CUDA-sDBSCAN: GPU Accelerated Density-Based Clustering With Random Projections

Hugo Phibbs
hphi344@aucklanduni.ac.nz
Repository: `https://github.com/HugoPhibbs/CUDA-sDBSCAN`

November 2024

Abstract

CUDA-sDBSCAN is a GPU-accelerated adaptation of sDBSCAN, designed to address the scalability challenges of density-based clustering in high-dimensional, large-scale datasets. By combining random projections with the parallel computation power of GPUs, CUDA-sDBSCAN significantly reduces runtime while maintaining clustering accuracy. Our implementation demonstrates at least a $2\times$ speedup compared to its CPU counterpart, especially for larger datasets. Through detailed experiments, we showcase the algorithm's scalability, efficiency, and potential for applications in data-intensive fields. This paper also identifies areas for further optimization, such as custom CUDA kernels and support for diverse GPU architectures, paving the way for future advancements in using GPU hardware for clustering applications.

1 Introduction

DBSCAN [18] is a density-based clustering algorithm with applications in machine learning, data mining and biology [13]. In its simplest form, DBSCAN aims to partition a d-dimensional dataset X containing n points into groups called clusters - where each cluster is ideally an isolated region of densely packed points. DBSCAN's applicability has stood the test of time [2] - appearing in a plethora of data analysis tools such as ELKI [16], scikit-learn [17], R [25] and Weka [51].

Despite DBSCAN's widespread usage, its scalability remains a constraining issue [50]. For instance, DBSCAN has a worst-case runtime of $\mathcal{O}(n^2 \cdot d)$ [24] [1], and a lower bound of $\Omega(n^{4/3})$ for Euclidean DBSCAN with $d \geq 3$ [19]. Consequently, significant research efforts have focused on enhancing DBSCAN's scalability through two notable strategies: reducing the overall computational load via approximate methods and leveraging GPU-based parallelisation to address primary computational bottlenecks.

This paper presents CUDA-sDBSCAN, an approximate DBSCAN implementation that utilises random projections and GPU acceleration. CUDA-sDBSCAN is primarily based on adapting the random projections based sDBSCAN developed by Xu et al. [56]. The contributions of this paper are as follows:

1. We present CUDA-sDBSCAN, a GPU-accelerated variant of sDBSCAN, specifically optimised to reduce computational load through random projections and to leverage parallel GPU processing of large datasets.
2. Through experimental evaluation, we demonstrate that CUDA-sDBSCAN achieves substantial speed-ups over sDBSCAN while maintaining comparable clustering accuracy, especially on high-dimensional datasets.
3. We establish foundational work for future advancements, including enhancing CUDA-sDBSCAN's compatibility across diverse GPU architectures and improving its scalability for larger datasets.

[1] Note, the original authors give $\mathcal{O}(n^2)$, however, $\mathcal{O}(n^2 \cdot d)$ is more accurate for large scale datasets where d is non-trivially small compared to n.

The paper is organized as follows: We review the literature surrounding CUDA-sDBSCAN and survey recent advancements. Next, we provide the background and preliminary concepts for understanding our algorithm. Following this, we delve into a comprehensive explanation of CUDA-sDBSCAN's inner mechanisms. We then evaluate CUDA-sDBSCAN's performance across several datasets and conclude with discussions on potential future work and final remarks.

2 Related Work

A primary weakness of a naïve implementation of DBSCAN is its relative in-scalability for large-scale datasets [11, 34]. Consequently, an extensive research area has emerged to enhance DBSCAN's scalability for big data applications. We will provide a brief literature review of topics relating to our research direction.

2.1 Accelerated CPU-based DBSCAN

Several key strategies have emerged to accelerate DBSCAN, particularly for algorithms running primarily on CPUs. One prominent avenue involves spatial indexing to speed up the ε-neighbourhood search. Notable techniques in this domain include R-Trees [6], grid-based methods [7, 55], and k-d trees [57].

Another significant advancement comes from approximate methods that trade off some accuracy to reduce DBSCAN's time complexity. Sampling approaches are central to this, as demonstrated by sngDBSCAN [32], DBSCAN++ [29], and ρ-approximate DBSCAN [20]. A particularly innovative technique is the *leaders* used by ROUGH-DBSCAN to yield similar results to an exact DBSCAN in linear time [53].

2.2 GPU Accelerated DBSCAN

Other approaches to scaling up DBSCAN have been using the power of GPUs (Graphical Processing Units) to speed up DBSCAN, utilising the power of parallel computation. Due to their suitability for parallel tasks, GPUs are particularly well aligned with DBSCAN's requirements for numerous independent distance computations, which GPU architectures can efficiently handle. G-DBSCAN [4], introduced by Andrade et al. in 2013, represents an early attempt at GPU-based DBSCAN, utilising a relatively straightforward method of parallelising most algorithm components. Although G-DBSCAN marked significant progress, it has proven inadequate for large-scale datasets due to substantial memory demands [47].

More recent algorithms, such as CUDA-DClust+ [47], have surpassed G-DBSCAN by integrating spatial indexing with highly optimised kernel fusion, thereby reducing CPU-GPU communication overhead. Additionally, RT-DBSCAN [37] introduces an innovative technique by leveraging ray-tracing (RT) hardware, demonstrating exceptional performance on low-dimensional datasets. It outperforms CUDA-DClust+ and G-DBSCAN on larger datasets due to its lower memory footprint. F-DBSCAN [48], in contrast, employs a tree-based method and, while competitive with other GPU-based algorithms, is ultimately surpassed by RT-DBSCAN in terms of speed [37].

Gowanlock et al. proposed HYBRID-DBSCAN [22], which combines both the CPU and the GPU interleaved. Here, spatial-based indexing combines with an efficient data-transfer batching scheme between the CPU and GPU. Its performance, however, has only been evaluated on low-dimensional data and has been untested against rival low-dimensional GPU DBSCAN algorithms.

Thus far, GPU-accelerated DBSCAN implementations have primarily been practical for low-dimensional data applications. This is due to the limitations of spatial indexing structures, which are ill-suited for high-dimensional spaces or excessive memory requirements. AC-DBSCAN [30], by contrast, leverages GPU power and is optimised specifically for high-dimensional data applications. It has demonstrated exceptional speed when applied to large-scale datasets, both in size and dimensionality.

3 Preliminaries

3.1 DBSCAN

DBSCAN [12] is a clustering algorithm based on grouping *densely* packed points based on a specified distance metric. Given a dataset, X, and a point $\mathbf{x} \in X$, the region around $\mathbf{x}$ is considered *dense* if there at least $minPts$

number of points within a ε distance of $\mathbf{x}$. The algorithm comprises two fundamental processes: classifying points identified as *core* and forming clusters based on these core points. To demonstrate, consider a set of points $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, where *dim*$(\mathbf{x}_i) = d$. Core point identification is completed as follows (formalised in Alg. 1):

1. Perform a ε-*neighbourhood search* for each data point, treating every data point as a query point: For each query point $\mathbf{q} \in X$, find the distance $\mathrm{dist}(\mathbf{x}, \mathbf{q})$ to all other data points $\mathbf{x} \in X$. If $\mathrm{dist}(\mathbf{x}, \mathbf{q}) \leq \varepsilon$, add $\mathbf{x}$ to the neighbourhood set of $\mathbf{q}$ - denoted by $B_\varepsilon(\mathbf{q})$. *Note*: $\mathrm{dist}(\mathbf{x}, \mathbf{q})$ can be an arbitrary distance measure [50] such as cosine distance, L^1/L^2 distance, or Jaccard Similarity [33].

2. For each $\mathbf{q} \in X$, if $|B_\varepsilon(\mathbf{q})| \geq minPts$, then $\mathbf{q}$ is a *core-point*. Add the tuple $(\mathbf{q}, B_\varepsilon(\mathbf{q}))$ to the core-point neighbourhood set C.

Once core points are identified, points are grouped into clusters or classified as *noise*. A cluster consists of points that are *density reachable* from each other. Two points are density reachable if they directly neighbour the same core point or can be connected through a chain of neighbouring core points. Any point that is not part of a cluster is classified as *noise*, and a point that is neither *noise* nor *core* is regarded as a *border* point.

The result of the DBSCAN algorithm can be represented as a graph $G(V, E)$, where each data point $\mathbf{x} \in X$ is a node. An edge $e_{i \to j} \in E$ connects two points $\mathbf{x}_i$ and $\mathbf{x}_j$ if they belong to the same cluster and are within ε distance of each other. The process of creating the DBSCAN graph $G(V, E)$ is formalised in Alg. 2. Fig. 1 shows the formation of a simple DBSCAN cluster.

