2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)

Interactive Kernel Dimension Alternative Clustering on GPUs

Xiangyu Li, Chieh Wu, Shi Dong, Jennifer Dy, David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA
{*xili, chiehwu, shidong, jdy, kaeli*}*@ece.neu.edu*

***Abstract*—Machine learning has seen tremendous growth in recent years thanks to two key advances in technology: massive data generation and highly-parallel accelerator architectures. The rate that data is being generated is exploding across multiple domains, including medical research, environmental science, web-search, and e-commerce. Many of these advances have benefited from emergent web-based applications, and improvements in data storage and sensing technologies. Innovations in parallel accelerator hardware, such as GPUs, has made it possible to process massive amounts of data in a timely fashion. Given these advanced data acquisition technology and hardware, machine learning researchers are equipped to generate and sift through much larger and complex datasets quickly.**

In this work, we focus on accelerating Kernel Dimension Alternative Clustering algorithms using GPUs. We conduct a thorough performance analysis by using both synthetic and real-world datasets, while also modifying both the structure of the data, and the size of the datasets. Our GPU implementation reduces execution time from minutes to seconds, which enables us to develop a web-based application for users to, interactively, view alternative clustering solutions.

***Keywords*-KDAC, clustering, GPU, Machine Learning, Big Data**

I. Introduction

The exponential growth of high-dimensional data from various domains presents many exciting opportunities for machine learning research. However, along with these opportunities, researchers are faced with a number of challenges: 1) they need more flexible and accurate machine learning algorithms to deal with the complexity of the data, and 2) they need more powerful computing platforms to run the algorithms efficiently. Machine learning researchers will need better tools and platforms to extract the relevant and insightful information hidden in these rich volumes of data.

Clustering is a widely-used unsupervised machine learning algorithm. A common objective of clustering is to discover hidden patterns within a given dataset, and then to use the patterns to form the basis of separating the data into meaningful subgroups. There already exist many well-established algorithms with admirable performances [1]. Although these algorithms hold great promise for clustering data into similar groups, using a single clustering algorithm may be insufficient given the characteristics of real-world data. This is because most clustering algorithms only discover one clustering solution, while real-world data can always be grouped in different ways for different purposes. Therefore, machine learning researchers often need to use multiple different clustering algorithms to reveal patterns to obtain a better perspective of the data. For example, given a dataset of pictures, with a single person on each picture, it is possible to cluster the pictures in different ways depending on the objective of the viewer. Each person could be grouped based on his/her hair color, height, race, gender, etc. It is clear we may need to apply different clustering algorithms to arrive at clustering solutions from multiple perspectives. Since the complexity of real-world applications rarely produces datasets with a single interpretation, using a clustering algorithm that is capable of shifting our clustering perspective could be the key to discovering new relationships.

Kernel Dimension Alternative Clustering (KDAC) [2] is a clustering algorithm that iteratively generates multiple alternative clustering solutions for the same dataset. The multiple views are generated within a single algorithmic framework, while users can control the trade-offs between the cluster quality and the cluster novelty. Figure 2b is an example of three different clustering views of a colored butterfly image. The applicability of KDAC to real-world problems is highly attractive to researchers, but this clustering approach introduces high computational complexity. Providing a KDAC implementation that can efficiently produce high-quality alternative clusters should have an impact regarding practical exploratory data analysis.

GPUs can provide high-throughput computations through their thousands of processing cores and high-bandwidth memory. GPUs have become great candidates for machine learning algorithms [3], [4] because of two reasons: 1) typical machine learning algorithms need to process large training datasets, in which each sample can be processed independently, matching a GPU's Single Instruction Multiple Data (SIMD) processing model well, and 2) GPUs have been shown to be extremely effective performing linear algebra operations [5], which are frequently used in machine learning algorithms.

IEEE/ACM ASONAM 2018, August 28-31, 2018, Barcelona, Spain
978-1-5386-6051-5/18/$31.00

This paper makes the following contributions:

- We conduct a thorough performance analysis of executing KDAC on different platforms using synthetic and real-world datasets.
- We reduce the execution time of KDAC from minutes to seconds, enabling interactive clustering to aid exploratory data analysis.
- We integrate KDAC into a big data analytics framework, which offers a web-based front-end that enables users to interactively visualize and analyze the clustering solutions.

The rest of the paper is organized as follows. Section II provides an overview of the KDAC algorithm and GPU architectures considered in this work. Section III describes the KDAC algorithm in detail and analyzes the complexity and performance bottlenecks associated with this algorithm. Section IV presents our efficient implementation of KDAC on a GPU. Section V briefly introduces our big data analytics framework. Section VI presents a thorough performance evaluation of KDAC in different settings, and Section VII concludes the paper.

II. Background

A. Kernel Dimension Alternative Clustering

Kernel Dimension Alternative Clustering (KDAC) is an iterative alternative clustering algorithm, in which a new clustering solution can be created based on the previous solutions. The alternative solution should be of both high quality and novelty, meaning that not only does it reveal the hidden pattern in the data, but it also is dissimilar to previous solutions, thus providing a new perspective for the same data.

With the first clustering solution denoted as P_0, the goal of KDAC is to find a new clustering solution P_1 based on P_0, and then to find P_2 based on the set of $\{P_0, P_1\}$. The iteration continues until P_t is found based on the set of all the previous solutions $\{P_0, P_1, ...P_{t-1}\}$ where t is the number of alternative solutions specified by users.

