster's membership characteristics. A membership vector is sparse if a data point is wholly a member of only one cluster and no others. It is the authors' assertion that there exists an optimal sparsity for each data point where it belongs to a few clusters but not others.

The algorithm proposed by Xu et. al. modifies the core FKM algorithm expressed above with the modification of equations (1) and (2). Several supplementary equations were also introduced by the authors of this paper to enforce robustness and sparsity. The following equations replaces (2) and (1), respectively:

$$\underset{\substack{u^i 1=1,\\ u^i \geq 0}}{min} \|u^i - \widetilde{h}^i\|_2^2 \tag{3}$$

This equation optimizes membership for a value denoted by row u_i, where $1 \leq i \leq n$, and n is the number of rows in our data matrix. Equation (3) is a wholly independent sub problem for each line. So with *n* values this minimization can be performed entirely in parallel with *n* cores. $\widetilde{h}^i$ is an auxiliary variable used to enforce sparsity that is stored in an auxiliary **H** matrix. The next equation updates the centroids and replaces (1):

$$v_k = \frac{\sum_{i=1}^{n} s_{ik} u_{ik} x_i}{\sum_{i=1}^{n} s_{ik} u_{ik}} \quad (4)$$

s_{ik} in this equation is an auxiliary variable which enforces robustness. If a euclidean distance between a centroid and data vector is within a user defined threshold then s_{ik} will be defined as the reciprocal of that distance, which reduces the weight of farther points. If its outside the threshold then s_{ik} will be defined as 0. This prevents outliers from influencing centroid updates. u_{ik} describes the membership percentage of data vector i in cluster k, and x_i describes data vector i out of n values being clustered. As many calculations in this algorithm require numerous auxiliary matrices and sum reductions plentiful opportunities for parallelization could be found.

C. Imputation Method

Data imputation with Fuzzy K-Means algorithms is relatively simplistic. After all data objects have been clustered, a missing data value *j* for a specific data vector x_i can be filled in with the following equation [8]:

$$x_{i,j} = \sum_{k=1}^{V} U(x_i, v_k) * v_{k,j} \quad (5)$$

To explain this formula in greater detail, $U(x_i, v_k)$ describes a membership value for a particular vector x_i in a cluster v_k. This value between, 0 and 1, can be used as a weight to scale how much a given centroid k should influence the sum which yields our missing value. The sum itself is the summation of the value at feature j in each centroid. With the weights, this should accurately place our missing value between all centroids of which it has some membership, this should also accurately reflect it's true value.

III. Methods

Four iterations of the improved RSFKM algorithm were implemented for experimentation and analysis. First, a standard CPU implementation of the RSFKM was implemented using a generic solver for equation (3). This implementation was developed to provide a clear baseline of the existing algorithm for timing purposes. Next, the sequential CPU-only implementation was optimized with the introduction of library free convex optimizer in lieu of the generic solver. From there, a single GPU implementation was introduced and optimized for speed and overhead efficiency. Finally, to account for restrictions on GPU memory and produce more consistent results, a multi-GPU iteration of this software was implemented.

A. Convex Optimization

In the work by Xu et. al., the authors reference the use of ”the technique” utilized by Huang et. al. [16] to solve (3). To speed up development time for this baseline code and to promote reproducibility among computer scientists who are not intimately familiar with convex optimization, we diverged from the method in Xu et. al. and utilized ECOS convex optimization solver[17] through the CVXpy interface [18]. Unfortunately, the introduction of this generic solver cased problems for plans of a GPU adaptation of this algorithm.

Presently, no generic solver currently exists which leverages the power of GPUs to speed up its numerous and dense calculations. Since preliminary timings of this solver indicated that this was a substantial bottleneck in the RSFKM algorithm, a workaround was necessary to enable the use of GPU architectures with RSFKM. To solve this problem, a library free solver, comprised of optimized C code generated by CVXGEN [19], was integrated into a modified sequential implementation of RSFKM. This improved implementation was significantly faster and provides a best case baseline to measure GPU versions against.