Algorithm 1 DBSCAN, classification of core points

Input: Data points $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, ε, $minPts$
Output: The set $C = \{(\mathbf{q}, B_\varepsilon(\mathbf{q})) \,|\mathbf{q} \text{ is core}\}$ containing core points and their ε-neighbourhoods
Initialize $C = \emptyset$
for each data point $\mathbf{q} \in X$ **do**
    Initialize $B_\varepsilon(\mathbf{q}) = \emptyset$
    for each data point $\mathbf{x} \in X$ **do**
        if $\mathrm{dist}(\mathbf{x}, \mathbf{q}) \leq \varepsilon$ **then**
            Add $\mathbf{x}$ to $B_\varepsilon(\mathbf{q})$
        end if
    end for
    if $|B_\varepsilon(\mathbf{q})| \geq minPts$ **then**
        Add $(\mathbf{q}, B_\varepsilon(\mathbf{q}))$ to C
    end if
end for

Algorithm 2 DBSCAN, forming clusters

Inputs: $X, \varepsilon, minPts$ the set $C = \{(\mathbf{q}, B_\varepsilon(\mathbf{q})) \,|\, \mathbf{q} \text{ is core}\}$
$G \leftarrow$ initialise empty graph
for each $\mathbf{q} \in C$ **do**
    Add an edge (and possibly a vertex or vertices) in G from $\mathbf{q}$ to all *core* points in $B_\varepsilon(\mathbf{q})$
    Add an edge (and possibly a vertex) in G from $\mathbf{q}$ to *non-core* points $\mathbf{x} \in B_\varepsilon(\mathbf{q})$ if $\mathbf{x}$ is not connected
end for
return connected components of G

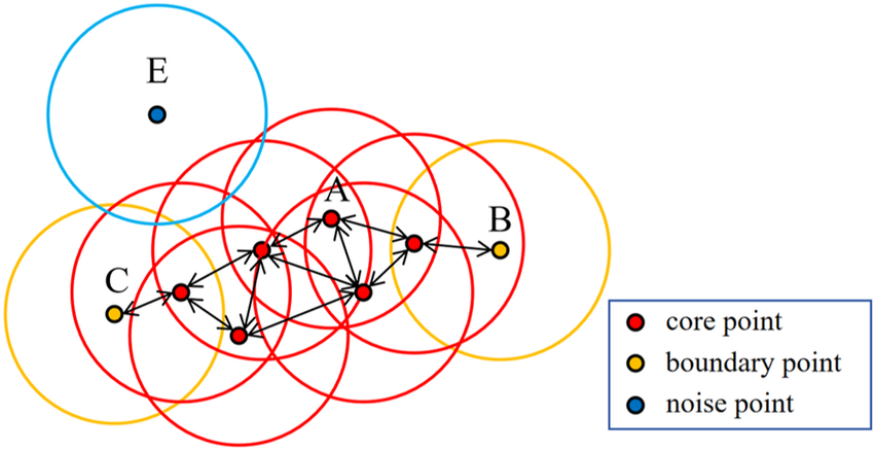


Figure 1: The formation of a DBSCAN cluster.
Cluster points are enclosed in the red circles. Source: [31]

3.2 sDBSCAN

sDBSCAN [56] is an implementation of DBSCAN based on using random projections to accelerate the ε-neighbourhood search for each data point. Besides the existing DBSCAN inputs of X, ε, and $minPts$, sDBSCAN adds a few more key parameters - these are described in Table 1. The sDBSCAN algorithm is differentiated from the naïve DBSCAN algorithm with its pre-processing and core points identification steps. Besides this, sDBSCAN's clustering step is almost identical to DBSCAN.

Parameter	Description
D	The number of random vectors sampled from the Gaussian distribution to generate. Vectors are stored in the set Y, where $Y = \{\mathbf{r}_i \mid \mathbf{r}_i \in \mathbb{R}^d,\ \mathbf{r}^i \sim \mathcal{N}(0,1),\ i \in [D]\}$
k	The number of closest/furthest random vectors from Y to store for each query $\mathbf{q} \in X$.
m	The number of closest/furthest query points from X to store for each random vector $\mathbf{r} \in Y$. Where $m = \mathcal{O}(minPts)$.

Table 1: Definitions and descriptions of sDBSCAN's parameters

3.2.1 Pre-processing

sDBSCAN uses random projections to perform an approximate nearest neighbour search for each query point. In essence, identifying approximately nearby points helps to conduct an *approximate* ε-neighborhood search for each query $\mathbf{q}$. More specifically, sDBSCAN relies on a technique called approximate maximum inner product search, known as CEOs [46]. This technique finds the approximate neighbours set $aNN(\mathbf{q})$ for a query point $\mathbf{q} \in X$ with the following steps:

1. Normalise $\mathbf{q}$ within the unit sphere using the L^2 norm, such that $||\mathbf{q}||_2 = 1$.
2. Project $\mathbf{q}$ onto D random vectors with entries sampled from the Gaussian distribution. This is done by taking the inner products to each random vector and storing them in a set S; Formally $S = \{\mathbf{q}\cdot\mathbf{r}_1, \mathbf{q}\cdot\mathbf{r}_2, \ldots, \mathbf{q}\cdot\mathbf{r}_D \mid \mathbf{r}_j \sim \mathcal{N}(0,1)\}$.
3. Search through S, and preserve the top k *closest* and *furthest* random vectors to $\mathbf{q}$ based on the inner products. A random vector $\mathbf{r}_j$ is considered close to a query $\mathbf{q}$ if the inner product $\mathbf{q}\cdot\mathbf{r}_j$ is large and positive. Conversely, a large negative inner product $\mathbf{q}\cdot\mathbf{r}_j$ indicates that $\mathbf{r}_j$ is further away from $\mathbf{q}$.
4. For each of the closest k random vectors $\mathbf{r}_j$ to $\mathbf{q}$, find its closest m dataset points by projecting $\mathbf{r}_j$ onto each point $\mathbf{x} \in X$. Store these closest m points in $aNN(\mathbf{q})$.

5. Similar to step 5: For each of the furthest k random vectors $\mathbf{r}_j$ to $\mathbf{q}$, find its furthest m dataset points by projecting $\mathbf{r}_j$ onto each point $\mathbf{x} \in X$. Store these furthest m points also in $aNN(\mathbf{q})$.

sDBSCAN repeats these previous steps to find the set of approximate nearest neighbours, $aNN(\mathbf{q})$, for all query points $\mathbf{q} \in X$. This finds $2km$ approximate nearest neighbours to each query. These pre-processing steps are formalised in Alg. 3. A geometric interpretation of using CEOs to find a close point $\mathbf{x}$ to a query $\mathbf{q}$ is illustrated in Fig. 2. In this example, CEOs selects $\mathbf{r}_1$ as a close random vector to the query $\mathbf{q}$. Consequently, $\mathbf{x}_1$ is identified as approximately close to $\mathbf{q}$, as its projection $\mathbf{x}_1 \cdot \mathbf{r}_1$ is larger than $\mathbf{x}_2 \cdot \mathbf{r}_1$ and $\mathbf{x}_3 \cdot \mathbf{r}_1$.

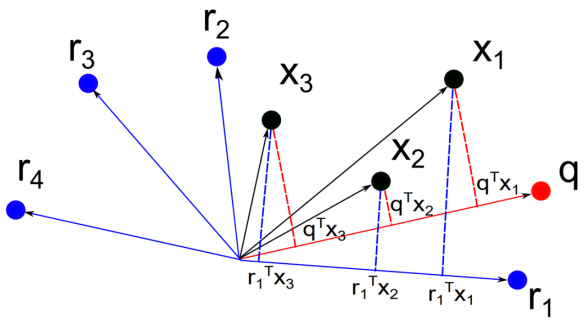


Figure 2: Geometric interpretation of CEOs. Note that this figure uses a different convention for the inner product: $\mathbf{r}_j^T \mathbf{x}_i$ is equivalent to $\mathbf{x}_i \cdot \mathbf{r}_j$.

Algorithm 3 pre-processing

1: **Inputs**: $X \subset \mathcal{S}^{d-1}$, D random vectors $\mathbf{r}_i$, $k, m = \mathcal{O}(minPts)$
2: **for** each $\mathbf{q} \in X$, compute and store top-k nearest and top-k farthest vectors $\mathbf{r}_i \in \mathbf{Y}$ to $\mathbf{q}$.
3: **for** each random vector $\mathbf{r}_i \in \mathbf{Y}$, compute and store top-$m$ nearest and top-m farthest points to $\mathbf{r}_i$.

3.3 Identifying core points and forming clusters

At this stage, the pre-processing steps enable an approximate ε-neighbourhood search for each query $\mathbf{q} \in X$. A naïve DBSCAN implementation calculates distances between each query point $\mathbf{q}$ and all other points $\mathbf{x} \in X$. In contrast, sDBSCAN limits this search to the set of approximate nearest neighbours, $aNN(\mathbf{q})$, which serve as *candidate points* for the query $\mathbf{q}$. This reduces the runtime complexity of finding the ε-neighbourhood for all points from $\mathcal{O}(n^2 \cdot d)$ to $\mathcal{O}(n \cdot dk \cdot minPts)$ [56] [2]. This process is formalised in Alg. 4

After completing the approximate ε-neighbourhood searches, apply standard DBSCAN clustering (Alg. 2) to the estimated core points and their neighbourhood sets $\widetilde{B}_\varepsilon(\mathbf{q})$ for each $\mathbf{q}$ identified as a core point (Alg. 5). Like the standard DBSCAN, sDBSCAN yields a resultant graph $G(V, E)$ of identified clusters.

[2] For most applications of sDBSCAN, $k \cdot minPts \ll n$

Algorithm 4 Finding core points and their neighbourhoods

Inputs: $X \subset \mathcal{S}^{d-1}$, Y, k, ε, $m = \mathcal{O}(minPts)$
Initialize an empty set $\mathbf{r_q}$ for each $\mathbf{q} \in X$
for each $\mathbf{q} \in X$ **do**
    for each $\mathbf{r}_i \in Y$ from top-k nearest random vectors of $\mathbf{q}$ **do**
        for each $\mathbf{x}$ from top-m nearest points of $\mathbf{r}_i$ **do**
            if $\mathrm{dist}(\mathbf{x}, \mathbf{q}) \leq \varepsilon$ **then**
                Insert $\mathbf{x}$ into $\widetilde{B}_\varepsilon(\mathbf{q})$ and insert $\mathbf{q}$ into $\widetilde{B}_\varepsilon(\mathbf{x})$
            end if
        end for
    end for
    for each $\mathbf{r}_i \in Y$ from top-k farthest random vectors of $\mathbf{q}$ **do**
        for each $\mathbf{x}$ from top-m farthest points of $\mathbf{r}_i$ **do**
            if $\mathrm{dist}(\mathbf{x}, \mathbf{q}) \leq \varepsilon$ **then**
                Insert $\mathbf{x}$ into $\widetilde{B}_\varepsilon(\mathbf{q})$ and insert $\mathbf{q}$ into $\widetilde{B}_\varepsilon(\mathbf{x})$
            end if
        end for
    end for
end for
for each $\mathbf{q} \in X$ **do**
    if $|\widetilde{B}_\varepsilon(\mathbf{q})| \geq minPts$ **then**
        Identify $\mathbf{q}$ as the core point
        Store $\widetilde{B}_\varepsilon(\mathbf{q})$ as an approximate result of $B_\varepsilon(\mathbf{q})$
    end if
end for

Algorithm 5 sDBSCAN

Inputs: $X \subset \mathcal{S}^{d-1}$, Y, ε, $minPts$
Call Alg. 3 for pre-processing with $m = \mathcal{O}(minPts)$
Call Alg. 4 to find the set
$C = \{(\mathbf{q}, \widetilde{B}_\varepsilon(\mathbf{q})) \,|\, \mathbf{q} \text{ is identified as core}\}$
Call DBSCAN (Alg. 2) given the output C from Algorithm 4

3.4 GPU Architecture and CUDA

A Graphical Processing Unit (GPU) is designed to leverage hundreds to thousands of independent cores to perform tasks efficiently in parallel. This approach excels in areas such as linear algebra, sorting algorithms, and solving differential equations [45]. To facilitate the programming of GPU-accelerated applications, NVIDIA, a major manufacturer of GPU chips, introduced the CUDA development platform [39]. In this section, we will explore the architecture of an NVIDIA GPU and its corresponding CUDA programming model.