The input for KDAC includes a dataset X and an existing clustering solution Y_0. X consists of n samples $\{x_0, ..., x_{n-1}\}$, where each sample x_i is a column vector with d features. Therefore, $X = [x_0, ..., x_{n-1}]^T \in \mathbb{R}^{n \times d}$. Let us denote c as the number of desired clusters, a binary cluster labeling matrix $Y_0 \in \mathbb{R}^{n \times c}$ is generated from P_0 where $y_{ij} = 1$ if x_i belongs to cluster j, and $y_{ij} = 0$ otherwise. Similarly, when we have multiple previous solutions $\{P_0, P_1, ...P_{t-1}\}$, an augmented matrix $Y = [Y_0, ..., .Y_{t-1}]$ is created. Y has n rows and $\sum_{j=0}^{t-1} c_j$ columns, where c_j is the number of clusters in the jth view. Denoting $U \in \mathbb{R}^{n \times c}$ as the cluster labeling matrix for P_t, the goal of KDAC is to find U from the dataset X and the existing solution Y, where U represents an alternative clustering solution, being good in both cluster quality and novelty. U is equivalent to the spectral embedding in spectral clustering algorithm [6]. Based on this embedding, the discrete clustering solution is obtained from a "rounding" step, where each row in U is normalized, and K-Means algorithm is applied to each normalized row. We then assign each x_i to the cluster that row u_i belongs to.

To achieve these goals, KDAC performs clustering upon a subspace $W \in \mathbb{R}^{d \times q}$, where $q \ll d$, and combines dimensionality reduction and Hilbert-Schmidt Independence Criterion (HSIC) [7] to form its objective function:

$$\max_{U,W} \ \mathrm{HSIC}(XW, U) - \lambda\, \mathrm{HSIC}(XW, Y) \qquad (1)$$
$$s.t \quad U^T U = I,\ W^T W = I$$

To maximize (1) is to ensure the first term is large and the second term is small. This optimization searches for a subspace W such that the relationship between the projected data and the new clustering solution U is strong (a clustering solution with high quality), while its relationship with the existing clustering Y is weak (a clustering solution with high novelty). The parameter λ is specified by the user to control the balance of the cluster quality and novelty.

Let us denote K as the kernel matrix defined based on the subspace of W, where $k_{ij} = k(W^T x_i, W^T x_j)$, D as the degree matrix of K, and H as the centering matrix. The first HSIC term in Equation 1 can be expressed using spectral clustering criterion as follows:

$$\max_{U,W} \ \mathrm{Tr}\left(U^T D^{-\frac{1}{2}} K D^{-\frac{1}{2}} U\right) - \lambda\, \mathrm{Tr}\left(Y^T HKHY\right) \qquad (2)$$
$$s.t \quad U^T U = I,\ W^T W = I$$

By optimizing equation (2), we are able to simultaneously discover a projection matrix W and an alternative clustering solution U. The optimization approach initializes both variables U_0 and W_0. By holding one variable constant while optimizing the other, the objective function converges toward a local minimum.

B. GPU Architectures and Programming Model

GPUs are highly parallel computing engines that devote most of their silicon to processing cores rather than complex latency hiding. OpenCL [8] and CUDA [9] providing a high-level programming model, allowing the user to harness the power of GPUs much more easily. We use CUDA terminology in the rest of the paper since our accelerated-KDAC is built upon an NVIDIA/CUDA GPU framework. A GPU is composed of multiple Streaming Multiprocessor (SMX), each of which consists of thousands of CUDA cores. For example, the Tesla K40 used in our experiment contains 2880 CUDA cores. Each SMX organizes 32 CUDA cores into a warp that execute in lockstep. Programmers can launch

a big number of parallel software threads, where each thread is mapped to a CUDA core and executes the same instruction simultaneously. These threads are grouped into blocks, and multiple blocks form a grid. A GPU program, also called a kernel, is executed by every thread in parallel. The memory hierarchy in a GPU stresses sustained bandwidth versus complexity — a large off-chip slow global memory with a small on-chip fast shared memory.

III. KDAC

Next, we describe in detail how to solve the objective function in Equation 2. We first fix the subspace W, and only optimize for the labeling matrix U. Then we fix U and solve for W. We describe these two steps in the following two sections.

A. Optimize Matrix U

When W is fixed, we are essentially maximizing the first term in Equation 2 because every component in the second term is fixed. We first generate the kernel matrix K based on input matrix X's projection on the subspace defined by W, and then we calculate the degree matrix by summing up each row in K. With K and D created, the optimization problems becomes an eigenvalue problem. The solution for U is equal to the first k eigenvectors (corresponds to the k largest eigenvalues) of the matrix $D^{-\frac{1}{2}}KD^{-\frac{1}{2}}$, where k is the cluster number specified by the user.