The code generated by CVXGEN was much more sympathetic to a GPU implementation of RSFKM compared to CVXpy as it was library free and written in C, which directly maps to CUDA. However, this code also introduced new problems hindering a successful GPU implementation. First, it generates code that only works on a pre-defined, constant vector size. In our case, that means it only works for a fixed number of centroids. Next, as this generated code was optimized for embedded systems, many frequently accessed variables were set at global scope. As this code would have to be adapted to run sequentially on a single thread (but with multiple instances running in parallel across multiple cores) this scoping creates problems. CUDA doesn't have an equivalent global scope that exists independently for each thread, so a means to trick the architecture was required.

B. GPU Adaptation

As alluded to in Section III.A., the approach of prioritizing the adaptation of optimization code to GPUs came from two directions. First, the mathematical minimization formula for membership is fundamentally independent, indicating that calculations for membership vector u_i can occur at the same time as u_{i+1}. This makes the corresponding algorithm embarrassingly parallel. Second, the significant bottleneck of this minimization problem demanded resolution before other approaches to optimization and parallelization could be considered.

The problem of the globally scoped variables was solved first. To properly adapt the generated C code, a method was devised to maintain the structure of the program as developed to run on a CPU. To maintain the illusion of global scope, per-thread, the entirety of generated code, approximately 1500 lines, was encapsulated into a class-like struct. Encasing our C code into a struct allowed all required variables to be treated like private members which could be freely and safely accessed by the component functions which performed the minimization solving. The struct functions themselves were labeled as “__device__” functions and could then be called freely by individual threads from the ”Update Membership” kernel.

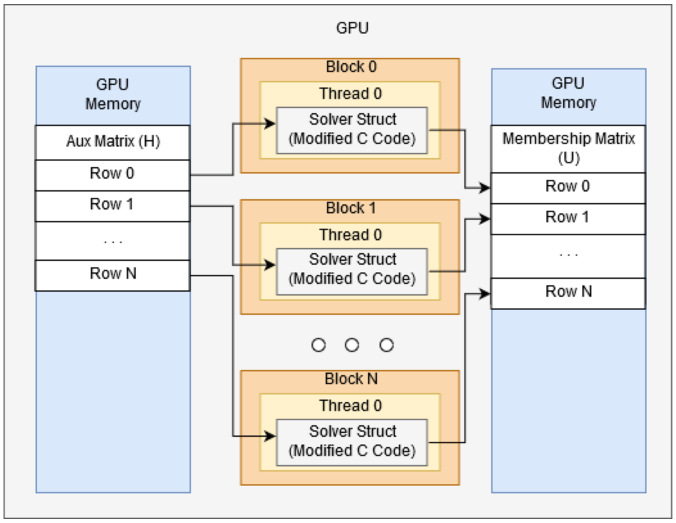

Fig. 1. A diagram showing a basic breakdown of the implementation and operation of the core solver implemented as a CUDA struct from modified C code. This diagram shows the mapping of the solver to each block and indicates how the problem is decomposed, where each line in our membership matrix is solved in parallel on the GPU. The decision to call this solver kernel on one thread per block came from slowdown observed with more than one thread per block.

As can be seen in figure 1, the modifed kernel was called on n blocks, each using one thread. Although this configuration seems naive, attempts to call this sequential code with multiple threads per block, in multiples of 32 to take advantage of warp efficiency, saw much slower results than the one-thread-per-block approach. Furthermore, very few threads-per-block could be instantiated due to a lack of register space. This is likely due to the size of the solver struct being loaded to run on the GPU. By using this structure, our solver could make the best use of GPU resources to perform the embarrassingly parallel computations promised by Equation (3) in the most straightforward manner possible.

C. Further Optimization

After converting the primary bottleneck of the minimization solver into GPU code, many other opportunities for optimization became evident. In addition to the aforementioned "Update Membership" function, there existed three other computationally intensive functions which comprised the core functionality of this algorithm. Those functions are **build_h_matrix**, **find_centroids**, **update_S**. Generally, these functions involved some kind of matrix manipulation with a sum reduction and could be easily mapped one matrix cell to a block with multiple threads doing reduction and simple addition work in parallel. Shared memory was used for each of these functions as a buffer to hold summed data, pre-reduction, for fast access and simplicity of implementation.