3.4.1 GPU Architecture

Note that this section focuses on the architecture of a standard NVIDIA GPU; however, other manufacturers generally follow similar design paradigms in their GPU architectures.

A GPU is primarily composed of several *streaming multiprocessors* (SMs). Each SM groups together numerous *streaming processors* (SPs), often referred to as *CUDA Cores* for simplicity. Modern GPUs, such as the NVIDIA RTX 4080, typically contain between 50 and 100 SMs and thousands of CUDA Cores. For instance, the RTX 4080 has 80 SMs and 10,240 CUDA Cores [28]. A CUDA Core is the smallest hardware unit in a GPU and is responsible for executing a single instruction, known as a *thread*.

Memory management is critical to GPU performance, with caching being a key factor in optimising access speeds. For our purposes, we simplify GPU memory into two broad categories: SM memory and global memory.

SM memory refers to low-latency memory accessible by all threads running within the same SM, while global memory is high-latency memory shared across all SMs. Close attention must be given to the utilisation of these memory types. For instance, SM memory is typically orders of magnitude smaller than global memory—for example, the RTX 4080 has 128 KB of memory per SM, compared to 16 GB of global memory [52]. Consequently, SM memory is generally used for caching and small variable storage, while global memory is used for large-scale data storage. The concepts of SMs, CUDA cores and GPU memory are shown in Fig 3.

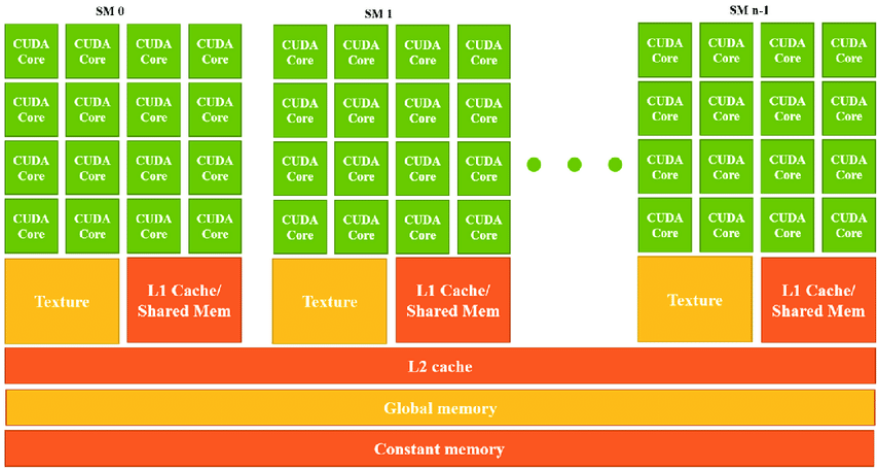


Figure 3: Simplified overview of typical GPU architecture [54].

3.4.2 CUDA Programming

For ease of development, the CUDA platform abstracts low-level GPU hardware into a high-level programming model. At the core of this model is the *kernel grid*, which executes a set of instructions called a *kernel* across the GPU. For example, a kernel might perform vector addition. A kernel grid is subdivided into *thread blocks*, each containing multiple *threads*. Multiple thread blocks can be executed concurrently on a single SM, though a single thread block cannot span multiple SMs. A thread executes one instance of a kernel on a CUDA Core. A graphical representation of these concepts is provided in Fig. 4.

Threads are executed in groups called *warps* [38], with each warp consisting of 32 threads. To maximise the occupancy of an SM during thread block execution, the number of threads per block should be a *multiple* of 32.

For illustration, consider the vector addition problem $\mathbf{u} + \mathbf{v} = \mathbf{w}$, where $dim(\mathbf{u}) = dim(\mathbf{v}) = 524288 = 1024 \times 512$. In this case, a kernel grid with 1024 blocks could be launched, with each block containing 512 threads. You can effectively think of one thread block as acting on a sequence of 512 indices within $\mathbf{u}$ and $\mathbf{v}$. Each thread would perform the operation $\mathbf{u}[i] + \mathbf{v}[i] = \mathbf{w}[i]$ for a global index $i \in [524288]$. An example pseudo-code for a kernel to solve this problem is presented in Alg. 6.

Algorithm 6 Vector Addition using CUDA

1: **Inputs**: Vector $\mathbf{u}$, Vector $\mathbf{v}$, Vector $\mathbf{w}$
2: **Define Kernel** `vectorAdd`($\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$):
3: Calculate global index i using block and thread indices
4: $\mathbf{u}[i] = \mathbf{v}[i] + \mathbf{w}[i]$
5: **End Kernel**

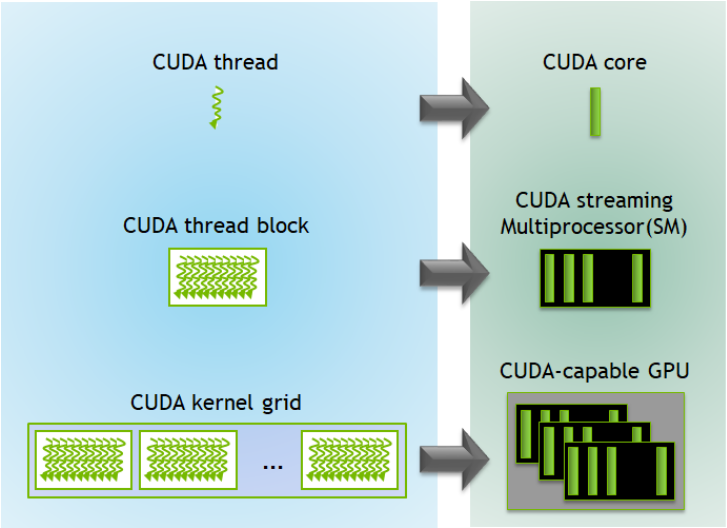


Figure 4: Simplified overview of the CUDA programming model [43]

4 CUDA-sDBSCAN

CUDA-sDBSCAN is an accelerated implementation of sDBSCAN using GPU hardware. It has the same underlying steps as the original CPU implemented sDBSCAN, except operations are transferred to the GPU when necessary. CUDA-sDBSCAN retains the same parameters as sDBSCAN and adds a few hyper-parameters to account for the use of a GPU. We break down CUDA-sDBSCAN along similar lines to sDBSCAN: pre-processing, finding approximate nearest neighbours, candidate distance computations, identification of core points and forming clusters. Once we outline the essential workings of CUDA-sDBSCAN, we will briefly discuss the use of batching techniques to enhance memory scalability.

4.1 Pre-processing

CUDA-sDBSCAN's pre-processing, like sDBSCAN's, aims to identify the approximate nearest neighbours for each query point $\mathbf{q} \in X$. However, CUDA-sDBSCAN differs from sDBSCAN in that it utilises a GPU to parallelise the random projections. We break the pre-processing steps into two sub-steps, which are performing random projections and finding approximate nearest neighbours.

4.1.1 Performing Random Projections

The pairwise projections between all dataset points $\mathbf{x}_i \in X$ and random vectors $\mathbf{r}_j \in Y$ can be computed efficiently using large matrix multiplication. This operation maps naturally to GPU hardware for parallel execution. The preparation for the pre-processing steps are as follows:

1. First, create a matrix $\mathbf{Y} \in \mathbb{R}^{d \times D}$ with entries sampled from $\mathcal{N}(0, 1)$. The shape of $\mathbf{Y}$ means that D random vectors of length d are stored *column-wise* within it.

2. Store the set of data points X in a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$. The shape of $\mathbf{X}$ means that the n dataset points with dimension d are stored *row-wise* within it.

3. If cosine distance is being used for distance computations, normalise $\mathbf{X}$ *row-wise* so that each data point stored along the rows has unit length. Otherwise, if other distance metrics such as Jaccard, L^1, L^2, or χ^2 are desired, embed $\mathbf{X}$ using randomized feature mappings[3]. Let the embedded or normalised form of $\mathbf{X}$ be $\widetilde{\mathbf{X}}$.

[3]See Section 4.4 from [56]

Now given the matrices $\mathbf{Y}$ and $\widetilde{\mathbf{X}}$, simply calculate the matrix product $\mathbf{P} = \widetilde{\mathbf{X}}\mathbf{Y} \in \mathbb{R}^{n \times d}$. By the rules of matrix multiplication, a given entry $\mathbf{P}_{ij}$ is simply the random projection $\tilde{\mathbf{x}}_i \cdot \mathbf{r}_j$. For the following steps, if cosine distance is being used as the distance metric, set $\mathbf{X} \leftarrow \widetilde{\mathbf{X}}$; otherwise, keep $\mathbf{X}$ unchanged.