B. Optimize Matrix W

When U is fixed, Equation 2 can be written in the following form using cyclic property of the trace function:

$$\max_W \ \operatorname{Tr}(D^{-\frac{1}{2}}UU^TD^{-\frac{1}{2}}K) - \lambda \operatorname{Tr}\left(HYY^THK\right) \quad (3)$$

We use matrix A to denote $D^{-\frac{1}{2}}UU^TD^{-\frac{1}{2}}$ and matrix B to denote HYY^TH, Equation 3 can be written as

$$\max_W \ \operatorname{Tr}(AK) - \lambda \operatorname{Tr}(BK) \quad (4)$$

Because $tr(AK) = \sum_{ij} a_{ij}k_{ji}$ and K is symmetric, $tr(AK)$ can be represented as $\sum_{ij} a_{ij}k_{ij}$. We can then re-write Equation 4 as:

$$\begin{aligned} &\max_W \ \sum_{ij} \frac{\boldsymbol{u_i^T u_j}}{d_i d_j} k_{ij} - \lambda \sum_{ij} \widetilde{y}_{ij} k_{ij} \\ =\ &\max_W \ \sum_{ij} (\frac{\boldsymbol{u_i^T u_j}}{d_i d_j} - \lambda \widetilde{y}_{ij}) k_{ij} \\ =\ &\max_W \ \sum_{ij} \gamma_{ij} k_{ij}, \end{aligned} \quad (5)$$

where $\boldsymbol{u_i}$ is the column vector representing the ith row in matrix U, and $d_i = \sqrt{d_{ii}}$ (the square root of the ith diagonal element of D). $\frac{\boldsymbol{u_i^T u_j}}{d_i d_j}$ is the element (i, j) of matrix $(D^{-\frac{1}{2}}UU^TD^{-\frac{1}{2}})$, and $\widetilde{y}_{ij}$ corresponds to the elements of matrix (HYY^TH). Using γ_{ij} to represent $(\frac{\boldsymbol{u_i^T u_j}}{d_i d_j} - \lambda \widetilde{y}_{ij})$, our objective becomes a linear combination of the kernel functions $k_{ij} = k(W^Tx_i, W^Tx_j)$ with the coefficients γ_{ij}.

Algorithm 1 outlines the two optimization steps in KDAC. We use the dimension growth algorithm [10] to optimize W, which is also the performance bottleneck of KDAC, as we observed in experiments described in Section VI. We describe in detail how the dimension growth algorithm works with an example in the next section.

Algorithm 1 Kernel Dimension Alternative Clustering (KDAC)

Input: Data X, existing labeling Y, reduced dimension q, cluster number c,

1: Initialize subspace $W = I$
2: **while** W or U not converge **do**
3: **Step 1:** Calculate kernel matrix K and degree matrix D based on the subspace projected matrix XW. Matrix U then equals to the top c eigenvectors of $D^{-\frac{1}{2}}KD^{-\frac{1}{2}}$
4: **Step 2:** Given U, update W according to the **dimension growth algorithm**
5: **end while**
6: Assign points in U to c clusters using K-Means

Output: Alternative clustering U and transformation matrix W

C. Dimension Growth Algorithm

When optimizing the objective in Equation 5, we can consider γ_{ij} as a fixed constant, and we aim to find a subspace W such that the resulting kernel matrix K on that subspace maximizes the linear combination of its element k_{ij} and γ_{ij}. To find this W, we use the dimension growth algorithm where we optimize one column w_l of W at a time $(l = 0, 1, \ldots q-1)$, until all columns are optimized.

We use Gaussian kernel as an example to illustrate how to optimize W. With a Gaussian kernel, Equation 5 can be re-written as

$$\begin{aligned} &\max_W \ \sum_{ij} \gamma_{ij} \exp\left(-\frac{\|x_iW - x_jW\|^2}{2\sigma^2}\right) \\ =\ &\max_W \ \sum_{ij} \gamma_{ij} \exp\left(-\frac{\left\|\Delta x_{ij}^T W\right\|^2}{2\sigma^2}\right) \\ =\ &\max_W \ \sum_{ij} \gamma_{ij} \exp\left(-\frac{\Delta x_{ij}^T WW^T \Delta x_{ij}}{2\sigma^2}\right), \end{aligned} \quad (6)$$

where $\Delta x_{ij}^T WW^T \Delta x_{ij}$ is the l_2-norm of $x_i - x_j$ projected to subspace defined by W. Using the definition and cyclic property of trace, we can rewrite Equation 6 as:

$$\begin{aligned} &\max_W \ \sum_{ij} \gamma_{ij} \exp\left(-\frac{\operatorname{Tr}(W^T \Delta x_{ij} \Delta x_{ij}^T W)}{2\sigma^2}\right) \\ =\ &\max_W \ \sum_{ij} \gamma_{ij} \exp\left(-\frac{w_0^T A_{ij} w_0 + w_1^T A_{ij} w_1 + \ldots}{2\sigma^2}\right) \\ =\ &\max_W \ \sum_{ij} \gamma_{ij} \exp\left(-\frac{w_0^T A_{ij} w_0}{2\sigma^2}\right) \exp\left(-\frac{w_1^T A_{ij} w_1}{2\sigma^2}\right) \ldots, \end{aligned} \quad (7)$$

where A_{ij} represents the matrix $\Delta x_{ij}\Delta x_{ij}^T$. Using the dimension growth algorithm, we optimize w_0 first, with the objective being: $\max_{w_0} \sum_{ij} \gamma_{ij} \exp\left(-\frac{w_0^T A_{ij} w_0}{2\sigma^2}\right)$. We initialize w_0 by random projection and normalize it to have unit norm, and then we use gradient ascent method to maximize the objective.