Through rewriting these functions in CUDA to run on the GPU, and adding supplemental functions to store and calculate auxiliary scalars between iterations, the entirety of this program (within a single iteration) was successfully configured to run on the GPU. We successfully minimized overhead with this full conversion by initializing all derived matrices, **U**,**V**,**H**, and **S** in global memory on the GPU. Each of these matrices can be maintained on the GPUs main memory between the core kernel calls which perform the computations of our clustering algorithm. Although highly efficient, this memory configuration caused problems in terms of space efficiency which are addressed by the multi-GPU implementation of this algorithm.

TABLE I
An excerpt of environmental data used for bench marking the effectiveness of the improved RSFKM clustering algorithm. The values depicted here are humidity and temperature measurements collected from many sites.

1/25/2018 1:19	-5.678	-6.482	-6.499	-6.455	0.491	. . .
1/25/2018 1:20	-5.654	-6.474	-6.499	-6.452	0.492	
1/25/2018 1:21	-5.697	-6.462	-6.479	-6.452	0.494	
1/25/2018 1:22	-5.774	-6.481	-6.499	-6.449	0.494	
1/25/2018 1:23	-5.788	-6.491	-6.515	-6.472	0.494	
1/25/2018 1:24	-5.793	-6.503	-6.519	-6.478	0.492	
1/25/2018 1:25	-5.732	-6.515	-6.538	-6.492	0.494	

The only matrix which could not be initialized on the GPU was the original data matrix **X**, because its contents are read from the disk. This however does not significantly impact overall run time because the overhead of transferring this data to the GPU occurs only once before clustering begins. After this, the only continuous data transfer occurs at the end of each iteration when our centroids, **V**, are transferred back down to the CPU to check for convergence. The overhead of this transfer is very minimal however, as the number of centroids is generally going to be relatively small to provide meaningful clustering. The final overhead from data movement between host and device comes from the output of the membership matrix **U**. Like our data matrix, this matrix transmits between GPU to CPU only one time after all the iterations of this clustering algorithm have concluded. Because of this, the overhead of this retrieval can be considered negligible.

D. Multi-GPU

The need for a multi-GPU implementation of this algorithm was driven by the memory limitations of our single-GPU implementation. **calculate_centroids** requires a substantial sum reduction resulting from a matrix multiplication between the data matrix and an auxiliary matrix which holds a regulating scalar. After multiplication, but before reduction, result vectors are stored in shared memory assigned to a block. This requires our shared buffer to hold n spaces of size 8 bytes for the double precision values. CUDA provides a max of 48 KB per block. Accordingly, this only allows for approximately 6,000 double precision values to be processed by this function.At one point in this algorithm, this function handles every data value we intend to cluster. With the space limitations, this indicates that we cannot cluster more than 6,000 values with this program.

Although there are certainly many approaches and solutions to this problem, we chose to leverage data decomposition and split our calculations across multiple GPUs. By randomly sampling data points and distributing them across GPUs

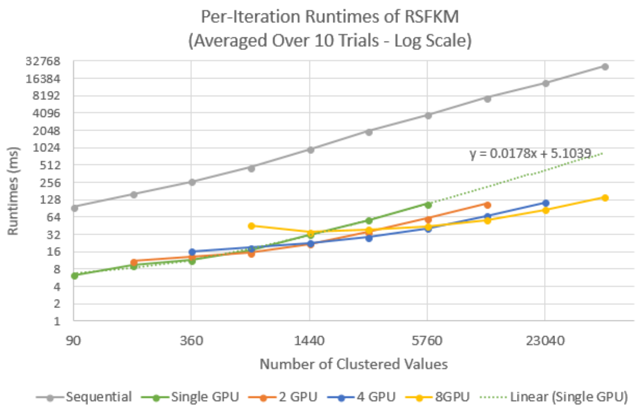

Fig. 2. A line plot showing the runtimes of the RSFKM algorithm running sequentially on a CPU and in parallel on one, two, four and eight GPUs. The runtimes are per iteration of the clustering algorithm and averaged over ten trials. The dotted line indicates a trendline which shows the projected runtimes of a single GPU experiment. The equation next to the trendline was used to derive runtime values for efficiency calculations.

