Accelerating the Yinyang K-Means Algorithm Using the GPU

Colin Taylor
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
cmt389@nau.edu

Michael Gowanlock
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
michael.gowanlock@nau.edu

***Abstract*—The k-means clustering algorithm is widely employed for unsupervised learning. The algorithm takes as input a multidimensional dataset of points and number of clusters/centroids, k, where each point is assigned to one of the clusters. For exact k-means clustering, the algorithm must compute the same result as Lloyd's algorithm, which is well-known to be computationally expensive due to the large number of distance comparisons between each point and the k centroids. Several algorithms have been proposed for k-means clustering that avoid distance calculations but produce an exact result. However, these algorithms have all been designed for execution using the CPU, and no published works have examined using the GPU to accelerate k-means while simultaneously avoiding distance calculations. This paper examines the state-of-the-art Yinyang algorithm that avoids distance calculations as executed on the GPU. Since Lloyd's algorithm is well-suited to a GPU execution, it is not clear whether the Yinyang algorithm will obtain significant performance gains on GPU hardware. In this context, this paper: (i) proposes the first GPU-accelerated Yinyang algorithm in the literature; (ii) advances several optimizations to GPU kernels; (iii) contrasts and evaluates different degrees of distance calculation pruning; and, (iv) compares the performance of our GPU-accelerated Yinyang algorithm to four reference implementations. Our GPU algorithm achieves a speedup over the multi-core CPU Yinyang algorithm of up to 8× on real-world datasets.**

***Index Terms*—GPGPU, K-Means, Lloyd's Algorithm, Yinyang Algorithm**

I. Introduction

The k-means clustering algorithm takes as input a dataset, with n data points (or feature vectors), and a number of clusters, k. The algorithm outputs an assignment of each point to one of the k clusters. The algorithm can be employed for supervised [1], semi-supervised [2], and unsupervised classification [3] tasks that are commonly found in data analysis workflows and applications [4]. Lloyd's (the standard) algorithm [5] has two major phases: (i) the distance between each point in the dataset is computed between each of the k centroids, where each point is assigned to the closest centroid; and, (ii) an update step is performed which recomputes the location of each centroid based on the mean location of all points assigned to the centroid. The algorithm iterates until the points assigned to each centroid are stable and do not change centroids between two consecutive iterations.

This paper focuses on the *exact* k-means algorithm; in particular, like prior work, we focus on optimizations that reduce the number of distance comparisons between points and centroids at each iteration [6]–[10].

Modern GPUs have tremendous throughput relative to the CPU. For example, the Nvidia Ampere A100 has 1,200 GiB/s of on-card global memory bandwidth [11], which is over an order-of-magnitude greater than the bandwidth between the CPU and main memory. Since the k-means algorithm needs to perform many independent distance calculations, the algorithm is well-suited to GPU-acceleration, and several studies have examined using the GPU to improve the performance of Lloyd's algorithm [12], [13]. However, many of the state-of-the-art CPU algorithms that reduce the number of distance comparisons between points and centroids have not exploited the GPU. These algorithms introduce data-dependent execution pathways, additional branch conditions, and other characteristics that are not well-suited to the GPU's Single Instruction Multiple Thread (SIMT) architecture. For instance, threads executing in lock-step in a warp[1] that take divergent branches will serialize the execution, thus causing a loss of parallel efficiency [14]. Therefore, a major goal of this work is to examine how one of these distance avoiding algorithms performs on the GPU. In this paper, we propose accelerating the Yinyang k-means algorithm. This paper makes the following contributions:

- We propose the first GPU-accelerated Yinyang algorithm.
- We examine several GPU kernel optimizations, including exploiting shared memory to reduce register usage, and a centroid update scheme that achieves high resource utilization despite requiring synchronized global memory accesses.
- We compare the performance of our GPU algorithms to their parallelized multi-core CPU counterparts. Despite GPU-related overheads, the GPU Yinyang algorithm generally does not degrade performance relative to the CPU Yinyang, or Lloyd's algorithm as executed on the GPU. Therefore, our algorithm can directly replace these other implementations.