Once w_0 is optimized, it is fixed and absorbed into the coefficient terms. We move on to optimize w_1, with an updated objective:

$$\max_{w_1} \sum_{ij} \gamma_{ij} g(w_0) \exp\left(-\frac{w_1^T A_{ij} w_1}{2\sigma^2}\right), \tag{8}$$

where $g(w_0)$ is $\exp\left(-\frac{w_0^T A_{ij} w_0}{2\sigma^2}\right)$. We first initialize w_1 with random projection, then project it to the space orthogonal to w_0, and at last normalize it to have unit norm. Denoting the gradient of w_l as ∇f, we can calculate ∇f as follows:

$$\nabla f = \sum_{ij} -\gamma_{ij} \frac{1}{\sigma^2} g(w_0) \exp\left(-\frac{w_1^T A_{ij} w_1}{2\sigma^2}\right) A_{ij} w_1 \tag{9}$$

∇f is then decomposed into two parts:

$$\nabla f = \nabla f_{proj} + \nabla f_{\perp}, \tag{10}$$

where ∇f_{proj} is the projection of ∇f onto the space spanned by w_0 and w_1, leaving the second term $\nabla f_{\perp}$ orthogonal to ∇f_{proj}. $\nabla f \perp$ is also normalized afterwards. We use a modified gradient ascent method to update w_1 with the following equation:

$$w_{1,new} = \sqrt{1-\alpha^2} w_{1,old} + \alpha \nabla f_{\perp}, \tag{11}$$

where α is the step length. Note that this differs from traditional gradient ascent method as we need to ensure w_1 always has unit norm and is orthogonal to w_0, therefore we apply $\sqrt{1-\alpha^2}$ to the old w_1 term and use $\nabla f_{\perp}$ as the search direction. To find the proper step length α, a line search method is used to make sure the following Wolfe condition is satisfied [11]:

$$\Phi(\alpha) \geq \Phi(0) + a_1 \alpha \Phi'(0), \tag{12}$$

where

$$\Phi(\alpha) = \sum_{ij} \gamma_{ij} g(w_0) \exp\left(-\frac{w_{1,new}^T A_{ij} w_{1,new}}{2\sigma^2}\right), \tag{13}$$

$0 < a_1 < 1$, and $\Phi'(0) = \nabla f^T \nabla f_{\perp}$. We start with $\alpha = 1$, and update α by multiplying it with 0.8 in each iteration, until the proper α that satisfies the Wolfe condition is found. We then update w_1 according to Equation 11, and repeat 9, 10, and 11 until w_1 is converged. After w_0 and w_1 are optimized, we update the remaining w_j (where $j = 2, \ldots q-1$) the same way. Because we use a modified gradient ascent method (Equation 7) to update each w_l, all q columns remain orthonormal to each other throughout the process, satisfying the constraint imposed by $W^T W = I$.

Algorithm 2 outlines the dimension growth algorithm. In the next section, we discuss the complexity of KDAC algorithm and identify the performance bottleneck.

Algorithm 2 Dimension Growth Algorithm

Input: Projection matrix W, learning rate α

1: **for** Each column w_l in W **do**
2: **if** w_l is the first column **then**
3: Initialize w_l by random projection and normalize it
4: Optimize w_l using gradient ascent
5: **else**
6: Initialize w_l by random projection
7: Project w_l to the space orthogonal to all previous columns ($\{w_0, w_1, ...w_{l-1}\}$) and then normalize w_l
8: **while** w_l does not converge **do**
9: Calculate the gradient of w_l: ∇f
10: Calculate and normalize $\nabla f_{\perp}$, which is orthogonal to the space spanned by all $l+1$ columns: $\{w_0, w_1, ...w_l\}$
11: Initialize α, a_1, $\Phi(0)$, $\Phi'(0)$, and $\Phi(\alpha)$
12: **while** $\Phi(\alpha) < \Phi(0) + a_1 \alpha \Phi'(0)$ **do**
13: Update α: $\alpha = \alpha \times 0.8$
14: Update $\Phi(\alpha)$
15: **end while**
16: Update w_l: $w_l = \sqrt{1-\alpha^2} w_l + \alpha \nabla f_{\perp}$
17: **end while**
18: **end if**
19: **end for**

Output: Optimized W

Table I: Complexity of Optimizing U

Computation	XW	Generate K	$D^{-\frac{1}{2}} K D^{-\frac{1}{2}}$	Eigen-decomp
Compexity	$\mathcal{O}(ndq)$	$\mathcal{O}(n^2 d)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$

D. Complexity Analysis

Algorithm 1 outlines the overall steps in KDAC. The overall complexity of the KDAC algorithm is the sum of $\text{Complexity}_{\text{U}}$ and $\text{Complexity}_{\text{W}}$. We analyze $\text{Complexity}_{\text{U}}$ and $\text{Complexity}_{\text{W}}$ in the following paragraphs.

The complexity breakdown of generating different components when optimizing matrix U is summarized in Table I. Note that calculating matrix $D^{-\frac{1}{2}} K D^{-\frac{1}{2}}$ only incurs $\mathcal{O}(n^2)$ operations because we use $\frac{k_{ij}}{d_i d_j}$ instead of matrix multiplication to calculate the elements in the matrix. The overall complexity of optimizing matrix U is then:

$$\text{Complexity}_U = \mathcal{O}(ndq + n^2 d + n^2 + n^3) = \mathcal{O}(n^3) \tag{14}$$

where the eigen-decomposition is the bottleneck.