we can accurately cluster up to 48,000 data points in the same amount of time it would take for 6000. Although the memory limitation remains, per-gpu, it becomes significantly less impactful with each GPU we add. This horizontal scaling is not significantly impacted by overhead due to the design of our per-gpu algorithm, where all substantial matrices are instantiated and maintained on each GPU's main memory. Data transfer occurs minimally in between interactions to check for convergence and at the end to transfer back results for imputation.

E. Data Imputation

The data imputation algorithm was implemented in Python as a collection of methods which performed basic preprocessing, called the above detailed GPU clustering algorithm, removed data for imputation and performed the imputation algorithm specified in Equation (5) using the centroids and returned membership matrix retrieved from the GPU. The implemented imputation method checked its accuracy using a simple RMSE algorithm finding the difference between the actual values removed from the dataset earlier against those which were imputed.

IV. Experiments and Analysis

To evaluate the improved RSFKM algorithm on a single GPU and multiple GPUs, timings were collected on repeated clustering of a set of environmental sensor data downloaded from the Nevada Research Data Center's website [1]. Preliminary experiments showed that changing the number of features on our data and adjusting the number of centroids did very little to influence overall runtimes for either CPU or GPU implementations of this algorithm. So, for all the following experiments, the number of centroids used is fixed at 15 and the number of features is fixed at 11.

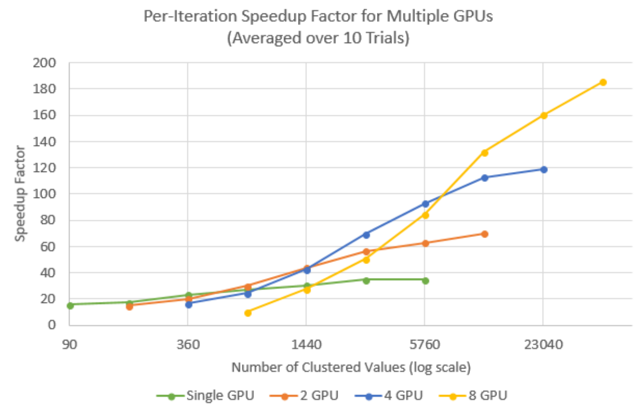

Fig. 3. A line plot showing the per-iteration speedup factor of singular and multi-GPU experiments run with the RSFKM algorithm. For larger numbers of clustered values, the addition of more GPUs produces near equivalently scaled speedup factors.

A. Experimental Setup

The following experiments were performed on a remote server containing 24 2.00 Ghz Intel Xeon CPUs connected over two PCIe buses to 8 GeForce GTX 1080 GPUs. The GTX 1080s are grouped 4 cards to a single bus. Each GPU has an available memory of 8GB with 6KB of shared memory available per block. Each GPU implements the Pascal architecture which provides support for advanced processing features like unified memory.

As an initial proof of concept, the clustered data set was a selection of 550,000 data points of time series data organized into 50,000 vectors with 11 features. Each vector represented a per-minute log of autonomously collected meteorological data with each feature representing a measurement collected at that minute from a distinct sensor. Table I shows an excerpt of the data set used.

B. Data Imputation

Although not explored in great detail for this paper due to constraints of space, this algorithm performed well in providing reasonably accurate data imputation. With RMSE values as low as 0.18 the GPU implementation of this algorithm provided accurate clustering on par with the CPU implementation. Since the applications and datasets were very different between this paper and those tested by Xu et. al. [15], a more detailed imputation experiment will be required and explored in future work.