This paper is organized as follows. Section II presents related work and background material. Section III presents our GPU-accelerated k-means algorithms and optimizations. Section IV evaluates the proposed algorithms. Lastly, Section V concludes the paper and discusses future work directions.

[1]We use CUDA terminology throughout this paper.

II. Related Work

A. Exact K-Means Clustering and Prospects for the GPU

We summarize the literature that employs the CPU and the triangle inequality [6]–[8], which allows for using the distance between one point and a centroid to bound the distance to another centroid. This allows for reducing distance calculations when the bounded distance of a given centroid and a point exceeds the distance to a different centroid with a smaller known distance to the point. Elkan [6] proposed using the triangle inequality to speed up k-means. Hamerly [7] proposed adding a new lower bound to that proposed by Elkan [6]. Drake [15] extended the work of Hamerly [7] by including another additional test to reduce the number of distance calculations. Ding et al. [9] proposed the Yinyang algorithm, which builds on the aforementioned works by adding lower bounds to groups of clusters to reach a trade-off between the single lower bound proposed by Hamerly [7] and the $k-1$ lower bounds proposed by Elkan [6].

Newling and Fleuret [10] found that the final local filter in the Yinyang algorithm degrades performance compared to simply using the first two filters. Thus, we implement the Yinyang algorithm without the final local filter. Also, in our experimental evaluation, we employ the Yinyang algorithm with only the global filter to observe whether it may perform better than the two-filter version of the algorithm.

B. GPU-Accelerated K-Means

Few papers have examined using the GPU for k-means, and most focus on Lloyd's algorithm. We discuss the centroid update step, which has been a major focus of GPU k-means.

The centroid update step in k-means requires computing the mean of the positions of the points assigned to the centroid, which is inherently sequential. Li et al. [13] propose accelerating Lloyd's algorithm using the GPU and use a parallel reduction to update the positions of centroids. There exists source code for a Yinyang GPU algorithm called KMCUDA [16], but there are no published works on this implementation. Nelson and Palmieri [17] showed that k-means performs better using locks instead of synchronization-free approaches. KMCUDA updates centroid positions by assigning one GPU thread per centroid to independently update centroid positions without memory access conflicts [17]. Nelson and Palmieri [17] improve the performance of KMCUDA by evaluating several locking schemes to update centroids which allows more threads to update the centroid positions at the expense of synchronization. This method achieves better performance than the original KMCUDA implementation.

Drawing on the above, our algorithm updates centroids into two steps. We use atomic updates to ensure that multiple GPU threads can concurrently update the arrays needed to compute the centroid positions. Then, we utilize k threads to finalize the centroid calculation step.

III. GPU-Accelerated Yinyang k-means

In this section, we describe our simplified Yinyang algorithm, GPU-YYS. We begin by summarizing the algorithm.

A. Overview of the Yinyang Algorithm

We give a brief overview of Yinyang k-means, but refer the reader to Ding et al. [9] for a more thorough explanation. We use similar notation as that used by Ding et al. [9]. The Yinyang algorithm optimizes the point assignment step and the centroid update step of the standard k-means algorithm.

We describe the triangle inequality in the context of k-means as follows. Consider a dataset X consisting of n points. Let $a \in X$ be a point in the dataset and b and c be two centroids. Let $d(p, q)$ denote the metric distance (e.g., Euclidean distance) between a point p and a centroid q. The triangle inequality states that $|d(a, b) - d(b, c)| \leq d(a, c) \leq d(a, b) + d(b, c)$. Therefore, we can bound the distance between a and a third point, centroid c, using the known distance $d(a, b)$. This avoids unnecessary distance calculations.

The major difference among algorithms using the triangle inequality is the number of lower bounds that are stored. Yinyang has a parameter, t, which controls how many groups the centroids are placed into prior to the start of the algorithm. Each point $x \in X$ has one upper bound, $ub(x)$, and t lower bounds, $lb(x, g)$, where $g = 1, 2, \dots, t$.