4.1.2 Finding Approximate Nearest Neighbours

While the original CPU implementation of sDBSCAN relied on a priority queue to find $aNN(\mathbf{q})$ for each query $\mathbf{q}$, using the power of a GPU, we can make this process much simpler. We can *sort* the projections matrix $\mathbf{P}$ to find the top $2km$ candidate vectors. This is first along the rows to find the closest/furthest k random vectors for each data point, then likewise along the columns to find the closest/furthest m data points to each random vector. Since a GPU can sort individual rows or columns in parallel, this is a relatively trivial step. For ease of notation, we will define the lists Φ_i, $\overline{\Phi}_i$, Ω_j and $\overline{\Omega}_j$. The definitions of these lists are shown in Table 2.

Notation	Definition
Φ_i	The indices of the k *closest* random vectors in R to the dataset vector $\mathbf{x}_i$.
$\overline{\Phi}_i$	The indices of the k *furthest* random vectors in R to the dataset vector $\mathbf{x}_i$.
Ω_j	The indices of the m *closest* dataset points in X to the random vector $\mathbf{r}_j$.
$\overline{\Omega}_j$	The indices of the m *furthest* dataset points in X to the random vector $\mathbf{r}_j$.

Table 2: List definitions used for the pre-processing steps

To easily index large matrices when working with a GPU, the content of the above lists are modified and stored in the matrices $\mathbf{A} \in \mathbb{N}^{n \times 2k}$ (Eq. 1) and $\mathbf{B} \in \mathbb{N}^{2D \times m}$ (Eq. 2). The reasoning behind the structure of $\mathbf{A}$ and $\mathbf{B}$ will be more apparent later in this paper - essentially, they are composed this way to allow for easy tensor indexing using libraries such as PyTorch [5] or CuPy [44].

$$\mathbf{A} = \begin{bmatrix} 2\times\Phi_{1,1} & 2\times\Phi_{1,2} & \cdots & 2\times\Phi_{1,k} & 2\times\overline{\Phi}_{1,1}+1 & 2\times\overline{\Phi}_{1,2}+1 & \cdots & 2\times\overline{\Phi}_{1,k}+1 \\ 2\times\Phi_{2,1} & 2\times\Phi_{2,2} & \cdots & 2\times\Phi_{2,k} & 2\times\overline{\Phi}_{2,1}+1 & 2\times\overline{\Phi}_{2,2}+1 & \cdots & 2\times\overline{\Phi}_{2,k}+1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 2\times\Phi_{n,1} & 2\times\Phi_{n,2} & \cdots & 2\times\Phi_{n,k} & 2\times\overline{\Phi}_{n,1}+1 & 2\times\overline{\Phi}_{n,2}+1 & \cdots & 2\times\overline{\Phi}_{n,k}+1 \end{bmatrix} \tag{1}$$

$$\mathbf{B} = \begin{bmatrix} \Omega_{0,1} & \Omega_{0,2} & \cdots & \Omega_{0,m} \\ \overline{\Omega}_{0,1} & \overline{\Omega}_{0,2} & \cdots & \overline{\Omega}_{0,m} \\ \Omega_{1,1} & \Omega_{1,2} & \cdots & \Omega_{1,m} \\ \overline{\Omega}_{1,1} & \overline{\Omega}_{1,2} & \cdots & \overline{\Omega}_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ \Omega_{D,1} & \Omega_{D,2} & \cdots & \Omega_{D,m} \\ \overline{\Omega}_{D,1} & \overline{\Omega}_{D,2} & \cdots & \overline{\Omega}_{D,m} \end{bmatrix} \tag{2}$$

4.2 Candidate Distance Computations

Once the pre-processing steps have been completed, the distances between each query point $\mathbf{q}$ and their corresponding $2km$ candidate points in $aNN(\mathbf{q})$ can be found. This is done by forming the distance matrix $\mathbf{D} \in \mathbb{R}^{n \times 2km}$.

The matrices $\mathbf{A}$ and $\mathbf{B}$ are used to index the matrix $\mathbf{X}$ to find the candidate points for each query $\mathbf{q}$. Essentially, the i^{th} row in $\mathbf{A}$ corresponds to the data point $\mathbf{x}_i$. Conversely, the $(2j+b)^{th}$ (for $b \in \{0,1\}$) rows of $\mathbf{B}$ correspond to the random vector $\mathbf{r}_j$. For a given row of $\mathbf{A}$, the first k entries index within the *even* rows of $\mathbf{B}$, while the later k entries index the *odd* rows of $\mathbf{B}$. Alg. 7 gives the pseudocode for this indexing process.

Algorithm 7 Finding the indices of candidate points

Inputs: $\mathbf{A} \in \mathbb{N}^{n \times 2k}$, $\mathbf{B} \in \mathbb{N}^{2D \times m}$
  Initialize empty $\mathbf{IaNN} \in \mathbb{N}^{n \times 2km}$ ▷ $\mathbf{IaNN}$ stores the indices of approximate nearest neighbours *row-wise*

  for each $i \in \{0, 1, \ldots, n-1\}$ **do**
    $i_{col} \leftarrow 0$
    for each $i_A \in \{0, 1, \ldots, 2k-1\}$ **do**
      $a = \mathbf{A}[i, i_A]$
      for each each $i_B \in \{0, 1, \ldots, m\}$ **do**
        $\mathbf{IaNN}[i, i_{col}] \leftarrow \mathbf{B}[a, i_B]$
        $i_{col} \leftarrow \alpha + 1$
      end for
    end for
  end for
return $\mathbf{I}$

The matrices $\mathbf{A}$ and $\mathbf{B}$ are intended to be used with modern matrix libraries. Consequently, the matrix indexing equivalent of 7 is much simpler than it initially appears. Instead of using a nested for-loop approach, one can use matrix indexing shown in Equation 3.

$$\mathbf{IaNN} \leftarrow \mathbf{B}[\mathbf{A}] \tag{3}$$

Using this indexing trick, we can efficiently compute distance computations using tensor operations and hence find the matrix $\mathbf{D}$ - the pseudocode for this process is shown in Alg. 8.

A key strength of CUDA-sDBSCAN is its flexibility with distance metrics. To use a custom metric, define a custom dist function as used in Alg. 8. For instance, if cosine distance is desired, dist can be implemented as illustrated in Alg. 9.

Algorithm 8 Creation of the distances matrix $\mathbf{D}$

Inputs: $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{A} \in \mathbb{N}^{n \times 2k}$, $\mathbf{B} \in \mathbb{N}^{2D \times m}$
  Initialize an empty distances matrix $\mathbf{D} \in \mathbb{R}^{n \times 2km}$
  Retrieve candidate vectors $\mathbf{aNN} \leftarrow \mathbf{X}[\mathbf{B}[\mathbf{A}]]$ ▷ $\mathbf{aNN} \in \mathbb{R}^{n \times 2k \times m \times d}$
  Reshape $\mathbf{aNN}$ into $\mathbf{aNN} \in \mathbb{R}^{n \times 2km \times d}$
  Compute $\mathbf{D} \leftarrow \mathrm{dist}(\mathbf{X}, \mathbf{aNN})$ ▷ dist is an arbitrary distance function
return $\mathbf{D}$

Algorithm 9 Implementation of a dist function to find cosine distances

Inputs: $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{aNN} \in \mathbb{R}^{n \times 2km \times d}$
  Compute $\mathbf{T} \leftarrow \mathbf{aNN} \times \mathbf{X}[:, \text{newaxis}, :]$ ▷ $\mathbf{T} \in \mathbb{R}^{n \times 2km \times d}$. NumPy notation used
return $\mathbf{1}_{n \times 2km} - \mathrm{sum}(\mathbf{T}, \text{axis} = 2)$ ▷ Perform sum reduction along the 2nd axis

4.3 Identifying core points and forming clusters

Once the distances matrix $\mathbf{D}$ is found, core points are identified, and then clusters are formed. This stage of CUDA-sDBSCAN borrows heavily from the approach taken by Andrade et al.'s G-DBSCAN [21]. The strategy of this paper is to create a graph adjacency list [10] that can be used to form clusters with a breadth-first search (BFS).

An adjacency list can be thought of as a dense *flattened* representation of the more conventional adjacency matrix [1]. To demonstrate, take an undirected graph $G(V, E)$ composed of a set of vertices V ($|V| = n$) and a set of edges E. Define an adjacency list Ea, and the lists Da and Sa, where $|Ea| = |E|$ and $|Da| = |Sa| = n$. To then find the list of the adjacent vertices for the vertex V_i (denoted by $Ea(V_i)$), use the following steps:

1. Find Da_i, this is the *degree* of the vertex $v_i \in V$ in the graph $G(V, E)$. Take $Da_i = \beta$ for simplicity.
2. Now find Sa_i, this indicates the *start index* of the stored adjacent vertices for the vertex V_i within the adjacency list Ea. Take $Sa_i = \alpha$ for simplicity.
3. Starting at the index Ea_β, the next α places in Ea are the indices of the adjacency vertices to the vertex V_i. Formally: $Ea(V_i) = \{Ea_\beta, Ea_{\beta+1}, \ldots, Ea_{\beta+\alpha} \mid Sa_i = \alpha \textit{ and } Da_i = \beta\}$

The clusters formed by DBSCAN on a dataset X can be represented as a graph $G(V, E)$, where each vertex $V_i \in V$ corresponds to the i^{th} data point $\mathbf{x}_i \in X$. An edge e_{ij} connects two vertices v_i and v_j if dist$(\mathbf{x}_i, \mathbf{x}_j) < \varepsilon$. Fig. 5 shows an example of these concepts using a simple graph $G(V, E)$ with $n = 4$.