When it comes to optimizing W, the algorithm introduces a three-level nested loops at line 1, 8, and 12 in Algorithm 2. We denote these three locations as l_1, l_2 and l_3. The number of iterations at l_1 is q as we are optimizing one column w_l in one iteration and there are q columns in W. In contrast, the number of iterations at l_2 is dependent on the convergence rate of the dimension growth algorithm, while the number of iterations at l_3 equals to the number of trials before we find

Table II: Complexity of Optimizing W

Location	l_1	l_2				l_3
# Iterations	q	t_2				t_3
Computation	w_l	∇f	$\nabla f_\perp$	$\Phi(0)$	$\Phi'(0)$	$\Phi(\alpha)$
Compexity	$\mathcal{O}(dq)$	$\mathcal{O}(n^2d^2)$	$\mathcal{O}(dq)$	$\mathcal{O}(n^2d)$	$\mathcal{O}(d)$	$\mathcal{O}(n^2d)$

the proper α that satisfies the Wolfe condition. Therefore, we use variable t_2 and t_3 to denote the number of iterations at l_2 and l_3, respectively. The complexity of optimizing W is described in Table II.

Note that even though calculating ∇f has the highest complexity of $\mathcal{O}(n^2d^2)$, we need to take into account the number of times the calculation is executed before determining if it is the computational bottleneck. Because ∇f is in the second level of the nested loop, it is calculated for qt_2 times, therefore the total complexity of calculating ∇f becomes $\mathcal{O}(t_2n^2d^2q)$. Because the calculation of $\Phi(\alpha)$ happens in the innermost loop, its complexity is $\mathcal{O}(t_2t_3n^2dq)$, therefore it could be equal to or higher than the complexity of ∇f,when $t_3 \geq d$. To conclude, the overall complexity of optimizing W is:

$$\text{Complexity}_W = \mathcal{O}(t_2n^2d^2q + t_2t_3n^2dq), \tag{15}$$

Comparing 14 and 15, we can see that the computational bottleneck in KDAC lies in calculating Equation 9 or Equation 12, which is also confirmed in our experiments described in Section VI. In next section, we will describe how to offload these bottlenecks to a GPU for accelerated and parallel execution.

IV. Implementation Details of KDAC

A. Removing redundant computation

In this section, we discuss the GPU optimization techniques used to accelerate the computationally intensive parts of the KDAC algorithm. As discussed in the last section, the bottleneck of KDAC lies in calculating ∇f and $\Phi(\alpha)$, as shown in Equations 9 and 12, respectively. These two equations share many common components; once a component is calculated in one equation, t can be reused in the other. This reuse of existing results avoids a number of redundant computations, and significantly improves performance. For example, in calculating $\Phi(\alpha)$, when we enter the last iteration of loop $l3$ (i.e., when the proper α that satisfies the Wolfe condition is found), the updated $w_{1,new}$ is plugged right back into Equation 9 to calculate the gradient. As a result, the kernel function term $\exp\left(-\frac{w_{1,new}^T A_{ij} w_{1,new}}{2\sigma^2}\right)$ in Equation 12 is equivalent to the term $\exp\left(-\frac{w_1^T A_{ij} w_1}{2\sigma^2}\right)$ in Equation 9, calculated later. We will refer to this exponential term as e_{ij} for the rest of the paper. In addition, the matrix-vector multiplication term $A_{ij}w_1$ in Equation 9 is also partially calculated in Equation 12. We can use temporary data structures in our program to save these redundant values to remove unnecessary computations. As a result, the complexity to optimize W in Equation 15 is reduced to:

$$\text{Complexity}_W = \mathcal{O}(t_2n^2dq + t_2t_3n^2dq) = \mathcal{O}(t_2t_3n^2dq) \tag{16}$$

The bottleneck now shifts entirely to calculating $\Phi(\alpha)$. We describe our GPU optimization techniques in the next section to tackle this bottleneck.

B. Offloading computational bottleneck to GPU

We offload the most compute-intensive portion of Equation 12, $e_{ij} = \exp\left(-\frac{w_k^T A_{ij} w_k}{2\sigma^2}\right)$, where $k = 0, \ldots, q-1$, to the GPU. Because $A_{ij} = \Delta x_{ij} \Delta x_{ij}^T$, we can rewrite the term as:

$$\exp(-\frac{w_k^T \Delta x_{ij} \Delta x_{ij}^T w_k}{2\sigma^2}) = \exp(-\frac{(w_k^T \Delta x_{ij})^2}{2\sigma^2}) \tag{17}$$

Calculating Equation 17 is equivalent to calculating a d-dimensional vector-vector dot product for each (i, j) pair, where $i, j = 0, \ldots, n-1$. Because each dot product can be calculated independently, we launch $n \times n$ blocks in a two-dimensional grid, where we calculate $w_k \Delta x_{ij}$ terms all in parallel.