Yinyang has three filters that use the point's upper and lower bounds to avoid distance calculations. The global and local filter both use the triangle inequality. The group filter uses a variant of the global filtering condition to avoid calculations with groups of centroids.

Before utilizing the filters, the Yinyang algorithm updates each lower bound $lb(x, g)$ using the maximum drift found among each group's centroids, where the maximum drift is denoted as $\delta(g)$. This is a value common to all points in the dataset that is determined during the centroid update step of the algorithm. It also updates the upper bound $ub(x)$ by the drift of each point's assigned centroid $\delta(b'(x))$. These updates ensure that each point's bounds remain accurate without performing additional distance calculations. The lower bounds $lb(x, g)$ and upper bound $ub(x)$ are initialized in the initial assignment step, then updated with data already calculated during the centroid update step. Once the bounds are updated, three separate filters are used (see Figure 1 and caption for detail).

B. Removing the Local Filter

Newling and Fleuret [10] showed that the local filter degrades performance. We remove the local filter and perform distance calculations on all points in each group that passed through the global and group filters. All of the exact distance avoiding algorithms that we described in Section II-A were designed for the CPU. The GPU uses the SIMT architecture, which makes thread divergence degrade performance due to the serialization of instructions [14]. Therefore, since the local filter adds additional branching, using this filter may degrade performance on the GPU. Consequently, we only use the global and group filters (Figure 1).

C. Overview of the Algorithm

The input parameters of GPU-YYS are as follows. The dataset, X, is a set of n points. Each point $x \in X$ contains the