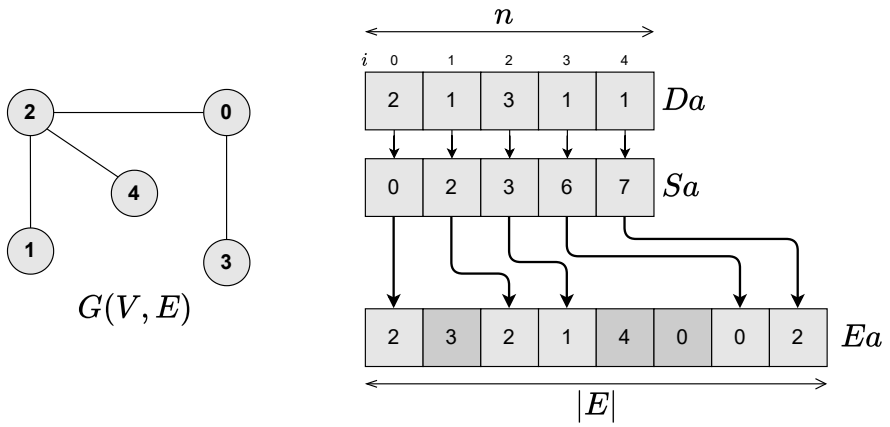


Figure 5: Using the arrays Ea, Da and Sa to represent the graph $G(V, E)$. Adapted from Figure 2 from G-DBSCAN [3] to use this paper's notation.

4.3.1 Finding the Da and Sa arrays

The arrays Da and Sa can be easily derived from the distances matrix $\mathbf{D}$. To find an entry Da_i, simply count the entries of the i^{th}-row of $\mathbf{D}$ that are less than ε. Then, once Da is created, simply perform an exclusive scan [26] of Da to get Sa. This operation can be easily completed using modern GPU-based matrix libraries. The pseudocode to find Da and Sa is shown in Alg. 10.

Algorithm 10 Finding the arrays Da and Sa

1: **Inputs**: n, $\mathbf{D} \in \mathbb{R}^{n \times 2km}$
2: Compute $Da \leftarrow \text{sum}(\mathbf{D} \leq \varepsilon, \text{axis} = 1)$ ▷ Reduction sum across the rows of $\mathbf{D}$
3: Compute $Sa \leftarrow \text{exclusiveScan}(Da)$
4: **return** Da, Sa

4.3.2 Finding the cluster graph adjacency list

Once the arrays Da and Sa are created, the graph adjacency list Ea can be found. For this step, we borrow heavily from the parallel adjacency list construction used by G-DBSCAN [4] - adapting where necessary for our implementation.

Alg. 11 details the pseudo code to construct the adjacency list Ea. The kernel `constructAdjList` is launched with a grid configuration of $\lceil n/256 \rceil$ blocks, each containing 256 threads [4]. In this setup, each thread is assigned

[4]The choice of 256 threads per block was made as it is a multiple of 32. Given the minor time impact of creating the adjacency list (included in the *other* category in Fig. 8), further block size experimentation was not pursued.

a unique index from $[n]$, ensuring parallel processing across all indices. Since CUDA kernels have strict input formats, we use *flattened* row-wise representations of the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$, which are denoted as $\widetilde{\mathbf{A}}$, $\widetilde{\mathbf{B}}$, and $\widetilde{\mathbf{D}}$, respectively.

Algorithm 11 Function to construct the adjacency list Ea

Inputs: $\widetilde{\mathbf{A}} \in \mathbb{R}^{2nk}$, $\widetilde{\mathbf{B}} \in \mathbb{R}^{2Dm}$, $\widetilde{\mathbf{D}} \in \mathbb{R}^{2nkm}$, $Da, Sa \in \mathbb{R}^n$, $n, k, m \in \mathbb{N}$, $\varepsilon \in \mathbb{R}$
  Initialize empty device array Ea with size $Da[n-1] + Sa[n-1]$
  Call `constructAdjlist` to fill values of Ea
return Ea

Define Kernel `constructAdjList`$(\widetilde{\mathbf{A}}, \widetilde{\mathbf{B}}, \widetilde{\mathbf{D}}, Ea, Da, Sa, n, k, m, \varepsilon)$:
  threadIdx $\leftarrow$ blockIdx.x * blockDim.x + threadIdx.x
  currIdx $\leftarrow Sa[tId]$ ▷ Current index within Ea
  nRows $\leftarrow 2 * k * m$

  for each $j \in [0, 1, 2, \ldots, nRows]$ **do**
    if $\widetilde{\mathbf{D}}$[threadIdx $* nRows + j$] $\leq \varepsilon$ **then**
      ACol $\leftarrow \lfloor j/m \rfloor$
      BCol $\leftarrow \mathrm{mod}(j, m)$
      BRow $\leftarrow \widetilde{\mathbf{A}}$[threadIdx $* 2 * k +$ ACol]
      candidateVecIdx $\leftarrow \widetilde{\mathbf{B}}$[BRow $* m +$ BCol]
      Ea[currIdx] $\leftarrow$ candidateVecIdx
      currIdx $\leftarrow$ currIdx $+ 1$
    end if
  end for
End Kernel

An important caveat: With sDBSCAN, the use of random projections can lead to asymmetry in recognising whether two points are within ε of each other. For instance, $\mathbf{x}_i$ may be in the $2km$ candidate neighbourhood of $\mathbf{x}_j$, but not the other way around. As a result, the adjacency list $\mathcal{E}$ may form a directed rather than an undirected graph, where an edge $e_{i \to j}$ links v_i to v_j, but no corresponding edge $e_{j \to i}$ exists. This asymmetry can cause disjoint cluster structures when clusters are later formed. To prevent this, edge symmetry within Ea is enforced, which can be achieved easily using parallel algorithms. However, for the sake of brevity, a detailed discussion is beyond the scope of this paper.

4.3.3 Forming Clusters

Once the adjacency list Ea has been found, the actual formation of clusters can be completed. This is done using a Breadth First Search (BFS) [27] on the CPU. To complete this process, copy the arrays Ea, Da and Sa from device (GPU) to host (CPU) memory. The BFS is done by expanding the frontier only for *core* points (a point $\mathbf{x}_i \in X$ is considered core if $Da_i \geq minPts$). This process is essentially the same as the original clustering approach taken by sDBSCAN [56].

4.4 Using batching techniques

For large-scale datasets, a naïve implementation of CUDA-sDBSCAN may struggle with memory scalability. As an example, consider if the dataset X is MNIST8M [8], with $n \approx 8 \times 10^6$, and $d = 784$. Then let $k = 2$ and $m = 2000$ [5]. Assuming each $\mathbf{x}_i \in X$ is stored as half-precision floats (2 bytes each), then during Alg. 8, the tensor $\mathbf{aNN} \in \mathbb{R}^{n \times 2km \times d}$ has in an approximate byte size shown by equation 4.

[5] Chosen due to their leading accuracy from experiments in sDBSCAN [56]

$$
\begin{aligned}
\text{size}(\mathbf{aNN}) &= |\mathbf{aNN}| \times 2\,B \\
&= (n \times 2km \times d) \times 2\,B \\
&\approx 8 \times 10^6 \times 2 \times 2 \times 784 \times 2\,B \\
&\approx 50,000,000,000\,B \\
&\approx 50\,GB
\end{aligned} \tag{4}
$$

Consequently, the projected size of $\mathbf{aNN}$ makes CUDA-sDBSCAN infeasible for most everyday GPUS - for example, an NVIDIA GeForce RTX 4060 GPU [40] has less than 20GB of dedicated memory. Considering this, we have implemented batching techniques to construct the large matrices (notably $\mathbf{A}, \mathbf{B}, \mathbf{D}$ and $\mathbf{Y}$) sequentially. The batch size parameters can then be tuned so that these matrices satisfy specific GPU memory requirements. Nevertheless, for the sake of brevity, a detailed discussion of CUDA-sDBSCAN's batching techniques is beyond the scope of this paper.

5 Experiment

We run experiments on CUDA-sDBSCAN using large-scale data sets, which are detailed in Table 3. Suggested distance ε values for respective distance metrics were used as suggested by results from sDBSCAN [56]. Table 4 shows the default DBSCAN parameters used for each dataset; unless otherwise stated, assume that these parameters are used.

CUDA-sDBSCAN is compared to the original CPU implementation of sDBSCAN (CPU-sDBSCAN). Due to the absence of other GPU-accelerated DBSCAN implementations suited for high-dimensional data, we only compare CUDA-sDBSCAN to its CPU counterpart. For fair comparisons for the accuracy of CPU-sDBSCAN to CUDA-sDBSCAN, we use the standard sDBSCAN implementation, i.e. the sDBSCAN-1NN implementation [6] is not used. We reached out to Ji et al. for the source code of AC-DBSCAN [30] to run comparative experiments, but we did not receive a response.

We measure the accuracy of both sDBSCAN implementations using Normalised Mutual Information (NMI) [35] to measure the level of shared information between the generated clustered labels and the actual cluster labels.

For most datasets, X is stored using 4-byte floats (float32), while for the memory-intensive MNIST8M dataset, we use 2-byte floats (float16) instead.

When preparing the PAMAP2 dataset, we take a similar approach taken during experimentation from sDBSCAN [56] by dropping any columns with NaN values or the 0 class for transient activities. We removed the heart rate column due to the majority of its values being NaN, along with the timestamp column as well.

Both CPU-sDBSCAN and CUDA-sDBSCAN are written with C++ and compiled with g++ using the **-03** optimisation flag. LibTorch [5] and MatX [9] libraries are used to implement tensor operations. GPU components are executed on an NVIDIA GeForce RTX3090 GPU [41], and CPU parts (for both CUDA-sDBSCAN and CPU-sDBSCAN) with an AMD Ryzen Threadripper 3970X 2.2.GHz 32-core processor (64 threads) with 128GB of DRAM. When comparing CUDA-sDBSCAN to the CPU-sDBSCAN, we use multi-threaded sDBSCAN with 64 threads unless otherwise stated. All parallelisable CPU components of CUDA-sDBSCAN are also executed using 64 threads, using the **#pragma omp parallel** on **for** loops.