C. Timings

Fuzzy K Means clustering, is a variable length iterative algorithm that does not converge uniformly at a set number of iterations. Between runs where the same data set is organized into the same number of clusters, convergence can occur in half as many iterations as expected. Accordingly, this makes collecting timings of full program runs problematic, because times can vary wildly. To solve this problem the runtimes collected and manipulated in the following sections were

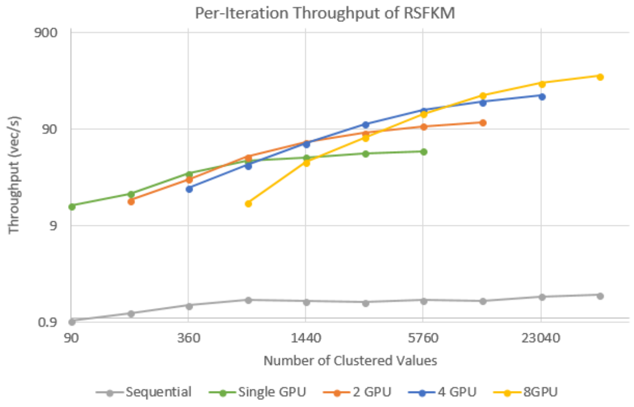

Fig. 4. A line plot showing per-iteration throughput of the RSFKM algorithm as run on CPU, one, two, four and eight GPUs. Throughput for this paper was expressed in terms of vectors/ms and was calculated as $thp = n/ms$ where n is the number of vectors input for clustering. GPU throughput handily outstrips sequential throughput by a wide amount, even when processing a small number of values.

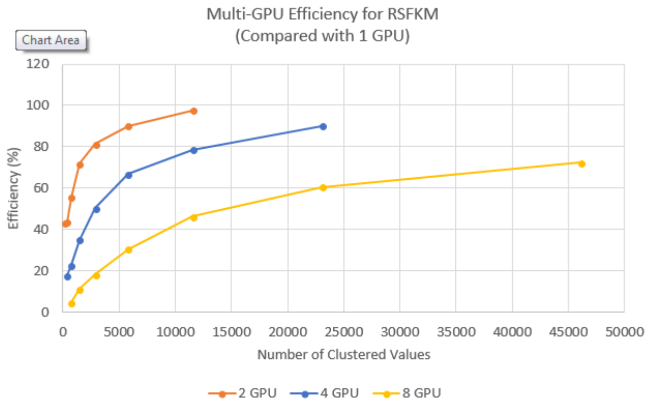

Fig. 5. A line plot showing the relative efficiency of multi gpu implementations vs. single GPU implementation. Calculated as $E = t_{gpu}/(t_{gpuN} * N) * 100$, where N is the number of GPUs being used, this graph indicates how much meaningful work each GPU is performing.

calculated per-iteration by dividing the overall runtime of this algorithm by the number of iterations taken. Using this timing scheme significantly reduced outliers in our timing data and produced meaningful results.

The raw per-iteration timings, shown in Fig. 2, show a comparison between the sequential implementation of RSFKM and our modified GPU implementation, on one, two, four and eight GPUs. The sequential runtimes, indicated by the grey line, indicate a clear linear curve which is to be expected by the *O(NVF)* complexity of our algorithm. As *V*, the number of centroids, and *F*, the number of features in the data vectors, are fixed at relatively small sizes compared to *N*, the algorithm becomes linear as *N* becomes sufficiently large.

At a glance its evident that GPU implementations were very successful in reducing the runtimes compared to the sequential algorithm. On this logarithmically scaled graph, per-iteration runtimes of GPU implementations remain relatively level as data inputs scale from 90 vectors to approx. 40000, ending with runtimes of 128ms up from 6ms. By comparison, across the same spread of processed data points the CPU implementation grows significantly in runtimes, from 128ms to almost 32000ms. The difference between these runtimes is so severe that when rendered with the CPU runtimes, without a logarithmic scaling, the GPU runtimes cannot be seen as anything other than a multicolored horizontal line on the bottom of the graph.