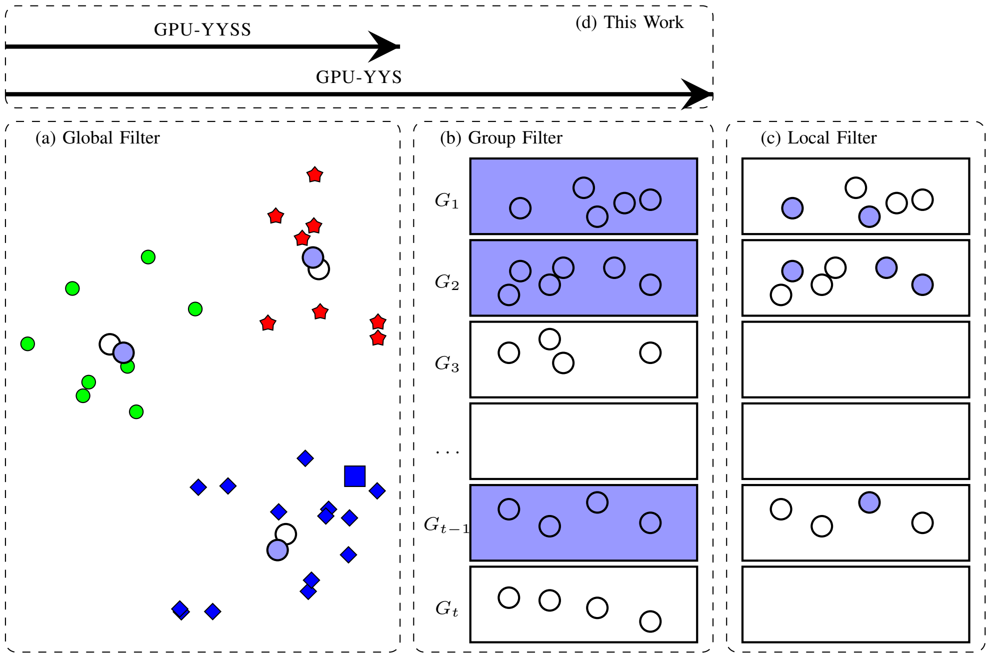

Fig. 1. The three filtering steps of Yinyang k-means. **(a) Global Filter:** Using the triangle inequality, the algorithm can avoid computing the distance to remote centroids. The point shown as a blue square in the figure will require the distance to be calculated between itself and the other three centroids, including its assigned centroid. Markers (small circles, stars, and diamonds) denote the assignment of points to centroids. Centroids in the previous iteration and current iteration are shown as large filled and unfilled circles, respectively. **(b) Group Filter:** The centroids are partitioned into t groups. Using the triangle inequality, the query point (blue square) only needs to consider those candidate centroids within the shaded groups, $G_g \in \{G_1, G_2, G_{t-1}\}$, whereas the centroids in the non-shaded groups can be safely ignored. **(c) Local Filter:** Within those groups that were not excluded from the group filter, the local filter further reduces the number of candidate centroids (shaded circles) within each group. This further reduces the number of distance calculations between a point and the centroids at each iteration. **(d) This Work:** We examine using the simplified Yinyang algorithm with the global and group filters (GPU-YYS), and the super simplified algorithm that only uses the global filter (GPU-YYSS).

coordinates of each point in d dimensions, an assigned centroid index ($b'(x)$) that was found in the current iteration, r, a former centroid ($b(x)$) found last iteration, $r-1$, a single upper bound $ub(x)$, and t lower bounds $lb(x, g)$, where $g = 1, 2, \ldots, t$. The algorithm requires the initial centroids generated for the dataset, that we denote as the set C. We denote each centroid as $c_j \in C$ where $j = 1, 2, \ldots, k$. All $c_j \in C$ are grouped into t groups, denoted by G_g.

A pseudocode overview is outlined in Algorithm 1. Our implementation of GPU-YYS utilizes a master loop that launches the kernels. Before the loop, a host function assigns centroids to t groups. The standard k-means algorithm is executed using the generated centroids as input to GPU-YYS. As prescribed by Ding et al. [9], standard k-means is executed for 5 iterations on the centroids in order to produce reasonable groupings for the Yinyang algorithm. These groupings do not change after their assignment in this function. This occurs on the CPU before transferring the set of centroids, C, to the GPU. We elected to compute this on the host because the GPU is likely to be underutilized for this task.

GPU-YYS is made up of four main kernels and several support kernels. Due to space constraints, we are unable to describe these kernels in detail. The INITIALIZEPOINTS kernel assigns each point to its initial cluster. The CALCULATECENTROIDDATA and CALCULATENEWCENTROIDS kernels are used to update the centroids in the Yinyang algorithm. CALCULATECENTROIDDATA uses n threads to update four arrays stored in global memory. These four arrays (P, P', Y, and Y') are all of size k and store the data used in the new cluster centroids. P and P' are two arrays of size k that store the number of points assigned to each centroid in both the previous and current iteration, respectively. Similarly, Y and Y' are two arrays of size k that contain position vectors representing the sums of all points assigned to each centroid in both iteration r and $r-1$. They record the number of points assigned to each centroid in the current and previous iteration, and k vector sums of every point assigned to that centroid.

The ASSIGNPOINTS and POINTCALCULATIONS kernels are used to assign the points to each centroid and perform distance calculations as necessary (POINTCALCULATIONS is called by ASSIGNPOINTS). Common to all of these kernels is employing the GPU's massive parallelism to assign work to threads (e.g., each point is assigned a single thread in three of the four main kernels).

D. *Summary of Optimizations*

GPU architecture necessitates careful usage of limited resources. Because the Yinyang algorithm has to perform record keeping for many intermediate values, it is important to exploit shared memory to achieve good performance. Shared memory can be used to store values that occupy more space than what can be efficiently stored in registers. We utilize shared memory

Algorithm 1 Yinyang Master Loop

```
1: procedure YINYANGGPU($X$, $C$, $\Delta$)
2:     GROUPCENTROIDS($C$)
3:     INITIALIZEPOINTS($X$, $C$)
4:     while NotConverged do
5:         CALCULATECENTROIDDATA($X$, $C$, $Y$, $Y'$, $P$, $P'$)
6:         CALCULATENEWCENTROIDS($X$, $C$, $\Delta$, $Y$, $Y'$, $P$, $P'$)
7:         ASSIGNPOINTS($X$, $C$, $\Delta$)
8:     end while
9: end procedure
```

within the ASSIGNPOINTS kernel that stores $4t$ bytes for each point. This space is used to mark groups that the point must be compared to in the group filter. We use shared memory as we found that storing large masking arrays in registers severely limited the occupancy of the kernel due to high register usage.