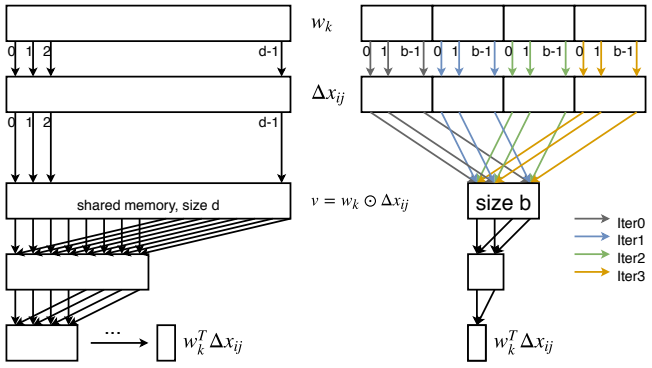


Figure 1: Basic kernel vs. our improved kernel

The left-hand side of Figure 1 lists a simple version of the GPU kernel for computing $w_k^T \Delta x_{ij}$. We denote this kernel as *kernel*$_1$ in the rest of the paper. The two input vectors to this kernel are w_k and Δx_{ij}. w_k changes each time a new α is applied in the line search algorithm, which requires memory transfers for each kernel invocation; in contrast, Δx_{ij} remains the same throughout the KDAC algorithm. We pin Δx_{ij} in the GPU's device memory. Note that instead of storing all n^2 Δx_{ij} in GPU memory, we only store the input matrix X, and compute Δx_{ij} on the fly every time when it is needed. For simplicity, we assume single-precision data is used when discussing memory sizes for the remainder of this paper. Only storing matrix X saves $4(d-1)n^2$ bytes of GPU memory and significantly improves the program's scalability when working with larger datasets.

An arrow in Figure 1 represents a thread, which is labeled by its thread ID $tid = 0, \ldots, d-1$. The number of threads

per block (block size b) in *kernel*$_1$ is equal to d, indicating that one thread reads two elements — one from each input vector, multiplies them, and writes the result to another vector $v = w_k \odot \Delta x_{ij}$. We use a parallel reduction [12] to sum up v, producing the final result $w_k^T \Delta x_{ij}$. Because a parallel reduction makes repeated accesses to vector v, we use shared memory to store the vector, which significantly reduces the access latency compared to using global memory. The size of the shared memory is denoted as s. In *kernel*$_1$, $s = d$, where the shared memory reserves a spot for every element of v. *kernel*$_1$ is straightforward and easy to implement. But *kernel*$_1$ has two shortcomings: 1) it does not efficiently utilize GPU memory and compute resources, and 2) it does not scale well when d increases. We will discuss these two problems in detail, and present an improved version of *kernel*$_1$.

C. Efficiency

For a GPU program to benefit from using shared memory, the data stored in shared memory must be accessed multiple times, so that the one-time cost of loading data from global memory to shared memory is amortized. In *kernel*$_1$, however, after the element-wise multiplication is completed, elements from $v[\frac{d}{2}]$ to $v[d-1]$ are loaded into shared memory, only to be used once during the first step of the parallel reduction. As a result, allocating half of shared memory (a precious memory resource on the GPU) results in no performance benefits at all. In addition to memory inefficiency, half of the threads are idle after v is generated, as only the first half of the threads need to execute the add operation in the parallel reduction. The compute-to-memory-access ratio for *kernel*$_1$ is low.

D. Scalability

Another shortcoming of the *kernel*$_1$ is its lack of scalability. Because both b and s have an upper bound in the CUDA programming model (e.g., $b = 1024, s = 12K$ on a majority of current GPUs, *kernel*$_1$ cannot be used for the input matrix when $d > 1024$. Even though we can resolve this issue by oversubscribing each thread (assigning multiple elements to one thread, in an element-wise multiplication), the limited size of s still requires $d < 12K$.

E. Improved kernel

The right-hand side of Figure 1 presents an improved version of the kernel — *kernel*$_2$. We set $b = \min(\lceil \frac{d}{2} \rceil, 256)$, where the ceiling function returns the smallest power of 2 that is greater than or equal to $\frac{d}{2}$; e.g. $b = 128$ when $d = 256$, meaning that each thread needs to compute the element-wise multiplication for 2 elements from each input vector. In the example shown in Figure 1, we have $d = 1024$ and $b = 256$. This happens when $\lceil \frac{d}{2} \rceil$ is larger than 256 — the upper bound of b; each thread in this case is assigned 4 elements from each input vector. In the first iteration, all b threads read the first b elements in parallel from both input vectors, multiply them, and write b products to v in shared memory. In the second iteration, b threads process the second b elements from the input, and update v by adding the outcomes to the existing values. After all 4 iterations are completed, a parallel reduction is performed on a vector of v that is only one fourth as large as the one used in *kernel*$_1$.

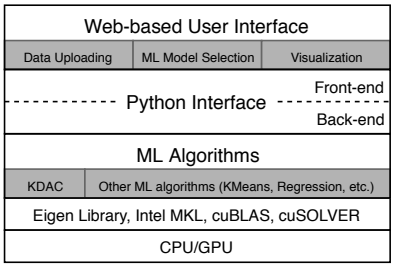
kernel$_2$ improves memory and compute efficiency, as shared memory is fully utilized and the number of idle threads is reduced, thus increasing the compute-to-memory-access ratio. More importantly, *kernel*$_2$ is scalable because the maximum value for both b and c is 256. b is well within the limit imposed by the CUDA programming model, and the shared memory size per block is maximumly 1024 bytes, allowing as many as 48 blocks to run simultaneously per SMX, without exhausting for shared memory space. In the next section, we will present our software stack that provides a web-based front-end for interactive data clustering visualization, and a high performance back-end that integrates our KDAC implementation.