The significance of this difference in runtimes is further reinforced by Fig. 3 which shows the speedup factor of various GPU timings when compared with the sequential timings. The speedup factor, *S*, was calculated with the following equation: $S = t_s/t_p$, with t_s and t_p representing sequential and GPU runtimes respectively. Overall, the speedup was very promising with a low of around 10x speedup for 8 GPUs (with a small number of values) and a high of over 180x speedup for 8 GPUs. Speedup was shown to be more significant as more GPUs were added and additional data points could be processed. This increase in speedup is certainly consistent with expectations because there is very little cost to calculations performed with two GPUs, compared to one due to a lack of communication overhead. Broadly this indicates that when the fundamental overhead of CPU/GPU communication is overcome by data processing then speedup scales with the number of GPUs utilized. Analysis of throughput and efficiency reinforce the conclusions made in reference to timings and speedup. In Fig. 4, its shown that overall throughput increases with more GPUs but only for sufficiently large amounts of data. For smaller amounts of clustered data we see a general dropoff of throughput when compared with fewer GPU configurations, again likely due to increased overhead that comes with the addition of multiple GPUs.

Similarly, Fig. 5 most clearly shows that at lower values very little work is being done on each GPU. As more GPUs are added the lack of work being done is exacerbated dramatically. For 360 values, 2 GPUs get near 40% efficiency with below 10% 8 GPUs. This difference in efficiency drops off dramatically however as more data is processed by each GPU. We do see that overall efficiency still drops off with the addition of more GPUs despite increased throughput and speedup. Again, this is likely due to increased competition for bus access which is harder to offset as more GPUs are added.

Within the GPU runtimes themselves some characteristics of these lines should be noted. First, there is a trendline on the single GPU runtimes. As explained in Section III, space limitations prevented successful operation of this program on one GPU with more than 6000 initial values. A trendline enables us to visualize and estimate runtimes going out towards the max processed 40,000 values. This trendline and its derived data will help provide efficiency data for our multi-GPU implementations.

It should be noted that with the two, four and eight GPU experiments in Fig. 2 a bow-like curvature can be seen. This is almost certainly due to the overhead of transferred data between our multiple GPUs and the CPU. Since only two PCIe buses connect GPUs and CPUs the overhead of data transfer becomes more pronounced as GPUs are added and each GPU must compete to transfer data. And, even though there is relatively little data transferred between the GPU and CPU with this algorithm, there is certainly enough, especially with the per iteration transfer of our centroids, that slowdowns occur when there is not a sufficient amount of data for each GPU to process. The downward curve occurs as a result of more data being computed on each GPU and causing less frequent calls back to the CPU to check for convergence. This in turn, reduces simultaneous demand for the limited bus access.

V. Conclusion and Future Works

This paper presented improvements to an existing Robust and Sparse Fuzzy K Means algorithm, that allowed for the entirety of its processing to be performed on a single GPU or multiple GPUs. This improved algorithm was demonstrated to provide accurate imputation and facilitate much faster clustering through numerous experiments performed on environmental time-series data acquired from the NRDC.

Specifically, this paper demonstrated that robust clustering can be performed on a single GPU with as much as 34 times speedup. It was also shown that this algorithm scales very well horizontally with allowing for the use of up to 8 GPUs, resulting in as much as 185 times speedup over sequential methods. It was also demonstrated that this algorithm was very efficiently designed to minimize communication overhead between CPU and GPU, with efficiencies as high as 97% shown with two GPUs.

In developing this GPU-based iRSFKM algorithm many avenues for continued development and research appeared. First, adjustments to current memory organization in this algorithm which are needed to facilitate each GPU handling more than 6000 data points. When a key goal of graphics processing is the ability to manipulate large amounts of data quickly, this restriction significantly limits the applicability of iRSFKM (even with the help of additional GPUs).

A further area of research which expands on this work would certainly be a much more comprehensive exploration of the effectiveness of data imputation with this algorithm on environmental sensor data. A better curated and pre-processed data set could provide significantly better results for data imputation than what was shown in this paper. Additionally, the data imputation algorithm in this paper was implemented for CPU only and can be adapted for GPUs. By implementing code which performs the imputation itself in CUDA the overall algorithm could run much faster and be better encapsulated into a distributable package for active use by data scientists in need of fast, effective imputation.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant number IIA1301726. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] NRDC. (2018) Nevada Research Data Center. [last accessed December 12, 2018]. [Online]. Available: http://sensor.nevada.edu/NRDC/

[2] P. Schmitt, J. Mandel, and M. Guedj, "A comparison of six methods for missing data imputation," *Journal of Biometrics & Biostatistics*, vol. 06, May 2015.