We use two kernels to update centroids. We examined using a single kernel that uses k threads to update the centroids, which is similar to the CPU Yinyang centroid update step (see Section II-B). However, this solution leads to extremely inefficient use of the GPU's resources because typically $k \ll n$ (i.e., we execute far fewer threads than are required to saturate GPU resources). To address this, we use a separate kernel which utilizes n total threads that atomically update k memory locations in global memory to update the centroids. This significantly increases the amount of work that is computed in parallel and outweighs synchronization overhead.

IV. Experimental Evaluation

A. Experimental Methodology

All host code is written in C/C++ and GPU code is written in CUDA v.9. The host code is compiled using the GNU compiler with the O3 optimization flag. Our experiments are carried out on a platform with 2x 2.10 GHz Intel Xeon E5-2620 v4 CPUs with 16 total physical cores, and 128 GiB of main memory. The GPU is an Nvidia GP100 with 16 GiB of global memory. All data is stored as 64-bit floating point values. The source code is publicly available[2].

All time measurements are averaged over three time trials. We include the total end-to-end time including grouping cluster centroids on the host, data transfers from the host to the device and vice versa, and allocating data on the device. However, we exclude the time to load the dataset from disk and generate initial cluster centroids, as these initial centroids are computed in all implementations. All algorithms execute until they converge, which occurs when none of the points, $x \in X$, change their cluster assignments between two consecutive iterations. All algorithms are exact, meaning that they output the same clusters as Lloyd's algorithm; therefore, they all converge at the same iteration. We limit the maximum number of iterations to 3,000; however, there was only a single case in our experiments where this occurred. Also, like related work, we use the Euclidean distance.

Lloyd's algorithm does not prescribe how to treat centroids without members. In all of our implementations, if a centroid does not have any points assigned to it at a given iteration, we simply leave the centroid at its current position, as the centroid may gain new members in future iterations.

[2]https://github.com/ctaylor389/k_means_yinyang_gpu.

TABLE I
PROPERTIES OF THE REAL-WORLD DATASETS EMPLOYED IN THE EVALUATION, SHOWING THE NUMBER OF DATA POINTS, n, THE NUMBER OF DIMENSIONS, d, AND REFERENCES TO THE DATA.

Dataset	n	d	Ref.
MSD	515,345	90	[18]
Census	2,458,285	68	[20]
SuSy	5,000,000	18	[19]
Higgs	11,000,000	28	[19]

TABLE II
THE NUMBER OF ITERATIONS REQUIRED BEFORE CONVERGENCE ON THE REAL-WORLD DATASETS FOR SELECTED VALUES OF k.

k	*MSD*	*Census*	*SuSy*	*Higgs*
8	142	5	83	92
16	143	45	214	360
32	472	49	374	695
64	353	123	540	1424
128	812	77	793	1325
256	518	257	1009	2855
512	740	205	1264	1744
1024	599	259	2298	3000

B. Datasets

We use four real-world datasets in our evaluation. The real-world datasets are outlined as follows: *MSD* ($d = 90$) contains the features of songs [18], *Census* ($d = 68$) contains data from the 1990 census in the U.S.A., *SuSy* ($d = 18$) and *Higgs* ($d = 28$) contain particle physics feature vectors [19]. Table I summarizes the properties of the real-world datasets. Table II outlines the number of iterations required before convergence.