Dataset Name	Size (n)	Dimension (d)	#clusters
MNIST [15]	$70,000$	784	10
MNIST8M [8]	$8,100,000$	784	10
PAMAP2 [49]	$1,770,131$	51	18
PAMAP2-100K [49]	$100,000$	51	19
ISOLET [14]	7797	617	26

Table 3: Details of datasets used for experiments. PAMAP2-100K is a 100K sample of the PAMAP2 dataset

[6] See Section 4.5 from sDBSCAN [56]

Dataset Name	minPts	ε (cosine distance)	k	m
MNIST	50	0.11	2	2000
MNIST8M	50	0.16	10	50
PAMAP2	50	0.04	10	50
PAMAP2-100K	50	0.04	2	2000
ISOLET	50	0.12	5	200

Table 4: Default Parameters used for each dataset

5.1 Experimental Summary using Cosine Distance

Table 5 shows a summary of runtime and accuracy results of CUDA-sDBSCAN compared to CPU-sDBSCAN on select datasets mentioned in Table 3. Algorithm parameters are chosen according to Table 4. We also contrast results to the exact DBSCAN clustering accuracy for select datasets using scikit-learn's DBSCAN [17], which is run with multi-threading.

A clear trend from these results is that CUDA-sDBSCAN is faster than CPU-sDBSCAN for most applications. However, for ISOLET, CUDA-sDBSCAN is dramatically slower than CPU-sDBSCAN. We suspect the reason for this is due to the time to copy the dataset from host to device memory, along with the more complex pre-processing steps of CUDA-sDBSCAN. Another thing to note is that CUDA-sDBSCAN has poor accuracy on MNIST8 - the reason for this is yet unknown, although we suspect it is due to a programming bug.

Dataset	**Runtime (seconds)**			**Accuracy (NMI)**		
	CUDA-sDBSCAN	CPU-sDBSCAN	scikit-learn	CUDA-sDBSCAN	CPU-sDBSCAN	scikit-learn
MNIST	9.3	23	64	0.43	0.42	0.43
Pamap	9.9	25.9	N/A	0.34	0.34	N/A
ISOLET	**0.99**	0.11	2.01	0.27	0.28	0.27
MNIST8M	157.2	488	N/A	**0.08**	0.32	N/A

Table 5: Runtime and Accuracy Comparison. Note that scikit-learn cannot run on PAMAP2 and MNIST8M due to memory constraints [56]. Note that we use $\varepsilon = 0.11$ for MNIST with scikit-learn due to accuracy results from Xu et al. [56].

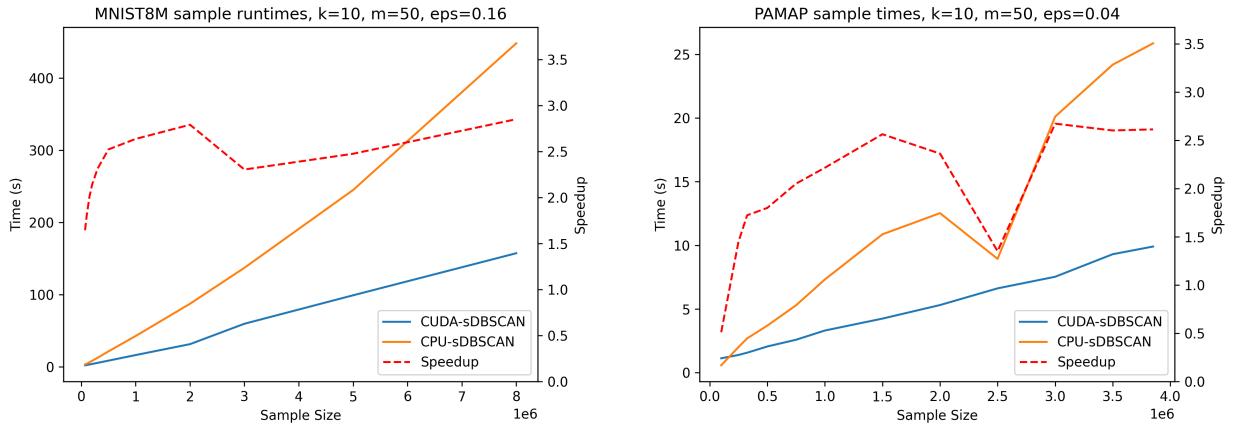
.

5.2 Experimental Study on Scalability with Cosine Distance

CUDA-sDBSCAN is proven to have greater scalability than CPU-sDBSCAN on large-scale datasets. We compare the runtime performance of CUDA-sDBSCAN to CPU-sDBSCAN in Fig. 6 using samples of the PAMAP2 and MNIST datasets. A key trend is that CUDA-sDBSCAN easily outperforms CPU-sDBSCAN for all relatively large sample sizes.

Another observation is that CUDA-sDBSCAN demonstrates significantly *less* speed-up compared to its CPU counterpart than can be theoretically expected. The RTX 3090 GPU has a theoretical performance of 35.58 TFLOPS [7]. Comparatively, the AMD ThreadRipper CPU offers 6.91 TFLOPS, suggesting a potential speed-up of **5.84**×. However, observed speedups, as seen in Fig. 6, averaged at just **2.32**× and **1.99**× for MNIST8M and PAMAP2 samples respectively. Of course, it is not as straightforward as merely comparing the theoretical FLOPS of GPUs and CPUs and presuming an expected speed-up. Other factors, such as memory access speeds, caching mechanisms, and data transfers, are crucial considerations that significantly impact real-world performance [23], [36]. While these factors complicate the ability to predict the full speed-up potential of CUDA-sDBSCAN, it remains unclear how much performance improvement is realistically achievable, leaving room for further optimisation and exploration.

[7] Trillions of Floating Point Operations per Second



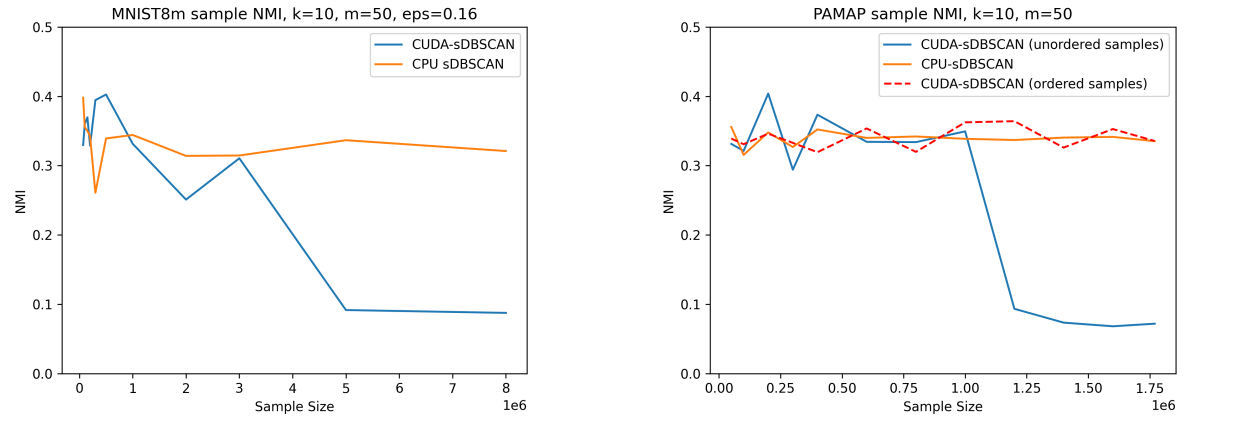
(a) MNIST8M (b) PAMAP2

Figure 6: Comparing CUDA-sDBSCAN and CPU-sDBSCAN runtimes across sample sizes PAMAP2 and MNIST8M

5.3 Experimental Study on Accuracy with Cosine Distance

In terms of NMI accuracy, CUDA-sDBSCAN can yield a performance that is comparable to the original CPU implementation. Figure 7 shows the NMI of both algorithms for the PAMAP and MNIST8M datasets as the sample size is varied. There are issues, however, with the decline of NMI values as the sample size increases for the MNIST8M and PAMAP2 datasets - an effect not seen in CPU-sDBSCAN. The reason for this is unclear; we suspect its due to an implementation error.

Another problem is that CUDA-sDBSCAN seems to be noticeably *variant* to the order in which a dataset is passed into it. As a baseline, we created PAMAP2 samples with no predefined ordering. Next, we made samples sorted in ascending order based on their indices in the original dataset (for example, a sample of $\{\mathbf{x}_2, \mathbf{x}_4, \mathbf{x}_1, \mathbf{x}_3\}$ was reordered as $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$). There is a significant difference in NMI between large unordered and ordered samples. Notably, ordering the PAMAP2 samples appears to *restore* CUDA-sDBSCAN's accuracy to its CPU counterpart. This restoration brings the performance to a level comparable to the inherent variability introduced by the randomised nature of sDBSCAN. You can see the effect of differing NMI accuracy for ordered and unordered samples in Fig. 7b. We suspect this issue is likely attributable to a programming error.



(a) MNIST8M (b) PAMAP2

Figure 7: Comparing CUDA-sDBSCAN and CPU-sDBSCAN accuracy across sample sizes PAMAP2 and MNIST8M

5.4 Analysing Runtimes of Algorithm components

We also analysed the runtimes of the different parts of CUDA-sDBSCAN over different dataset sample sizes. Fig. 8 shows a breakdown of the various components of the algorithm running samples of the MNIST8M and PAMAP2 datasets. It's clear from this figure that the distance computations dominate the algorithm's runtime for large dataset sizes.

Another critical observation is that the time to copy the dataset from CPU to GPU memory is a non-trivial runtime component on small datasets. For example, for a small sample of PAMAP2, this is over 45% of the total runtime. However, as the sample size increases, other components, such as distance computations, easily dominate the CPU to GPU copying time.