V. NICE Framework

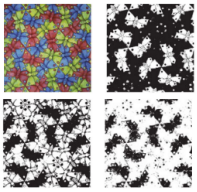
NICE (Northeastern Interactive Clustering Engine) [13] is a data analytics framework that provides interactive data visualization/analysis for large datasets. As shown in Figure 2a, the NICE framework provides users with a web-based interface as the front-end, while maintains an efficient and performant back-end. The back-end utilizes the Eigen library and the Intel Math Kernel Library (MKL) for CPU-based algorithms. The Eigen library provides a simple interface for matrix-based data structures and linear algebra operations (e.g., eigen-decomposition in KDAC), and MKL uses SSE instructions to exploit data-level parallelism for these operations. For GPU-based algorithms, CUDA libraries such as cuBLAS and cuSOLVER are integrated. The front-end is written in Python-based development tools. The C++-based back-end communicates with the front-end via the Boost.Python library, which maps data objects in C++ directly to Python with zero-copy memory. The framework is developed following software engineer principles; we use CMake for automated cross-platform compilation, and create numerous unit and integration tests with Google Test. The NICE framework allows researchers from different domains to test different machine learning algorithms on their data and see the results interactively. Figure 2b shows an example of the visualization of three different cluster solutions for one input dataset in the framework.

VI. Experiments

In this section, we describe our experiments of applying KDAC to datasets with different sizes and patterns, and evaluate the performance of KDAC on both CPUs and GPUs.



(a) Software Stack



(b) Three different clustering solutions for one input dataset.

Figure 2: NICE framework

Table III: System Configuration

Platform	Kepler	Maxwell
CPU	Intel Xeon E5-2630 V3	Intel Core i7-4790K
# of CPU Cores	8	8
Frequency	2.4GHz	4.0 GHz
System Memory	32GB	16 GB
GPU	NVIDIA Tesla K40	NVIDIA GTX 970
# of SMX	15	13
# of CUDA Cores	2880	1664
Device Memory	12 GB	4 GB

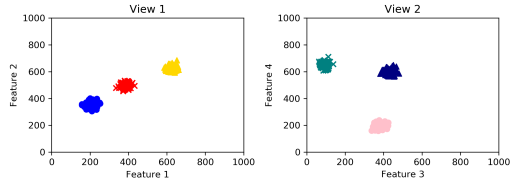
A. Experimental Setup

We perform experiments on two GPU architectures: Kepler (K40) and Maxwell (GTX 970). Table III lists the configuration of the platforms.

B. Datasets

1) Synthetic Gaussian Datasets: To test KDAC, we need to generate an $n \times d$ input dataset with 2 views, where we use the first view as the existing labeling Y as the base to find the alternative solution. We first generate k Gaussian clusters in the first $\lceil \frac{d}{3} \rceil$ space, where each cluster contains $\frac{n}{k}$ samples. Then we generate k additional Gaussian clusters in the second $\lceil \frac{d}{3} \rceil$ space, while leaving the remaining $d - 2\lceil \frac{d}{3} \rceil$ space with Gaussian noise. Figure 3 shows an example of 300 samples of 6 dimensions. Three Gaussian clusters are generated using the features $\{F_1, F_2\}$, and another three clusters are generated using the features $\{F_3, F_4\}$.

Figure 3: Synthetic Gaussian Dataset



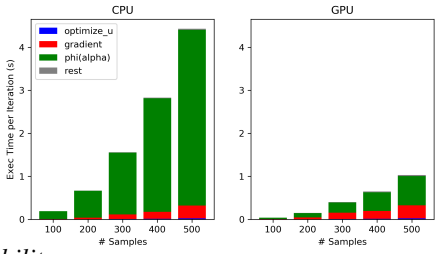
2) Real-world Datasets: The WebKB dataset [14] is a sub-sample of 1041 html documents from four universities. These web pages can be alternatively labeled as from four topics: courses, faculty, projects and students. We use the university information as features for the initial clustering and the web page topics for the alternative clustering, which we use to show the power of KDAC. Note that the size of X is only 1041×139 for this dataset. We use this dataset to validate the robustness of our implementation, while using the synthetic datasets to test the performance and scalability.

C. Bottlenecks

Before implementing the KDAC algorithm on a GPU, we first perform a preliminary performance evaluation on KDAC's CPU implementation. This helps us identify any inherent bottlenecks in the program. The left-hand side of Figure 4 shows the performance of running KDAC on the CPU of the Maxwell platform, using the synthetic datasets, where n is increased from 100 to 500, $d = 100$, and $k = 3$. Note that the execution time on the vertical axis is the average time across an iteration in the outermost loop, as shown in Line 2 in Algorithm 1.

As observed in the graph, the time to compute both the $\Phi(\alpha)$ and the gradient ∇f increases quadratically with n, confirming our complexity analysis of the n^2 term in Equation 16. Even though optimizing the matrix U has a complexity of $\mathcal{O}(n^3)$ for eigen-decomposition, the fact that it only occurs in the outermost loop, where its complexity does not scale by $t_2 t_3$, clearly makes it less of a concern when it comes to improving the overall performance. We focus on offloading the computation of $\Phi(\alpha)$ to the GPU for parallel execution and obtain the results shown on the right-hand side of Figure 4. Results show that, on average, we achieve a $5.8x$ speedup when computing $\Phi(\alpha)$.