C. Implementations

We outline our implementations and configurations.
•GPU-YYS is configured with a value of $t = 20$ for all values of k and includes the global and group filters used in the Yinyang algorithm. As we will show, t is an experimentally derived parameter controlling the number of groups, where $t = 20$ performs well across experimental scenarios. In all kernels, we use 256 threads per block.
•GPU-YYSS is the same as GPU-YYS, except it only uses the global filter. Because it does not have a group filter, it does not group centroids on the host and performs all distance calculations when any centroid passes the global filter.
•GPU-LLOYD is an implementation of the standard k-means algorithm on the GPU. It uses a two-part centroid calculation consistent with the analogous steps in the GPU Yinyang implementations.
•CPU-YYS is a parallel CPU implementation of the simplified Yinyang algorithm specified by Newling and Fleuret [10], and uses the global and group filters, similarly to GPU-YYS. It is parallelized using 16 threads using OpenMP, corresponding to the number of physical cores on our platform.
•CPU-LLOYD is a CPU implementation of the standard k-means algorithm, parallelized the same as CPU-YYS.

•KMCUDA is a GPU implementation [16] configured using 32-bit floats. When we compare GPU-YYS to KMCUDA, we configure our algorithm using 32-bit floats.

D. *Selecting the Number of Groups in the Yinyang Algorithm*

We performed experiments on the *MSD*, *Census*, and *SuSy* datasets for $k \in \{128, 1024\}$. In summary, we find that the heuristic proposed by Ding et al. [9] of $t = k/10$ yields good performance across these datasets and values of k on CPU-YYS. In contrast, on the GPU using GPU-YYS we find that when $t > 20$, there is either no performance gain or even an increase in average iteration runtime. Consequently, in our experiments, on CPU-YYS we select $t = k/10$, and on GPU-YYS, we select $t = 20$ for all datasets with $k \geq 20$, otherwise we select $t = k$ when $k < 20$.

E. *Scalability of the CPU Algorithms*

Both of the CPU implementations (CPU-LLOYD and CPU-YYS) consistently perform best using 16 CPU threads on our 16-core platform; therefore, in all that follows, we use 16 threads in our CPU implementations.

F. GPU-YYS*: Kernel Time Breakdown*

Table III shows the percentage of the total response time on the *MSD* and *SuSy* datasets for $k \in \{32, 64, 128\}$ for each of the four main kernels in GPU-YYS (Algorithm 1). These datasets span the minimum and maximum dimensionality of the four datasets. As with the standard k-means algorithm, the distance calculation kernel (ASSIGNPOINTS) in GPU-YYS is expected to require the greatest percentage of time. The percentage of time spent computing distance calculations in the ASSIGNPOINTS kernel generally increases with k, and it ranges from 61.02–89.15%. This indicates that the additional kernels that are required of GPU-YYS compared to GPU-LLOYD are not prohibitive to performance.

We find that the INITIALIZEPOINTS kernel requires a non-negligible fraction of the total time. CALCULATENEWCENTROIDS is generally negligible with the exception of *MSD* at $k = 32$. In addition, the "other" category which primarily refers to all other overheads (host/device data transfers and checking for convergence) requires a non-negligible fraction of time, but it decreases with k.

G. *Comparison of all Approaches*

Table IV presents the speedup of GPU-YYS over the other algorithms. Note that because CPU-LLOYD requires a significant amount of execution time, we report the speedup as the ratio of the average iteration execution times and limited CPU-LLOYD to 200 iterations. We describe the key points as follows.

1) GPU-YYS always outperforms the parallel multi-core CPU-YYS algorithm, yielding speedups between 2.08× and 8.12×.
2) Comparing the GPU implementations, we observe that across all datasets, the speedup of GPU-YYS over GPU-YYSS and GPU-LLOYD generally increases with k. Furthermore, *MSD* and *Census* are of greater dimensionality than *SuSy* and *Higgs*. We find that GPU-YYS has greater speedups over GPU-YYSS and GPU-LLOYD on datasets with greater dimensionality. Therefore, GPU-YYS performs best on datasets with larger values of k, and d (i.e., those datasets and parameters that have the greatest workloads). This demonstrates that the additional filtering power of GPU-YYS over GPU-YYSS improves performance.
3) The super simplified algorithm, GPU-YYSS only uses the global filter. It outperforms GPU-YYS on lower values of k on *SuSy* and *Higgs*. This indicates that when k and d are low, it may be preferable to avoid using the group filter.