[3] M. Soley-Bori, "Dealing with missing data: Key assumptions and methods for applied analysis," Boston University, School of Public Health, Department of Health Policy and Management, Tech. Rep. 4, May 2013.

[4] L. Beretta and A. Santaniello, "Nearest neighbor imputation algorithms: a critical evaluation," *BMC medical informatics and decision making*, vol. 16, no. 3, p. 74, 2016.

[5] J. C. Dunn, "A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters," *Journal of Cybernetics*, vol. 3, no. 3, pp. 32–57, 1973.

[6] T. Banerjee, J. M. Keller, M. Skubic, and E. Stone, "Day or night activity recognition from video using fuzzy clustering techniques," *IEEE Transactions on Fuzzy Systems*, vol. 22, no. 3, pp. 483–493, 2014.

[7] F. Valafar, "Pattern recognition techniques in microarray data analysis: A survey," *Annals of the New York Academy of Sciences*, vol. 980, no. 1, pp. 41–64, 2002.

[8] D. Li, J. Deogun, W. Spaulding, and B. Shuart, "Towards missing data imputation: A study of fuzzy k-means clustering method," in *Rough Sets and Current Trends in Computing*, S. Tsumoto, R. Słowiński, J. Komorowski, and J. W. Grzymała-Busse, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 573–579, 2004.

[9] Z. Liao, X. Lu, T. Yang, and H. Wang, "Missing data imputation: a fuzzy k-means clustering algorithm over sliding window," in *Fuzzy Systems and Knowledge Discovery, 2009. FSKD'09. Sixth International Conference on*, vol. 3. IEEE, pp. 133–137, 2008.

[10] J. Tang, G. Zhang, Y. Wang, H. Wang, and F. Liu, "A hybrid approach to integrate fuzzy c-means based imputation method with genetic algorithm for missing traffic volume data estimation," *Transportation Research Part C: Emerging Technologies*, vol. 51, pp. 29 – 40, Feb 2015.

[11] S. Azim and S. Aggarwal, "Hybrid model for data imputation: Using fuzzy c means and multi layer perceptron," in *2014 IEEE International Advance Computing Conference (IACC)*, pp. 1281–1285, Feb 2014.

[12] S. A. A. Shalom, M. Dash, and M. Tue, "Efficient k-means clustering using accelerated graphics processors," in *Data Warehousing and Knowledge Discovery*, I.-Y. Song, J. Eder, and T. M. Nguyen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 166–175, 2008.

[13] ——, "Graphics hardware based efficient and scalable fuzzy c-means clustering," in *Proceedings of the 7th Australasian Data Mining Conference - Volume 87*, ser. AusDM '08. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., pp. 179–186, 2008.

[14] M. Al-Ayyoub, A. M. Abu-Dalo, Y. Jararweh, M. Jarrah, and M. A. Sa'd, "A gpu-based implementations of the fuzzy c-means algorithms for medical image segmentation," *The Journal of Supercomputing*, vol. 71, no. 8, pp. 3149–3162, Aug 2015.

[15] J. Xu, J. Han, K. Xiong, and F. Nie, "Robust and sparse fuzzy k-means clustering," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, pp. 2224–2230, 2016.

[16] J. Huang, F. Nie, and H. Huang, "A new simplex sparse learning model to measure data similarity for clustering," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15. AAAI Press, pp. 3569–3575, 2015.

[17] A. Domahidi, E. Chu, and S. Boyd, "ECOS: An SOCP solver for embedded systems," in *European Control Conference (ECC)*, pp. 3071–3076, 2013.

[18] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.

[19] J. Mattingley and S. Boyd, "Cvxgen: a code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, Mar 2012.