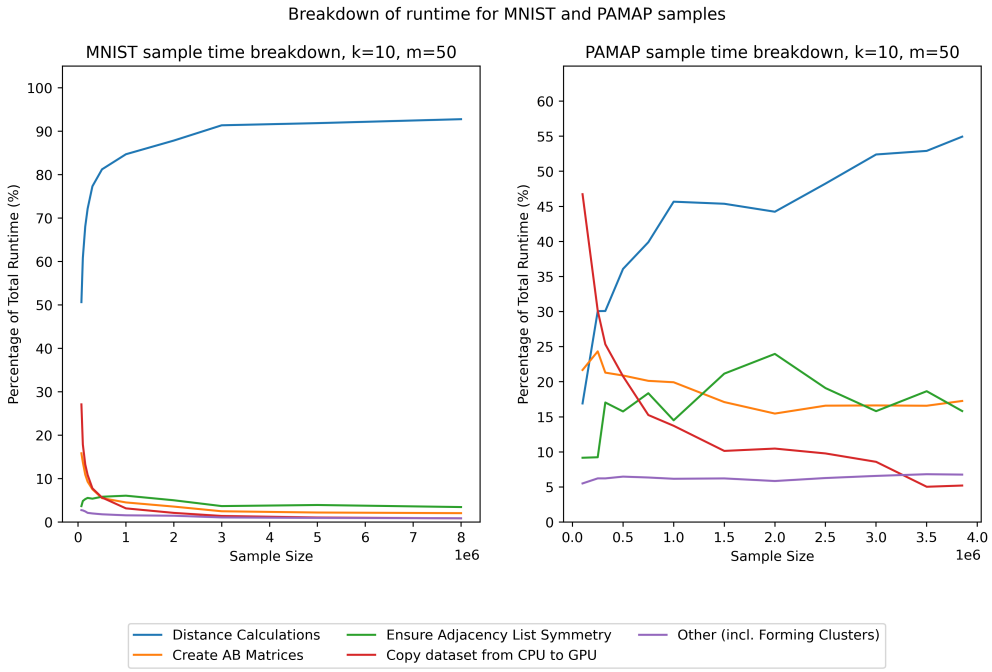


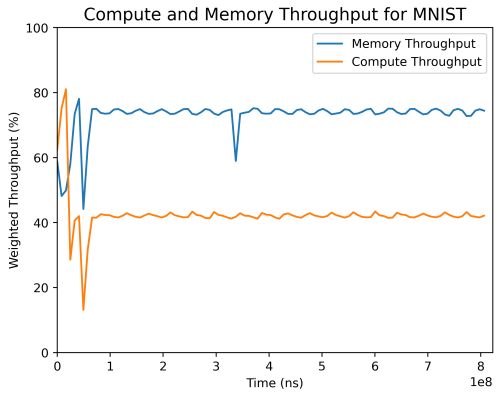
Figure 8: Breakdown of runtime duration per component of CUDA-sDBSCAN

5.5 Analysing CUDA-sDBSCAN with GPU Profiling

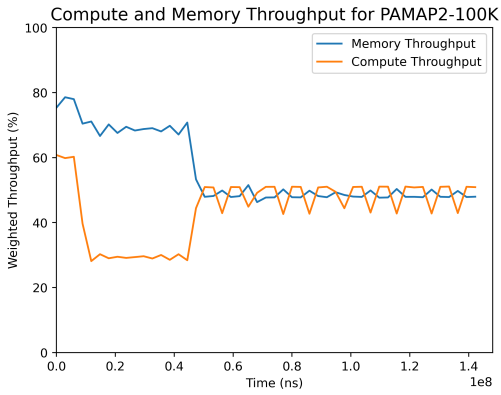
By using the Nsight Compute CLI [42], we can profile the performance of CUDA-sDBSCAN. Fig. 9 shows smoothed graphs for the average GPU memory and compute throughput across the MNIST, PAMAP-100K and ISOLET datasets. Due to how Nsight Compute implements profiling, it exclusively profiles the GPU - not a synthesis of CPU and GPU. A clear trend is the computing power of the GPU is underutilised - the compute throughput tends to fluctuate between 40% and 60% for each dataset. Future work on CUDA-sDBSCAN could focus on increasing its compute throughput - which would predictably lead to greater speed-ups compared to CPU-sDBSCAN.

5.6 Experimenting with different sDBSCAN parameters

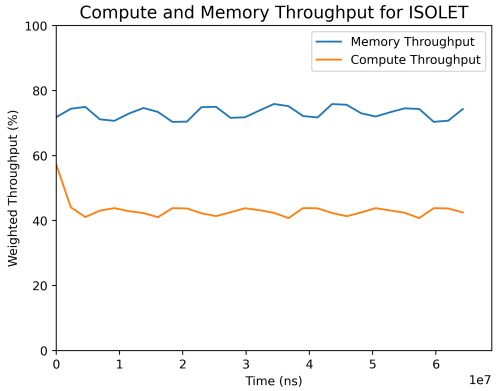
We also investigate the effect of changing sDBSCAN k and m parameters. This was done using a variety of combinations of k and m values on the MNIST and PAMAP-100K datasets. CUDA-sDBSCAN scaled almost linearly as a function of $2km$ (the size of the candidate neighbourhoods per dataset point). Since the distance calculations tend to dominate the runtime of CUDA-sDBSCAN for large-scale datasets (see Fig. 8), the number of distance calculations increases proportionally with the product of k and m, it's, therefore, unsurprising that the overall algorithm scales linearly as a function of $2km$. More detailed results of these experiments are shown in Table 6 within the Appendix.



(a) MNIST

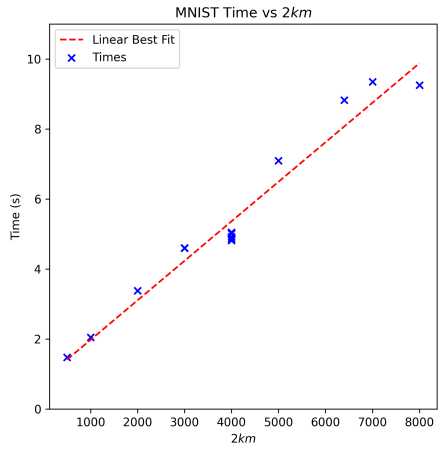


(b) PAMAP2-100K

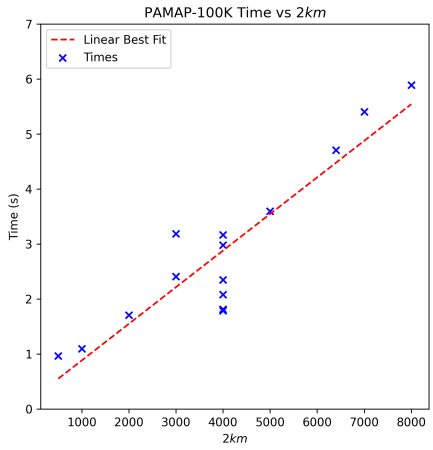


(c) ISOLET

Figure 9: Memory and Compute Throughput of CUDA-sDBSCAN on different datasets. Note for ISOLET, $k = 5$, $m = 50$



(a) MNIST



(b) PAMAP2-100K

Figure 10: Plotting effect of changing $2km$ on the runtime of CUDA-sDBSCAN

6 Future Work

Several different avenues can be explored to improve CUDA-sDBSCAN. Firstly, CUDA-sDBSCAN could be implemented to utilise the GPU's compute capacity better, accelerate the algorithm further, and improve runtimes. This could be done by implementing custom CUDA kernels instead of using library methods to accelerate the distance computations, which were a primary bottleneck to performance, as demonstrated by Fig. 8. One way custom CUDA kernels could offer an improvement is with memory optimisation - for example, algorithm variables could be explicitly stored on the shared memory of SMs, which offers much faster read speeds than global memory.

Additionally, since CUDA-sDBSCAN was primarily written for one GPU model (namely an NVIDIA GeForce RTX 3090), future work could optimise CUDA-sDBSCAN for a large variety of GPU configurations. A vital part of this optimisation could be to find intelligent ways to select the hyper-parameters of CUDA-sDBSCAN - primarily choosing appropriate batching sizes to suit the GPU device that CUDA-sDBSCAN is running on.

Exploring alternative distance measures beyond cosine distance—such as L^1, L^2, χ^2, or Jaccard Similarity could improve CUDA-sDBSCAN. These alternatives can significantly broaden CUDA-sDBSCAN's applicability across diverse use cases - for instance, geo-spatial clustering frequently employs L^2 distance.

7 Conclusion

This paper presented CUDA-sDBSCAN, an approximate DBSCAN algorithm that utilises the computational power of GPU acceleration and random projection techniques. We demonstrated CUDA-sDBSCAN's effectiveness on several large-scale datasets, simultaneously accelerating the foundational CPU-based sDBSCAN while achieving comparable clustering performance. These enhancements make CUDA-sDBSCAN a compelling choice for applications requiring rapid large-scale data clustering.

The experimental results of CUDA-sDBSCAN are encouraging; however, further exploration is necessary to extend its versatility across a broader range of clustering tasks and hardware configurations. Future research could focus on optimisation for diverse GPU architectures, refining its ability to various data distributions and addressing scalability challenges for ultra-large datasets. With further improvements to CUDA-sDBSCAN, we can expand its utility and robustness in real-world applications where efficiency and accuracy are vital.

8 Acknowledgements

I would like to first thank my supervisor, Ninh Pham, for his invaluable guidance, time and support throughout completing my dissertation. Additionally, I'd like to thank my family and friends for their kind words and interest in my project.

A note on using LLMs: Large Language Models (LLMs) were used throughout this project. They were to assist and supplement my efforts across this project. While they never outright replaced my role in this project, they made tasks such as drafting, coding and proofreading much more accessible.

References

[1] Adjacency Matrix, pages 25–44. Springer London, London, 2010.