We compare GPU-YYS to KMCUDA [16], which is a GPU-accelerated Yinyang algorithm that computes all three filtering stages (Section II-B). Our implementations by default are configured to use 64-bit floating point precision; however, KMCUDA uses 32-bit precision. Consequently, we execute GPU-YYS and KMCUDA with 32-bit precision to compare performance. We allow our clusters to regain members if they become empty in a given iteration and KMCUDA does not. Therefore, the number of iterations required for convergence vary between KMCUDA and GPU-YYS. Thus, to compare performance as the speedup of GPU-YYS over KMCUDA, we use the average iteration execution time. From Table IV, we find that GPU-YYS achieves a speedup over KMCUDA spanning 3.54–104.14×. Furthermore, in many cases, KMCUDA performs worse than GPU-LLOYD. We attribute this to the centroid update step, where our algorithm uses many more threads than KMCUDA at the expense of using atomic updates to memory locations. We found that this was much more efficient than using a lock-free approach that uses fewer threads. The centroid update step of KMCUDA is described in greater detail in Nelson and Palmieri [17] (Section II-B).

V. CONCLUSION

To our knowledge, this paper proposes the first GPU-accelerated Yinyang k-means algorithm in the literature. The Yinyang algorithm is one of several distance-avoiding algorithms that employs the triangle inequality and other filters to reduce the number of distance calculations while also outputting a solution identical to Lloyd's algorithm. Lloyd's algorithm requires performing many independent distance calculations which is a task well-suited to execution on the GPU. In contrast, the Yinyang algorithm requires maintaining many bounds and other bookkeeping information to eliminate distance calculations. Consequently, the code complexity increases relative to Lloyd's algorithm; therefore, it was not guaranteed that Yinyang would outperform Lloyd's algorithm on the GPU.

We illustrated that on the higher workloads, such as high dimensionality (d), large numbers of centroids (k), and number of data points (n), GPU-YYS achieves significant performance gains over GPU-LLOYD and CPU-YYS. On the smaller workloads, we find that GPU-YYS achieves minimal performance gains over CPU-YYS, as the algorithm has fewer distance calculations to compute and hence, fewer calculations that can be avoided. Despite this, GPU-YYS

TABLE III

PERCENTAGE OF THE TOTAL TIME REQUIRED BY THE MAJOR GPU-YYS KERNELS ON THE *MSD* AND *SuSy* DATASETS.

	MSD ($k = 32$)	*MSD* ($k = 64$)	*MSD* ($k = 128$)	*SuSy* ($k = 32$)	*SuSy* ($k = 64$)	*SuSy* ($k = 128$)
ASSIGNPOINTS	61.02	79.13	89.15	64.61	68.66	76.44
CALCULATECENTROIDDATA	10.99	5.32	2.66	10.18	13.61	8.84
CALCULATENEWCENTROIDS	5.33	0.51	0.46	0.06	0.06	0.06
INITIALIZEPOINTS	7.11	7.59	5.07	2.15	2.97	4.01
Other	15.55	7.45	2.66	23.00	14.70	10.65

TABLE IV

SUMMARY OF THE SPEEDUP (OR SLOWDOWN) OF GPU-YYS OVER THE OTHER ALGORITHMS.