Figure 4: Execution time breakdown on the synthetic datasets, while scaling dataset sizes of n, run on a CPU versus a GPU.



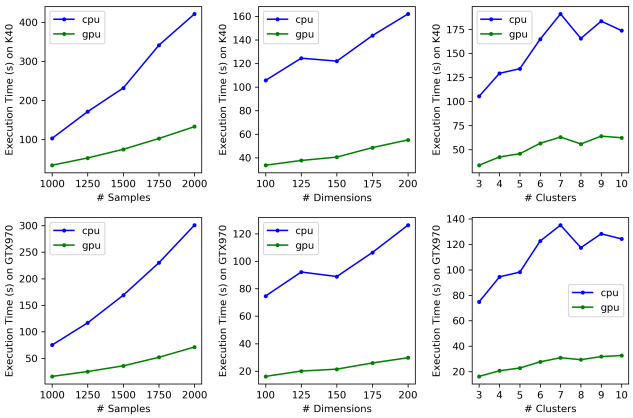
D. Scalability

In this section, we report on accelerated-KDAC's scalability on the synthetic Gaussian datasets with increasing n, d, and k.

We present results on both platforms. As shown in Figure 5, both the CPU and the GPU implementations achieve good scalability with increased n, d, and k on both platforms, while the GPU consistently outperforms the CPU, with an average speedup of $3.1x$ on the Kepler platform and a speedup of $4.4x$ on the Maxwell platform. Note that the execution time has a perfect quadratic relationship with n. The linear relationship with d is less consistent, as the data dimension could affect the convergence rate.

When we arbitrarily set k, which does not conform to the ground truth of the real data, the execution time can be non-deterministic — when n and k are fixed, the convergence rate of the dimensional growth algorithm determines the overall performance.

Figure 5: Execution time of KDAC on the synthetic datasets, while varying n, d, and k.



We also apply KDAC to the WebKB data on both platforms. The Kepler platform achieves a $3x$ speedup, while the Maxwell achieves a $4.1x$ speedup. This result is consistent with our observations on the synthetic datasets.

VII. Conclusion and Future Work

In this work, we study KDAC, a clustering algorithm that automatically provides complementary clustering solutions automatically, valuable to various data analytics tasks. We identify the performance bottlenecks of KDAC and implement a GPU-accelerated version on a big data analysis framework that provides users with ability to interactively visualize clustering solutions. We also conduct a thorough performance evaluation i using both synthetic data and real-world datasets.

For future work, we plan to further explore other methods of optimizing KDAC, including offloading the whole W optimization step to the GPU. This should reduce the memory copy overhead. We also would like to study and accelerate a new alternative clustering solution — Iterative Spectral Method (ISM) [15]. ISM does not rely on a dimensional growth algorithm, and has a faster convergence rate, which should be more suitable for an interactive data clustering environment.

VIII. Acknowledgements

This work has been supported in part by NSF BIG-DATA grant BIGDATA-1546428 and by NIEHS grant P42ES017198.

References

[1] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern recognition letters*, vol. 31, no. 8, pp. 651–666, 2010.

[2] D. Niu, J. G. Dy, and M. I. Jordan, "Iterative discovery of multiple alternativeclustering views," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 7, pp. 1340–1353, July 2014.

[3] F. Azmandian, A. Yilmazer, J. Dy, J. Aslam, and D. Kaeli, "GPU-accelerated feature selection for outlier detection using the local kernel density ratio," in *Proceedings of the IEEE 12th International Conference on Data Mining (ICDM)*, 2012, pp. 51–60.

[4] W.-m. W. Hwu, *GPU Computing Gems Emerald Edition*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[5] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–11.

[6] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in neural information processing systems*, 2002, pp. 849–856.

[7] A. Gretton, O. Bousquet, A. Smola, and B. Schölkopf, "Measuring statistical dependence with hilbert-schmidt norms," in *Proceedings of the 16th International Conference on Algorithmic Learning Theory*, ser. ALT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 63–77.

[8] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing with OpenCL 2.0*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.

[9] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[10] D. Niu, J. Dy, and M. Jordan, "Dimensionality reduction for spectral clustering," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 552–560.

[11] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York, NY, USA: Springer, 2006.

[12] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *ACM SIGGRAPH 2008 classes*. ACM, 2008, p. 16.

[13] M. Jorgensen, J. Spohn, C. Bunn, S. Dong, X. Li, and D. Kaeli, "An interactive big data processing/visualization framework," in *2017 IEEE MIT Undergraduate Research Technology Conference (URTC)*, Nov 2017, pp. 1–4.

[14] "Webkb data," http://www.cs.cmu.edu/ webkb/, 1997.

[15] C. Wu, S. Ioannidis, M. Sznaier, X. Li, D. Kaeli, and J. Dy, "Iterative spectral method for alternative clustering," in *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, A. Storkey and F. Perez-Cruz, Eds., vol. 84. Playa Blanca, Lanzarote, Canary Islands: PMLR, 09–11 Apr 2018, pp. 115–123. [Online]. Available: http://proceedings.mlr.press/v84/wu18a.html