[2] ACM SIGKDD. 2014 sigkdd test of time award winners, 2014. Accessed: 2024-10-31.

[3] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Oliveira, Renato Ferreira, and Leonardo Rocha. G-DBSCAN: A GPU accelerated algorithm for density-based clustering. 18:369–378.

[4] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Oliveira, Renato Ferreira, and Leonardo Rocha. G-dbscan: A gpu accelerated algorithm for density-based clustering. Procedia Computer Science, 18:369–378, 12 2013.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang,

Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24). ACM, April 2024.

[6] Derya Birant and Alp Kut. St-dbscan: An algorithm for clustering spatial–temporal data. Data & knowledge engineering, 60(1):208–221, 2007.

[7] Thapana Boonchoo, Xiang Ao, Yang Liu, Weizhong Zhao, Fuzhen Zhuang, and Qing He. Grid-based dbscan: Indexing and inference. Pattern Recognition, 90:271–284, 2019.

[8] Léon Bottou. Infimnist. `https://leon.bottou.org/projects/infimnist`, 2016. Accessed: 2024-10-03.

[9] Cliff Burdick, Justin Luitjens, and Adam Thompson. MatX Primitives Library for GPU-Accelerated Numerical Computing in C++, October 2023.

[10] F. Busato and N. Bombieri. Chapter 7 - graph algorithms on gpus. In Hamid Sarbazi-Azad, editor, Advances in GPU Research and Practice, Emerging Trends in Computer Science and Applied Computing, pages 163–198. Morgan Kaufmann, Boston, 2017.

[11] Adil Abdu Bushra and Gangman Yi. Comparative analysis review of pioneering dbscan and successive density-based clustering algorithms. IEEE Access, 9:87918–87935, 2021.

[12] Ricardo Campello, Peer Kröger, Joerg Sander, and Arthur Zimek. Density-based clustering. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 10, 10 2019.

[13] Ricardo J. G. B. Campello, Peer Kröger, Jörg Sander, and Arthur Zimek. Density-based clustering. Wiley Interdisciplinary Reviews. Data Mining and Knowledge Discovery, 10(2), 2020.

[14] Ron Cole and Mark Fanty. ISOLET. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C51G69.

[15] Li Deng. The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6):141–142, 2012.

[16] ELKI Development Team. ELKI: Environment for Developing KDD-Applications Supported by Index-Structures. ELKI Development Team, 2024.

[17] ELKI Development Team. scikit-learn: Machine Learning in Python, 2024.

[18] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD'96, page 226–231. AAAI Press, 1996.

[19] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In Proceedings of the 2015 ACM SIGMOD international conference on management of data, pages 519–530, 2015.

[20] Junhao Gan and Yufei Tao. On the hardness and approximation of euclidean dbscan. ACM Transactions on Database Systems (TODS), 42(3):1–45, 2017.

[21] Nahid Gholizadeh, Hamid Saadatfar, and Nooshin Hanafi. K-DBSCAN: An improved DBSCAN algorithm for big data. 77:1–22.

[22] Michael Gowanlock, Cody M Rude, David M Blair, Justin D Li, and Victor Pankratius. A hybrid approach for optimizing parallel clustering throughput using the gpu. IEEE Transactions on Parallel and Distributed Systems, 30(4):766–777, 2018.

[23] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, pages 134–144. IEEE, 2011.

[24] Ade Gunawan and M de Berg. A faster algorithm for dbscan. Master's thesis, 2013.

[25] Michael Hahsler, Matthew Piekenbrock, Sunil Arya, and David Mount. dbscan: Density Based Clustering of Applications with Noise (DBSCAN) and Related Algorithms, 2023. R package version 1.1-12.

[26] Mark Harris. Parallel prefix sum (scan) with cuda. In GPU Gems 3, chapter 39. Addison-Wesley Professional, 2007.

[27] Jason Holdsworth. The nature of breadth-first search. 02 1999.

[28] IGN Staff. Nvidia geforce rtx 4080 super review. IGN, 2024.

[29] Jennifer Jang and Heinrich Jiang. Dbscan++: Towards fast and scalable density clustering. In International conference on machine learning, pages 3019–3029. PMLR, 2019.

[30] Zhuoran Ji and Cho-Li Wang. Accelerating dbscan algorithm with ai chips for large datasets. In Proceedings of the 50th International Conference on Parallel Processing, pages 1–11, 2021.

[31] Zheng Jian, Guoyan Zhao, Peicong Wang, Xingquan Liu, Mingwei Jiang, Liu Leilei, and Ju Ma. Identification of the mining accidents by a two-step clustering method for the mining-induced seismicity. Frontiers in Earth Science, 12, 03 2024.

[32] Heinrich Jiang, Jennifer Jang, and Jakub Lacki. Faster dbscan via subsampled similarity queries. Advances in Neural Information Processing Systems, 33:22407–22419, 2020.

[33] S. Kannan, S. Karuppusamy, A. Nedunchezhian, P. Venkateshan, P. Wang, N. Bojja, and A. Kejariwal. Chapter 3 - big data analytics for social media. In Rajkumar Buyya, Rodrigo N. Calheiros, and Amir Vahid Dastjerdi, editors, Big Data, pages 63–94. Morgan Kaufmann, 2016.

[34] Kamran Khan, Saif Ur Rehman, Kamran Aziz, Simon Fong, and Sababady Sarasvady. Dbscan: Past, present and future. In The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014), pages 232–238. IEEE, 2014.

[35] Tarald O. Kvålseth. On normalized mutual information: Measure derivations and properties. Entropy, 19(11), 2017.

[36] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In Proceedings of the 37th annual international symposium on Computer architecture, pages 451–460, 2010.

[37] Vani Nagarajan and Milind Kulkarni. Rt-dbscan: Accelerating dbscan using ray tracing hardware. In 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 963–973. IEEE, 2023.

[38] NVIDIA. Using cuda warp-level primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, 2020. Accessed: 2024-10-30.

[39] NVIDIA. Cuda toolkit, 2023. Accessed: June 17, 2024.

[40] NVIDIA. Geforce rtx 4060 & 4060 ti graphics cards. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4060-4060ti/, 2024. Accessed: 2024-10-04.

[41] NVIDIA Corporation. NVIDIA GeForce RTX 3090/3090 Ti Graphics Cards, 2024. Accessed: 2024-10-22.

[42] NVIDIA Corporation. NVIDIA Nsight Compute CLI Documentation, 2024. Accessed: 2024-10-22.

[43] NVIDIA Developer. Cuda refresher: The cuda programming model, 2023. Accessed: 2024-06-19.

[44] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS), 2017.

[45] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. Proceedings of the IEEE, 96(5):879–899, 2008.

[46] Ninh Pham. Simple yet efficient algorithms for maximum inner product search via extreme order statistics. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, KDD '21, page 1339–1347, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Madhav Poudel and Michael Gowanlock. Cuda-dclust+: Revisiting early gpu-accelerated dbscan clustering designs. In 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), pages 354–363. IEEE, 2021.

[48] Andrey Prokopenko, Damien Lebrun-Grandie, and Daniel Arndt. Fast tree-based algorithms for dbscan for low-dimensional data on gpus. In Proceedings of the 52nd International Conference on Parallel Processing, pages 503–512, 2023.

[49] Attila Reiss. PAMAP2 Physical Activity Monitoring. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5NW2H.

[50] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: Why and how you should (still) use dbscan. ACM Trans. Database Syst., 42(3), jul 2017.

[51] Matthias Schubert, Zhanna Melnikova-Albrecht, and Rainer Holzmann. The Weka Workbench. Waikato Environment for Knowledge Analysis, 2023.

[52] TechPowerUp. Nvidia geforce rtx 4080 specifications, 2022. Accessed: 2024-11-08.

[53] P Viswanath and V Suresh Babu. Rough-dbscan: A fast hybrid density based clustering method for large data sets. Pattern Recognition Letters, 30(16):1477–1488, 2009.

[54] Tianyi Wang and Qian Kemao. GPU Acceleration for Optical Measurement. 12 2017.

[55] Chen Xiaoyun, Min Yufang, Zhao Yan, and Wang Ping. Gmdbscan: Multi-density dbscan cluster based on grid. In 2008 IEEE International Conference on e-Business Engineering, pages 780–783. IEEE, 2008.

[56] Haochuan Xu and Ninh Pham. Scalable density-based clustering with random projections, 2024.

[57] Keyu Yang, Yunjun Gao, Rui Ma, Lu Chen, Sai Wu, and Gang Chen. Dbscan-ms: distributed density-based clustering in metric spaces. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1346–1357. IEEE, 2019.

A Appendix

k	m	2km	Time (s)	NMI
5	50	500	1.48	0.28
10	50	1000	2.05	0.30
2	500	2000	3.38	0.37
2	750	3000	4.60	0.40
1	1500	3000	4.60	0.40
20	100	4000	4.82	0.36
40	50	4000	4.87	0.34
10	200	4000	4.92	0.38
1	2000	4000	5.01	0.41
5	400	4000	5.04	0.40
2	1000	4000	5.05	0.41
2	1250	5000	7.10	0.42
2	1600	6400	8.82	0.42
1	3500	7000	9.35	0.42
2	2000	8000	9.25	0.42

(a) MNIST

k	m	2km	Time (s)	NMI
5	50	500	0.97	0.27
10	50	1000	1.10	0.34
2	500	2000	1.70	0.20
2	750	3000	2.41	0.22
1	1500	3000	3.19	0.23
40	50	4000	1.79	0.31
20	100	4000	1.81	0.27
10	200	4000	2.08	0.22
5	400	4000	2.35	0.22
2	1000	4000	2.98	0.24
1	2000	4000	3.17	0.24
2	1250	5000	3.60	0.23
2	1600	6400	4.71	0.04
1	3500	7000	5.40	0.03
2	2000	8000	5.89	0.03

(b) PAMAP-100K

Table 6: Experiment results by varying k and m values for the MNIST and PAMAP100K datasets