k	Algorithm	*MSD*	*Census*	*SuSy*	*Higgs*
8	CPU-YYS	7.85	2.08	3.56	3.77
	CPU-LLOYD	45.34	3.28	24.80	23.13
	GPU-YYSS	1.32	1.11	0.67	0.95
	GPU-LLOYD	9.10	1.14	1.22	1.28
	KMCUDA	90.53	77.71	104.14	75.29
128	CPU-YYS	6.14	8.00	7.00	6.19
	CPU-LLOYD	41.29	28.62	61.09	70.43
	GPU-YYSS	7.30	4.97	0.52	0.65
	GPU-LLOYD	33.37	23.24	1.60	1.81
	KMCUDA	14.26	19.73	41.96	44.38
512	CPU-YYS	2.93	6.47	7.37	7.10
	CPU-LLOYD	30.09	61.89	98.23	115.29
	GPU-YYSS	9.42	9.55	0.87	1.15
	GPU-LLOYD	26.29	53.68	1.76	2.24
	KMCUDA	5.76	9.55	24.19	25.24
1024	CPU-YYS	2.09	5.59	8.12	6.51
	CPU-LLOYD	25.03	70.09	143.60	149.04
	GPU-YYSS	7.96	12.36	1.12	1.53
	GPU-LLOYD	21.83	62.45	2.33	2.70
	KMCUDA	3.54	12.22	12.62	14.95

obtains a speedup over CPU-YYS across all values of k on each dataset. Therefore, GPU-YYS can be utilized as a direct replacement for the CPU Yinyang and Lloyd's GPU algorithm.

We also found that our algorithm significantly outperformed KMCUDA. We believe that this is due to the way in which KMCUDA updates its centroids, and also that KMCUDA uses all three filtering stages which are likely less efficient than using the simplified two-filter algorithm.

Future work includes investigating using the Yinyang algorithm on multiple GPUs, in addition to scaling the algorithm on distributed-memory compute nodes equipped with GPUs. For this task, we may utilize emerging technologies designed for reaching exascale, such as ROCm [21].

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344.

REFERENCES

[1] B. Nguyen and B. De Baets, "Kernel-based distance metric learning for supervised k-means clustering," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 10, pp. 3084–3095, 2019.

[2] K. Thangavel and A. K. Mohideen, "Semi-supervised k-means clustering for outlier detection in mammogram classification," in *Trendz in Information Sciences & Computing (TISC2010)*. IEEE, 2010, pp. 68–72.

[3] G. Hamerly and J. Drake, "Accelerating Lloyd's algorithm for k-means clustering," in *Partitional clustering algorithms*. Springer, 2015, pp. 41–78.

[4] L. Lucchese and S. Mitra, "Unsupervised segmentation of color images based on k-means clustering in the chromaticity plane," in *Proceedings IEEE Workshop on Content-Based Access of Image and Video Libraries (CBAIVL'99)*. IEEE, 1999, pp. 74–78.

[5] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[6] C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, 2003, pp. 147–153.

[7] G. Hamerly, "Making k-means even faster," in *Proceedings of the 2010 SIAM international conference on data mining*. SIAM, 2010, pp. 130–140.

[8] J. Drake and G. Hamerly, "Accelerated k-means with adaptive distance bounds," in *5th NIPS workshop on optimization for machine learning*, vol. 8, 2012.

[9] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. Mytkowicz, "Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup," in *International Conference on Machine Learning*, 2015, pp. 579–587.

[10] J. Newling and F. Fleuret, "Fast k-means with accurate bounds," in *International Conference on Machine Learning*, 2016, pp. 936–944.

[11] "Nvidia Ampere," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, accessed: August 6, 2020.

[12] J. D. Hall and J. C. Hart, "GPU acceleration of iterative clustering," in *The ACM Workshop on General Purpose Computing on Graphics Processors*, 2004, pp. 45–52.

[13] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-means algorithm by GPUs," *Journal of Computer and System Sciences*, vol. 79, no. 2, pp. 216–229, 2013.

[14] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 83–94.

[15] J. Drake, "Faster k-means clustering," Ph.D. dissertation, 2013.

[16] V. Markovtsev, "KMCUDA," https://github.com/src-d/kmcuda, accessed: August 6, 2020.

[17] J. Nelson and R. Palmieri, "Don't Forget About Synchronization! A Case Study of K-Means on GPU," in *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2019, pp. 11–20.

[18] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval* , 2011.

[19] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, p. 4308, 2014.

[20] U. M. L. Repository, "US Census Data (1990) Dataset," accessed: August 31, 2020.

[21] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 209–218.